UNIVERSITY OF CAPE TOWN

Genetic Programming Applied to RFI Mitigation in Radio Astronomy

by

Kai Staats

A thesis presented for the degree of Master of Science

in the
Department of Applied Mathematics
University of Cape Town

Supervised by Prof. Bruce Bassett, PhD

December 2016

Declaration of Authorship

I, Kai Staats, declare that this thesis titled, "Genetic Programming Applied to RFI Mitigation in Radio Astronomy' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 16 December 2016

"Let the dataset change your mindset."

—Hans Rosling

UNIVERSITY OF CAPE TOWN

Abstract

Department of Applied Mathematics

Masters of Science

by Kai Staats

Genetic Programming is a type of machine learning that employs a stochastic search of a solutions space, genetic operators, a fitness function, and multiple generations of evolved programs to resolve a user-defined task, such as the classification of data.

At the time of this research, the application of machine learning to radio astronomy was relatively new, with a limited number of publications on the subject. Genetic Programming had never been applied, and as such, was a novel approach to this challenging arena. Foundational to this body of research, the application *Karoo GP* was developed in the programming language Python following the fundamentals of tree-based Genetic Programming described in "A Field Guide to Genetic Programming" by Poli, et al.

Karoo GP was tasked with the classification of data points as signal or radio frequency interference (RFI) generated by instruments and machinery which makes challenging astronomers' ability to discern the desired targets. The training data was derived from the output of an observation run of the *KAT-7* radio telescope array built by the South African Square Kilometre Array (SKA-SA). Karoo GP, kNN, and SVM were comparatively employed, the outcome of which provided noteworthy correlations between input parameters, the complexity of the evolved hypotheses, and performance of raw data versus engineered features.

This dissertation includes description of novel approaches to GP, such as upper and lower limits to the size of syntax trees, an auto-scaling multiclass classifier, and a Numpy array element manager. In addition to the research conducted at the SKA-SA, it is described how Karoo GP was applied to fine-tuning parameters of a weather prediction model at the South African Astronomical Observatory (SAAO), to glitch classification at the Laser Interferometer Gravitational-wave Observatory (LIGO), and to astro-particle physics at The Ohio State University.

Acknowledgements

While this work is primarily a demonstration of what I have learned, it is also a testament to the people with whom I have worked. I therefore humbly thank the following individuals for their steadfast commitment to my professional growth and ultimate success.

I am fortunate to have been granted opportunity to study under cosmologist Bruce Bassett who is unique in his interweaving of multiple disciplines, from cosmology to critical thinking, machine learning to meditation. Through his intensive JEDI workshops I was challenged to dive into projects which seemed impossible, yet one week later were achieved. In those hands-on, group settings I gained confidence in an arena wholly new to me, and gained skills that would otherwise encompass an entire semester in a classroom setting.

Colleague Emmanuel Dufourq encouraged me to understand the inner workings of an evolutionary algorithm through my own, hands-on exploration. At the very start, he provided sample Java code which I translated into Python in order to build the original versions of the methods "Root Node", "Function Nodes", and "Terminal Nodes" found in *karoo_gp_base_class.py*. His experience with Genetic Programming helped shape the early days of my coding and interpret the work of the founders of the field. Most of all, his response to my updates "Looks awesomesauce!" were frequently followed by direction to a journal article or important trend.

Postdoc researcher turned co-supervisor, Arun Aniyan embodies a kind of patience and wisdom usually attributed to someone three times his age. While I can attest to his having helped develop my working knowledge of machine learning, it was through our informal conversations that I gained the most from him. On the train and walking from the station to the SKA-SA offices and back again; over pizza, beer, and homemade Indian cuisine he helped me to find comfort in concepts otherwise foreign. Just when I thought I had asked one too many questions, that surely, this time, I had sent him over the edge, Arun would respond, "Hmmm. That's a good question ... let me think about it." He consistently offered explanation that opened yet another door to a new way of seeing the world—and my universe expanded.

Many thanks to Gilad for engaging conversations and memorable banter in the office, the Kruger National Park, even after we were mugged. Thank you Eli for stories of growing up in Namibia, swimming in monster infested rivers, and for your passion for an improved future. Rene, I enjoyed our weekly discussions of chocolate and running marathons.

During the course of this body of research, I had the incredible fortune of working with a number of experienced, talented, and supportive individuals at the Square Kilometre Array, Pinelands Office, South Africa. Thank you Jasper Horrell, Nadeem Oozeer, Laura Richter, Tom Mauch, and the entire SKA-SA team for your support of my research and code development. Just across the river, I had the great pleasure of working with Stephen Potter, head astronomer at the South African Astronomical Observatory (SAAO) for the improvement of his weather prediction model. My weekly conversations with him were intense, challenging, and ultimately rewarding as our work gained some international attention.

With my return to the United States, I was afforded the opportunity to work with The Ohio State University and the University of Mississippi, where I continued the application of Genetic Programming to diverse datasets. At the Ohio State University's Center for Cosmology and Astro-Particle Physics (CCAPP) I was called to co-host a workshop on the application of evolutionary computation to astro-particle physics. Thank you John Beacom, Amy Connolly, Jordan Hanson, Carl Pfendner, and Paul Sutter for your warm welcome and continued support. Over the past three months, I have been welcomed by Marco Cavaglia at the University of Mississippi and more broadly by the LIGO Scientific Collaboration, as Marco and I now lead three of his students in the effort to apply Karoo GP to glitch classification.

I gained tremendously from these experiences, for as the saying goes, "You learn what you know when given the opportunity to teach."

Finally, I thank the anonymous reviewers at UCT who dedicated a significant number of hours in a thorough examination of my draft thesis. Their many pages of notes ultimately led to this, the final and much improved document.

Contents

List of Figures

List of Tables

Dedicated to my father Richard Staats, for all the nights he stayed up late assisting with my maths homework; to my high school physics professor Dan Heim for instilling a lifelong passion for seeing the world through the eyes of scientific observation; and to Gaurav Khanna for guidance that has helped to reshape who I am.

Preface

Note that throughout this document the original Karoo GP multivariate expression notation is maintained, as both operators and operands in each expression are fundamental, engaged elements in the evolutionary process. Therefore, $2 * ch$ is written explicitly (instead of $2ch$) in order to emphasise the operators as elements critical to a multigenerational, evolutionary process.

Chapter 1
In this opening chapter *An Introduction to Radio Astronomy*, I introduce the reader to a variety of instruments that provide the means by which we study the sky, across the electromagnetic spectrum. I include a brief introduction to radio astronomy, from single dish antennae to interferometers.

Chapter 2
In *The SKA-SA and RFI Mitigation* we learn about the prototype KAT-7 array, Radio Frequency Interference, and how we might apply machine learning to flag RFI in radio astronomy data.

Chapter 3
In *An Introduction to Machine Learning*, we learn how complex datasets require the application of machine learning to discover correlations within the data, and how features are derived from raw data to give ML algorithms an improved chance at making those discoveries. This chapter includes a light introduction to select machine learning applications and the means by which hypotheses were evaluated.

Chapter 4
Genetic Programming takes us deeper into a specific form of machine learning, with an under-the-hood look at how GP uses a stochastic search, tournament, and fitness function to evolve a solution.

Chapter 5
Karoo GP introduces us to the original Python application developed during the course of this MSc research. I introduced to the workflow, the user interface, and three toy models included with Karoo GP and how each engages a unique fitness function.

Chapter 6
Machine Learning Applied to KAT-7 Data describes the datasets, procedures, and results of one month of data runs at the Square Kilometre Array, South Africa. This concludes the research conducted with the SKA-SA during the course of this MSc.

Chapter 1

An Introduction to Radio Astronomy

1.1 From Ancient Times

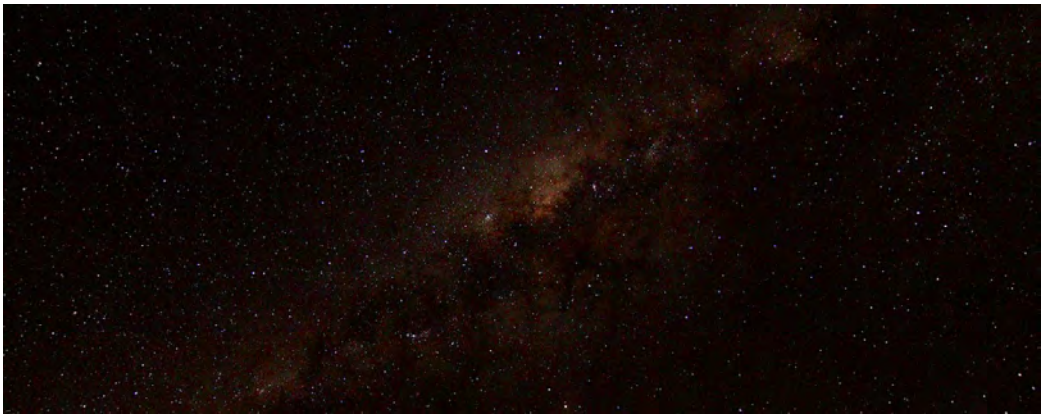


FIGURE 1.1: The Galactic Center, SAAO site, Sutherland, South Africa.

From ancient times, we have looked to the heavens in an attempt to discern our future, to find our way. The patterns in the stars have worked to guide seafaring craft across vast expanses of water and to guide farmers for the planting and harvesting of crops against the backdrop of changing seasons.

As a species keenly adapted at pattern recognition, our interest in the cosmos has relatively recently grown from a perceived, personal relationship with the stars overhead to one of looking deep into the history of the universe, at events which occurred millions, even billions of years ago. While we have learned that indeed our presence is incredibly insignificant in the scope of the cosmos, our effort to learn its inner workings has only begun.

The tools we employ have evolved to support our desire to gather more information, to more clearly resolve the data. In order to peer further into the past, our instruments demand higher image resolution and computational power.

Where once the naked eye was the tool for monitoring the motion of the planets and stars, the optical telescope granted astronomers magnification to resolve distant objects, to study not only the motion of a point of light, but to see detail in its features. With optical telescopes, we are seeing the universe in a relatively narrow band referred to as the *visible spectrum*, as shown in Figure 1.2.

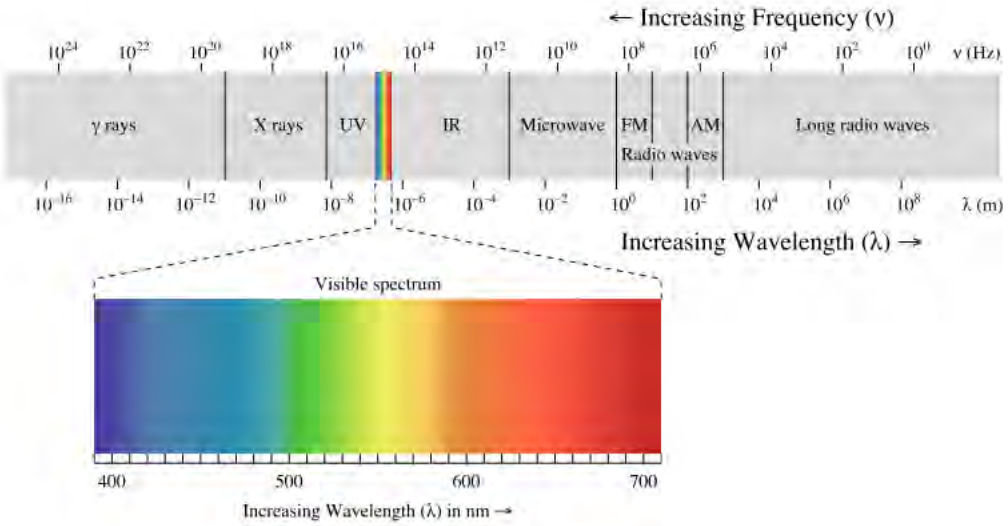


FIGURE 1.2: The electromagnetic spectrum [P. Ronan, Creative Commons].

Spectroscopy provides the capability to split visible light into its constituent wavelengths, exposing the unique signatures of the elements involved in stellar evolution and ultimately, the chemical composition of planets orbiting distant stars. Modern astronomy employs various instruments, each of which focuses on a portion of the electromagnetic spectrum, from high energy gamma rays and x-rays to visible light, from infrared to microwave to radio waves in the long end of the spectrum.

Each portion of the spectrum reveals a unique view of the cosmos. Some objects which appear nearly or fully invisible in visible light provide a breathtaking view of the complex chemistry or physics at other wavelengths.

1.2 The Birth of Radio Astronomy

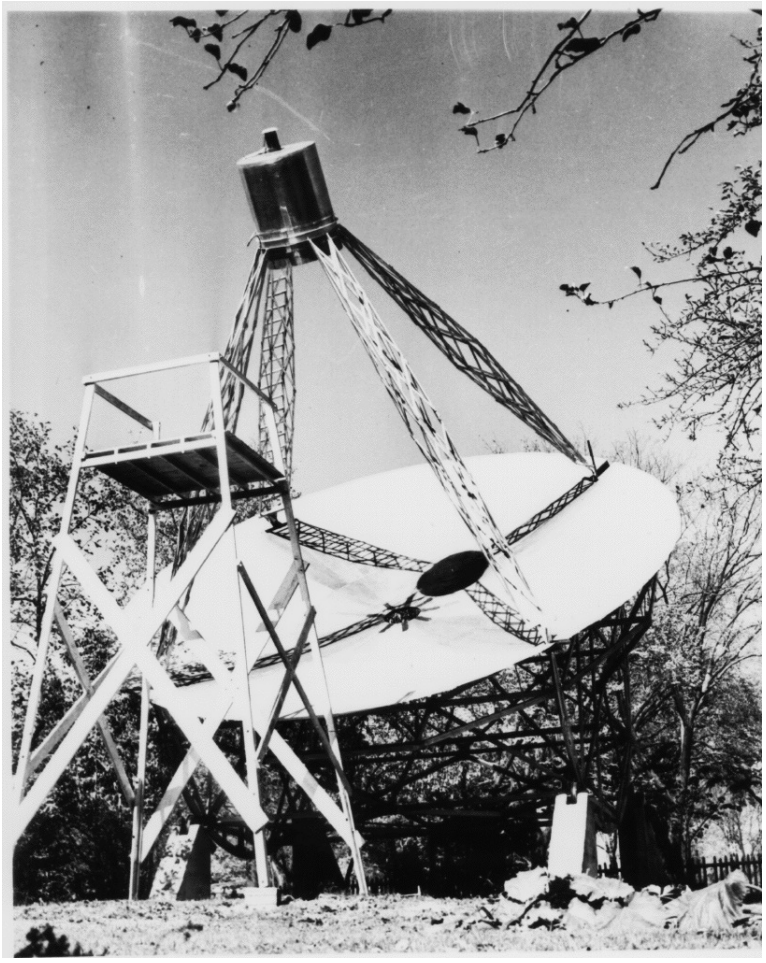


FIGURE 1.3: The first dish-based radio telescope, 1937 [G. Reber, Creative Commons].

In an attempt to locate and better understand sources of noise in the radio portion of the electromagnetic spectrum, Bell Telephone Laboratories employee Karl Jansky [4] was in the early 1930s engaged in the construction of the first radio telescope (Figure 1.3). He was successful in identifying two sources of noise which had been interfering with radio transmissions, but the third remained a mystery for it appeared to come from the central region of our Milky Way galaxy, in the constellation of Sagittarius [5].

Shortly thereafter, Grote Reber built the world's first radio telescope in the shape of a dish, increasing the power through concentration of a large antenna to a smaller focal point which contained a radio wave receiver [6].

Since those early days, the devices used to receive the relatively weak radio signals generated by distant sources have evolved tremendously. Yet the intent remains the same—to expand our understanding of the evolution of the universe. With larger, more powerful instruments, we are able to see further back in time with greater clarity. As we

improve our techniques for acquiring and filtering the data, we also improve the quantity and quality of information.

However, as human populations have grown across the face of the planet, the sky has become polluted with noise across much of the electromagnetic spectrum. Street and stadium lights limit one's ability to see the night sky, while modern machines and communication devices produce powerful radio signals which interfere with or override the relatively weak signals from distant, cosmic sources.

1.3 The Limitations of a Single Antenna

The need for higher resolution imaging has been driven by the increased demand for more data about distant objects. More data provides more information with which we are better able to understand the processes by which stars form and die, the expansion of the universe, and the formation of super-massive black holes.

In order to gain higher quality images, astronomers seek both improved sensitivity and resolution. This requires, in part, instruments capable of capturing the maximum possible number of photons. The total number of photons, or the apparent brightness of an object to the viewer is a function of the total surface area of the telescope's receiving dish. In both optical and radio astronomy, there are physical limits to the size of a single dish due to gravitational sag, differential heating, and in the case of radio astronomy whose dishes are exposed to sun, rain, snow, and torque caused by wind. In radio astronomy, this limit is approximately 100 meters [7].

One solution is an immobile radio antenna given permanent foundation on the earth such that the dish itself does not move. This kind of telescope scans the sky according to what is visible nearly directly overhead, where a suspended receiver moves with respect to the focus of the dish, providing an adjustable view of the otherwise fixed window to the sky.

Arecibo in Puerto Rico was the first radio telescope of this kind at 305 meters in diameter [1]. China is now constructing the hybrid radio dish *FAST*, the largest in the world at 500 meters in diameter. This will combine a fixed foundation with the ability to change the shape of a smaller, 300 meter diameter portion of the dish.

To be a valuable astronomical tool, a radio telescope must follow a source within $\sigma \approx \theta/10$ arc-radians to provide accurate photometry, or imaging [1]. As the angular resolution of a diffraction-limited telescope is $\theta \approx \lambda/D$ radians, construction of a single

[1] National Astronomy and Ionosphere Center (NAIC)

dish large enough to achieve sub-arcsecond resolution at radio wavelengths is physically impossible.

Therefore, use of two or more, physically separated radio telescope antennae enables the desired resolution. Multiple antennae radio telescope arrays do not require that all dishes focus their light onto a single detector, rather, each antennae works independently, the data-stream combined in real-time by means of radio interferometry.

1.4 The Radio Interferometer

FIGURE 1.4: The Very Large Array (VLA), Socorro, New Mexico.

Developed by British radio astronomer Martin Ryle, Australian engineer, radiophysicist, and radio astronomer Joseph Lade Pawsey, and Ruby Payne-Scott in 1946, radio interferometry provides a means to achieve high resolution imaging in radio astronomy [8]. The use of multiple, smaller antennae whose data streams are combined off-detector enables employment of a highly sensitive instrument which is both technically and financially feasible.

Modern radio interferometers are composed of two or more widely separated radio telescope antennae trained on the same object. These multiple, interconnected instruments together increase the total signal collected and employ *aperture synthesis* to dramatically increase resolution. This technique uses superimposed (interfering) signal waves from any two telescopes on the principle that two in-phase (coincident) waves will add to each other, while two waves with opposing phases will cancel each other out. This creates a single telescope with resolution related to the distance between the two most distant antennae pair in the array.

In order to produce a high quality image, various combinations of pairs of separated telescopes, referred to as a baseline, are required in order to achieve a high quality image. For example the Very Large Array (Figure 1.4) has 27 telescopes providing 351 independent baselines [9].

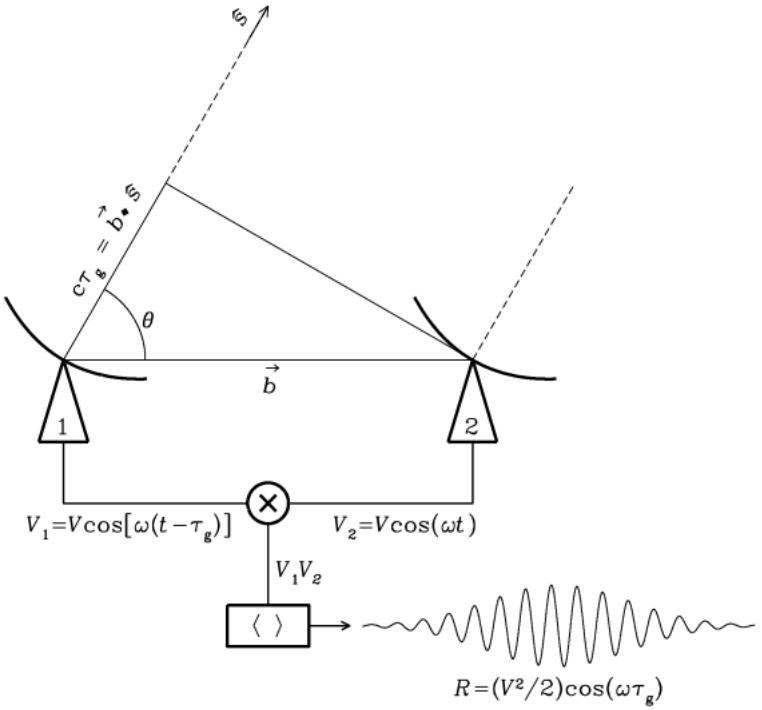


FIGURE 1.5: A 2-element interferometer [1].

The following is based upon "Essential Radio Astronomy" by James J. Condon and Scott M. Ransom, NRAO [1].

Given two or more ($N \geq 2$) radio antennae configured to function as an interferometer, $N(N-1)/2$ is the number of unique baseline pairs. As shown in Figure 1.5, the simplest interferometer is composed of two identical dishes separated by a baseline vector of length $\overrightarrow{b}$. Both point to the same, distant point source, in the direction specified by unit vector $\hat{s}$.

Plane waves from a distant point source travel an extra distance $\overrightarrow{b} \cdot \hat{S} = b\cos\theta$ to reach antenna 1 in relation to antenna 2. The output voltage (V_1) of antenna 1 is the same as that of antenna 2, but it lags in time by the geometric delay $\tau_g = \overrightarrow{b} \cdot \hat{S}/c$.

To compensate for the voltage lag of antenna 1, the correlator multiplies and time averages the voltage output from each antennae. This yields an output response with amplitude R that is proportional to the point-source flux density, whose phase depends on the delay and the frequency.

As shown, the quasi-sinusoidal output fringe occurs if the source direction in the interferometer frame is changing at a constant rate $d\theta/dt$.

Consider a quasi-monochromatic interferometer, one that responds only to radiation in a very narrow band centred on frequency $\nu = \omega/2\pi$. Accordingly, the output voltages

of antennae 1 and 2 can be written as:

$$V_1 = V\cos[\omega(t-\tau_g)] \text{ and } V_2 = V\cos(\omega t) \tag{1.1}$$

The correlator first multiplies these two voltages to yield the product:

$$V_1V_2 = V^2\cos(\omega t)\cos[\omega(t-\tau_g)] = \left(\frac{V^2}{2}\right)[\cos(2\omega t - \omega\tau_g) + \cos(\omega\tau_g)] \tag{1.2}$$

and then applies a time average $[\triangle t \gg (2\omega)^{-1}]$ long enough to remove the high-frequency term $cos(2\omega t - \omega\tau_g)$ from the output R:

$$R = \langle V_1V_2\rangle = \left(\frac{V^2}{2}\right)\cos(\omega\tau_g) \tag{1.3}$$

The amplitudes V_1 and V_2 are proportional to the electric field produced by the source multiplied by the voltage gains of antennae 1 and 2. Therefore, the output amplitude $V^2/2$ is proportional to the point-source flux density σ multiplied by $(A_1A_2)^{1/2}$, where A_1 and A_2 are the effective collecting areas of the two antennae.

In realworld radio astronomy, absolute positions with errors as small as $\sigma_\theta \approx 10^{-3}$ arcsec and differential positions with errors down to $\sigma \approx 10^{-5}$ $arcsec < 10^{-10}$ radians have frequently been measured. As such, radio interferometer arrays can determine the positions of compact radio sources with unmatched accuracy.

In this opening chapter we have traced the evolution of radio astronomy from its beginnings to the present, from relatively simple, single antenna to multiple antennae arrays. Given the increase in computing power, future evolution of radio astronomy will be driven as much by software as hardware. In the next chapter, we'll examine how this is unfolding now at the Square Kilometre Array in South Africa.

Chapter 2

The SKA-SA and RFI Mitigation

FIGURE 2.1: Future vision of the Square Kilometre Array [SKA, South Africa].

2.1 The Square Kilometre Array, South Africa

The Square Kilometre Array (SKA) is a multi-nation effort to build the world's largest radio telescope array. When complete, it will encompass more than a square kilometre (one million square metres) of collecting area. The SKA represents a tremendous advance in engineering, research and development to construct a unique scientific instrument. As one of the largest scientific endeavours in history, the SKA is bringing together the world's finest engineers and scientists [10].

The South African SKA Project (SKA-SA) is a business unit of the National Research Foundation of South Africa and is charged with hosting the international SKA project in South Africa. SKA-SA is funded by the (South African) Department of Science and Technology. SKA-SA also plays the lead role in bringing in the other African Partner Countries for the SKA.

Located in the Karoo, a semi-desert region of South Africa, and Western Australia's Murchison Shire, the Square Kilometre Array takes advantage of the relative radio silence of the region. SKA-SA has built and operates the KAT-7 array (Section 2.1.1), from which the data employed in the body of this research was generated, and is currently

constructing the next generation MeerKAT array. SKA-SA owns and operates the Karoo telescope site on which these and the future instruments will be located.

The SKA-SA will host the mid-frequency components while Australia's Murchison Shire will host the low frequency components of this intercontinental array. As described in Section 1.4, the interferometer enables pairs of antennae to be widely distributed such that their baselines may be spread across continents, increasing the total resolution of the instrument.

When complete, the SKA will enable astronomers to survey the sky far faster ($10^6\times$) than any system currently available, and image the sky in sensitivity ($100\times$) and detail never before achieved, with resolution surpassing that of the Hubble Space Telescope [10].

2.1.1 KAT-7

FIGURE 2.2: The KAT-7 Array [SKA, South Africa].

KAT-7 (Figure 2.2) is an engineering prototype for the MeerKAT array, which when completed will be composed of $64\times$ 13.5 metre diameter antennae [1]. Construction of the seven 12 metre KAT-7 antennae was completed with the close of 2010, operated through 2015 in an engineering test mode, and became fully operational in 2016.

The frequency range for the KAT-7 dishes is tunable within the 1200-1950 MHz range, with an instantaneous bandwidth of 256 MHz. This is the total bandwidth within which the dishes can operate for any given observation run. Within the given 256 MHz the dishes can be configured to capture a range of discreet channels, each of which has its own frequency range. The sum of all the channels always equals the total instantaneous bandwidth such that they cover the full 256 MHz without gaps [11].

[1] public.ska.ac.za/meerkat

2.2 Radio Frequency Interference

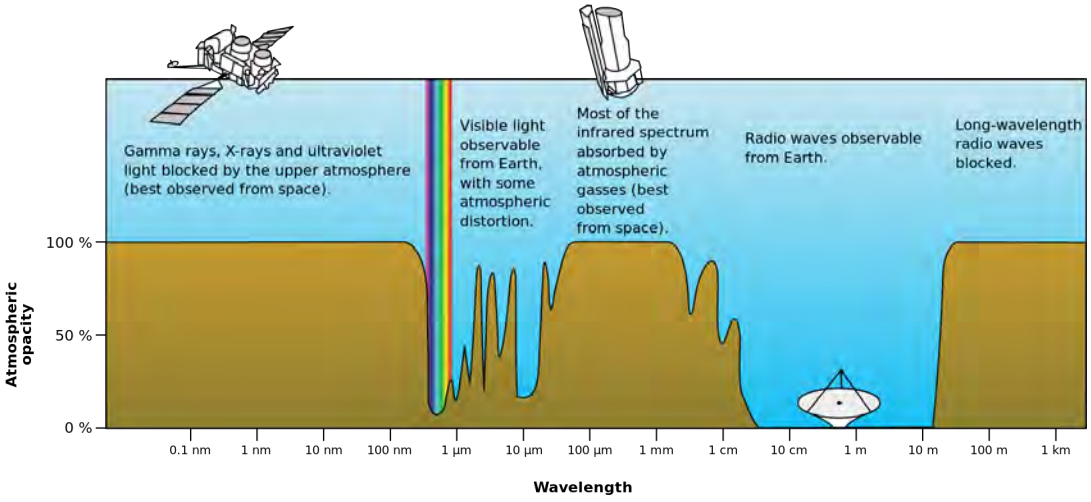


FIGURE 2.3: The Earth's atmosphere and the electromagnetic spectrum [NASA, Creative Commons].

Radio Frequency Interference, or RFI, is electromagnetic noise generated principally by human-made machines and electronic devices. RFI can be considerably stronger than the desired, relatively weak radio emissions from cosmic sources. RFI can overload the sensitive receivers and cause errors in signal calibration. RFI can appear in the same channel as the desired astronomical signal, causing ambiguity for the astronomer [12]. Because RFI interferes with astronomers' ability to derive valuable information from the data collected, radio astronomy must employ RFI mitigation strategies unique to each observatory.

RFI typically falls into one of two categories: *channelised* and *broadband*. Channelised RFI is typically found in only a few, discrete channels (as defined by the radio antenna's given configuration) over a long period of time. Broadband RFI is found across many channels for a brief period of time (milliseconds) [13].

Ground-based radio astronomy is limited by attenuation in the Earth's atmosphere, as shown in Figure 2.3, with a lower and upper limit. Due to the free electrons in the ionosphere, the atmosphere reflects radio waves below 30 MHz back into space. Due to gas molecule attenuation, the upper limit is above 300 GHz. Yet even within this relatively narrow band of the electromagnetic spectrum, television and radio broadcast, radar, communication devices, motors, personal electronics, aeroplanes and satellites in Earth orbit all severely interfere with the highly sensitive function of radio astronomy receivers [14].

For international radio astronomy projects such as the Very Long Baseline Interferometer (VLBI), it is imperative that all countries participating protect the same frequency bands in order that all participating observatories are working with comparable data. The International Telecommunication Union (ITU) governs the radio frequency spectrum worldwide, as presented in the ITU-R RA.769-2 "Protection criteria used for radio astronomical measurements" [14]. Specific to the SKA and the Karoo, the South African government passed the "GA Act" on June 11, 2008. This law provides for the preservation and protection of areas within the country of South Africa that are uniquely suited for optical and radio astronomy [15].

Protection of certain frequency bands used in radio astronomy assists ground-based radio astronomy to function with as little RFI as possible. However, radio-quiet zones limit but do not eliminate interference altogether. Even in relatively isolated regions, RFI continues to be produced by air traffic, satellite transmissions, and as transient electromagnetic radiation generated by passing vehicles and wind turbines [16].

2.2.1 RFI Mitigation

F_{low} [MHz]	F_{high} [MHz]	Source	Description
925	960	Terrestrial GSM towers	Sporadic in both time and frequency occupancy. Will be reduced over time as alternative communications system is deployed. Currently 925-935 MHz is unoccupied.
1085	1095	Airborne SSR radar	Persistent, but variable, during hours of air traffic (~06h00-23h00).
1082	1150	Airborne DME interrogators	Sparse frequency occupancy (~5 visible at any one time), narrow band (<1 MHz) signals, variable time occupancy during hours of air traffic.
1164	1300	GNSS Satellites	Entire band occupied all of the time.
1467	1492	WorldSpace Satellite	Strong and persistent. Does not occupy entire band. May be decommissioned in the future.
1525	1610	GNSS and Inmarsat Satellites	Strong and persistent.
1616	1626.5	Iridium Satellite	Persistent.

FIGURE 2.4: Frequency band, source, and description of known RFI sources with MeerKAT, 2015 [N. Oozer, SKA-SA].

As with optical astronomy, the first step in RFI mitigation is selecting a site far removed from the sources of terrestrial noise. However, there are electronic devices located at radio astronomy sites which themselves generate the very RFI that is problematic for

radio astronomy. Examples of these sources measured for MeerKAT (2015) include the minimum and maximum frequencies, source, and description, as shown in Figure 2.4. While mobile phones, laptops, and vehicles can be switched off, air conditioning and computers, for example, must be in operation during observation runs. The impact of these local RFI sources depends heavily upon the design of the infrastructure of the observatory and the shielding of the equipment.

For example, at the SKA-SA's Karoo observatory, the radio antennae are far from the buildings in which the observers live and work. Earthen mass surrounds the facilities while subterranean buildings help absorb RFI such that the effect is limited. The data centre which hosts the on-site computing equipment is very carefully shielded by a steel enclosure, essentially a large Faraday cage. Furthermore, as radio dish receivers are not often pointed below 20 degrees from the horizon due to the high attenuation by the atmosphere, the effect of terrestrial RFI is again reduced.

However, due to the high power and close proximity to local RFI sources, they remain disruptive to observations and the data collected. For example, terrestrial RFI can leak into the receiver through the side lobes of the antenna pattern. Strong RFI signals can penetrate shielded cabling and corrupt the observed channel. Most challenging of all, RFI generated in the desired, observed signal or nearby bands is highly problematic as it will pass through the bandpass filters of the receivers [14].

Each radio telescope incorporates built-in countermeasures to filter the RFI generated by the telescope instruments itself. In addition, there are mechanical and software-based systems designed to filter the most recognisable, external RFI. One method is to block an entire frequency band, for example to filter 90-110 MHz which thereby blocks all commercial FM radio stations. However, this removes *all* data that may have otherwise been observed in that frequency domain, both RFI and the desired signal [16].

Another method is the application of software designed to isolate RFI, either in the real-time pipeline or in post-processing of the data. This process is referred to as *data flagging* in which the data points believed to be associated with RFI are marked but not removed from the dataset. Each astronomer may then elect to enable or disable the flags in any given dataset.

Currently, astronomers use one or more flagging applications such as *AOFlagger*. AOFlagger employs several methods to discover and flag RFI: a sum of thresholds; smoothing, sliding window, median and high-pass filters, and morphological operators. These are combined and applied as a single strategy which can be successfully employed by almost any radio astronomy observation. AOFlagger can be set to run automatically or to engage one image at a time with user configurations applied [17]. An example of data

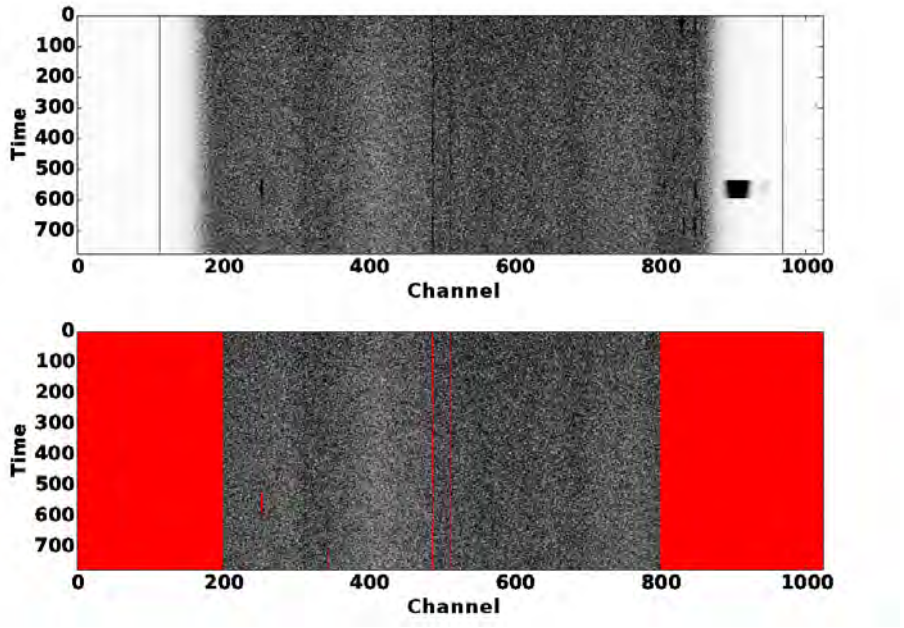


FIGURE 2.5: In a Channel (frequency) versus Time image constructed from the Fourier transform of the visibility plane [top], SKA-SA KAT-7 data is depicted without RFI flagging. The same data as presented is presented again [bottom], but the areas painted red have been flagged as RFI by the application AOFlagger [T. Mauch, SKA-SA].

from the SKA-SA KAT-7 array both before and after the application of AOFlagger is shown in Figure 2.5.

Data flagging is not, however, without consequence, for it is an imperfect methodology resulting in lost data. One measure of the accuracy of a given data flagging method is the average number of false-positive detections of RFI versus the average number of true-positives. It is important that RFI detection and flagging be conducted at a high time and frequency resolution in order to increase accuracy and decrease the loss of data [16].

With the KAT-7 array, RFI flagging is incorporated into the pipeline in three stages, as shown in the *KAT-7 Pipeline* below (Section 2.2.2): a) single pass in the *ingester*; b) an iterative pass incorporated with the *calibration* phase; and c) the application of one or more software packages by the astronomer, to conduct the final flagging.

2.2.2 KAT-7 Pipeline

1. Signal arrives at dish
2. RF signal detected in feed (H and V channels)
3. Low noise amplification
    - RF over fibre link back to the ADs and correlator
4. Analogue to digital conversion
5. Correlator:
    (a) Input frequency channelisation
    (b) Signal transposed
    (c) Cross-multiplication and averaging (correlator dumps output at configurable rate depending on array size and type of science, typically a few times per sec)
    (d) Output *visibilities* (HH,VV,HV,VH polarities) or Stokes parameters (I,Q,U,V) for each baseline
    (e) Timestamp
    (f) Frequency channel (configurable 1024 for KAT-7, 8192 for MeerKAT)
6. Ingester (Science Processor):
    - Preliminary RFI flagging (automated)
    - Data saved to HDF5
7. Calibration: Build the best model of the sky possible given noisy data and unknown telescope parameters
    - An iterative process removes artefacts and the effects of the *point spread function* (since the full UV plane is not sampled)
    - Processing repeatedly switches between the *visibility plane*, in which the interferometer measures the sky through its baselines, etc., and the *image plane*, the Fourier Transform of the visibility data, producing a human-readable image
    - Calibrate the various time and frequency dependent (complex *amp* and *phase*) gains in the system
8. Iterative flagging: Flagging happens primarily during this step, with up-front, high-speed calibration at ingest to the science data processor (above)

- RFLAG [2]
- MIRFLAG [3]
- PGFLAG [4]
- AOFLAGGER [17]

9. Imaging by Astronomers

Following calibration the ingester conducts preliminary, automated flagging. With automated, iterative, multi-pass flagging, data is calibrated, imaged, and flagged repeatedly. Each pass improves the quality of the data by removing the unwanted artefacts and RFI prior to the data being made available to the astronomers.

While these presets do remove the obvious, total bandwidth or erroneous, continuous time-line RFI data points, most flagging continues to be conducted by hand following the KAT-7 pipeline. This is time consuming and resource intensive, and therefore does not scale well [18].

2.3 MeerKAT Data

When the full 64 antennae MeerKAT comes online, and in the next decade the complete SKA-SA array, the quantity of data generated will render the current techniques inadequate. According to the SKA document *SKA1 SYSTEM BASELINE DESIGN, revision 001, 2013* [19], the proposed SKA1 and MeerKAT array with 190 SKA1-mid dishes and 64 MeerKAT dishes combined will generate a total input data rate for the Science Computing System (Stage 6 of Section 2.2.2) of 162—3250 Gigabytes per second for the configurations of 10—200 km max baselines respectively. Combined, the arrays will generate 0.50—10 petabytes of image data *per day*, as much as 3,000 petabytes of data per year.

Clearly, this level of data processing must be managed by a reliable, accurate processing system which does not require continuous human intervention nor validation or additional flagging prior to working the data.

It is imperative that more sophisticated, automated systems be developed such that computers are entrusted with the process of removing both artefacts and RFI, freeing the astronomers to work with clean images.

[2] casa.nrao.edu/docs/UserMan/UserMansu167.html
[3] www.atnf.csiro.au/computing/software/miriad/doc/mirflag.html
[4] www.atnf.csiro.au/computing/software/miriad/doc/pgflag.html

The ultimate goal of this and related bodies of research at the SKA-SA office in Pinelands, South Africa is to develop an automated system which performs better than the current combination of automated, iterative flagging in the KAT-7 pipeline and subsequent manual confirmation and often, further manual application of flagging applications [20].

Furthermore, it is desired that the application of machine learning might discover otherwise overlooked, interesting correlations in the data that lead to the discovery of new objects in the sky.

2.4 Machine Learning Applied to RFI

Traditionally, the final stage of the pipeline, RFI flagging, has been a manual or semi-automated process wherein the astronomer applies data flagging techniques unique to his or her own research. However, with radio astronomy now pressing into the realm of "big data", it is no longer possible to apply manual data flagging techniques for the speed at which a human can review the data is far too slow for the tremendous data flow.

This is the entry point for machine learning (formally defined in Chapter 3), a means to automate the flagging of RFI with equal or improved accuracy and at exponentially larger processing rates over that of the human. Various efforts have been made in this realm at several radio astronomy sites.

For example, in "A Machine Learning Classifier for Fast Radio Burst Detection at the VLBA" Kiri et al describe a machine learning classifier that identifies RFI versus a potential new discovery. This algorithm has afforded a decrease in time spent reviewing data and an increase in interesting candidates observed as it filters 80%—90% of the candidates with greater than 98% accuracy. Astronomers are then able to focus on 10%—20% of the total candidates observed [20]. PhD. candidate CJ Wolfaardt, Stellenbosch University, South Africa, investigates various methods to automatically classify RFI signals, including a Gaussian Mixture Model (GMM) and K-nearest neighbors (KNN) in "Machine learning approach to radio frequency interference(RFI) classification in Radio Astronomy" [21]. In "Radio Frequency Interference mitigation using deep convolutional neural networks", Akeret et al apply a *Convolutional Neural Network* (CNN), an advanced form of Artificial Neural Network (Section 3.4.1). This CNN classifies signal versus RFI in 2-dimensional, time-ordered data radio telescope data [22].

In 2012, Gary Doran at the NASA Jet Propulsion Laboratory, California Institute of Technology, applied supervised learning to the characterisation of RFI to a dataset produced at the Parkes Multibeam Pulsar Observatory in Australia [13]. His research and

application was focused on post-correlation data processing, that is, automated flagging of RFI in the data, not the pipeline. Working with 3.5 TB data from approximately 1000 hours observations over 4.5 years, Doran amassed 12,000 sets of 13 files for each beam.

His work does not apply a time-frequency threshold analysis, a simple means of filtering data by setting a threshold above or below which all data is flagged or removed; rather, the generation of features as follows:

- Intensity average or time average of frequency
- Time of day (represented as points on a circle)
- Direction antennae is pointing (azimuth, zenith) measured in degrees

At the time of his research Doran states there were no prior efforts to automatically categorise transient RFI events, and as such, there was no database from which to draw class labels for supervised machine learning (Chapter 3). Instead, Doran applied the unsupervised algorithm *Mini-Batch k-means* from the Python-based *scikit-learn* library.

Using these techniques, Doran processed more than 5.3 million RFI events, with approximately 3.4 million (65%) lasting only a single time sample ($125\mu s$), with an average event duration of 15.7 time samples, or 2ms. Some basic analysis showed a clear correlation between pointing the telescope toward the visitor centre, parking lot, cafe, and generator and an increase in RFI.

Cluster analysis with k-means generated cluster sizes from 1-3% of all events, with approximately 2/3 of the clusters exhibiting patterns related to average event intensity, time of day, and telescope pointing direction [13].

Chapter 3

An Introduction to Machine Learning

3.1 A Propensity for Pattern Recognition

We are a species that excels at pattern recognition. We depend upon pattern recognition for our very survival, from cloud formations to the migration of game to the motion of celestial bodies helping us to predict the change of the seasons. The behaviour of predator and prey, even that of our own human companions demands that we seek, understand, and act upon recognisable patterns.

Some patterns are discerned at an early age. Language, reading facial expressions and body language, the manipulation of objects in a field of gravity—all are innate to the development of our young. Where we once attributed the patterns we observed to our relationship with the supernatural, scientific observation and measurement of the natural world has helped us to better understand the underlying processes and associated patterns. These observations are analysed by means of empirical data [23].

Generally speaking, data is generated by two kinds of science: hypothesis-driven and data-driven science [2]. Hypothesis-driven science works to support or falsify a hypothesis and is governed by the parameters of the experiment. Data-driven science collects and then analyses data in order to discover patterns previously unknown.

All fields of modern research, including chemistry, biology, botany, sociology, and cosmology embrace both approaches. But with the close of the first decade of this century, a rapidly increasing capacity for processing larger datasets has seen the generation of scientific data grow exponentially [2], as depicted in Figure 3.1.

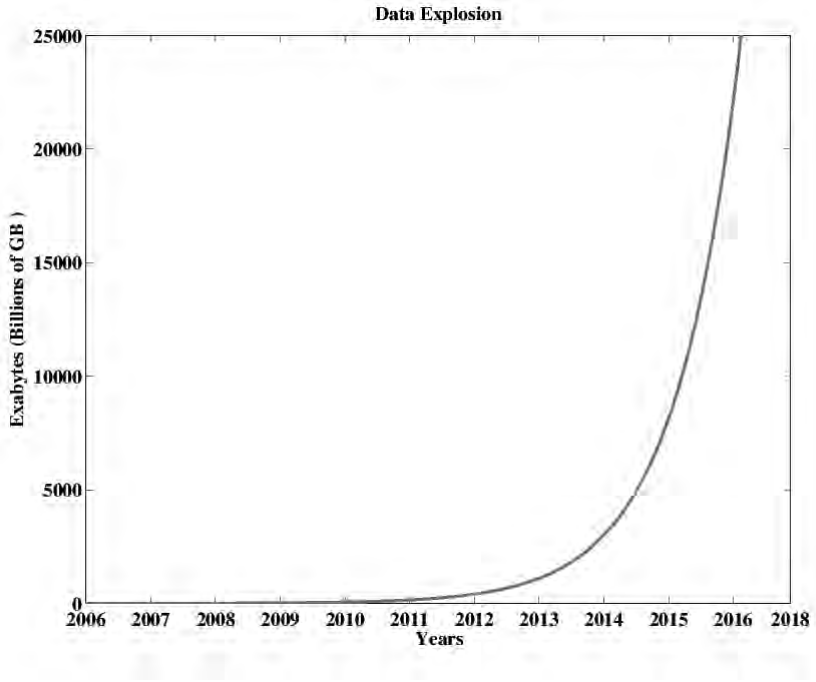


FIGURE 3.1: An exponential growth of data [2].

With this data at our disposal, we continue to ask similar questions and seek similar answers. But no longer can we find the patterns with our unaided eyes. Now we call upon sophisticated computer algorithms that learn from the data we provide in order to assist us in conducting our data-driven science.

In 1950 Alan Turing stimulated a new field of computing with his classic paper "Computing Machinery and Intelligence" [24]. In 1955 the term *Artificial Intelligence* (AI) was coined by John McCarthy as "the science and engineering of making intelligent machines" [25].

While the long-term goals of AI include granting capacity for self-awareness, human thought, and emotions to machines, the contemporary function of AI often takes form in the processing of data with accuracy that matches or surpasses that of a human, at far greater speeds.

A subset of Artificial Intelligence and cornerstone to AI since its inception, *machine learning* is the design, development, and study of computer algorithms that learn and then automatically improve through experience [26]. Or, as Arthur Samuel defined machine learning in 1959, a "Field of study that gives computers the ability to learn without being explicitly programmed" [27].

NASA's Solar Dynamics Observatory captured its 100 millionth photo of the sun in January 2015. The Large Hadron Collider captures 600 million collisions per second. We are now able to observe the interaction of 50,000 atoms in a fraction of a millisecond

and Facebook adds 300 million new photos every day. If built to its full design, the Square Kilometre Array will generate nearly 3 GB of astronomical data per second [28].

As challenging as is the quantity of data, the dimensionality of the data is equally complex. If the millions of detections per second generated by any instrument were described by simple x, y, and z coordinates, then the task of understanding the data would be one of processing speed alone. But each of the given examples is not so simple, for data-driven science often works with a quantity of data parameters far greater than that which can be viewed on a graph.

For example, in the "Machine Learning" lecture series by Andrew Ng [29], Ng refers to what appears to be a relatively simple effort to determine the value of a house in a given market. At first glance, it seems the number of bedrooms and bathrooms, total square-meters, and a few other parameters would be ample to properly estimate the selling price.

However, one quickly comes to understand there are far too many parameters to consider in order to accurately project the correct selling price through historical averaging, including but not limited to: number of bedrooms and size of each bedroom, number of baths (both full and half), size of garage; carpet, tile, or wood floors and type of kitchen counters, height of ceilings in each room; single, split, or multi-level arrangement; size of front and back gardens, recent selling prices of nearby houses and some quantifiable valuation of the neighbourhood.

The total number of parameters used to describe any given house can easily rise to over 100. To plot these on a single Cartesian coordinate system is impossible. To assess them in pairs would be far too resource intensive and ultimately unreliable, as the relationship between the parameters, when addressed just two or three at a time, may be non-obvious.

Therefore, we need a unique tool to assist us in this data discovery process. Machine learning enables the capacity for analysing both large volume and complex data, where the dimensionality p of the parameter space and the sample size n are used to characterise the dataset. The exact machine learning technique applied to a particular dataset depends upon the ascribed dimensionality [30], meaning one machine learning technique might work with one category of data, but fail on another. Many machine learning algorithms require that *features* be generated from the raw data before application and processing.

3.2 Feature Engineering

If a dataset is a collection of raw data points (one or more measurements of the real world) then features are transformed data, values extracted from the observational data. According to Dr. Arun Aniyan [2], post-doctorate fellow at the Square Kilometer Array, South Africa, features:

- Are informative and non-redundant
- Support subsequent learning and generalisation
- Reduce dimensionality (in most cases)
- Lead to better human interpretation of the overall data

It is often difficult to understand the difference between a datum and a feature, for in each field of study these definitions might vary, while in the field of machine learning the terms *dataset* may refer to raw data or a collection of features. Throughout this dissertation, I will describe features as belonging to a *featureset*.

In general, data are the raw values recorded by a human or machine, such as voltage, amperage, or frequency; humidity, temperature, or time of day; quantity of purchases, type of product, or number of products returned. Features express the relationships between these data, such as the rise or fall of temperature over a given period in time, the relationship between the quantity of products sold and the quantity returned in a given category; the Fourier transform of a signal; the average, kurtosis, and skew of continuous data [31].

Feature engineering is essential to machine learning, and is often the most arduous and challenging aspect in which one becomes intimately familiar with the data in order give the machine learning algorithms the greatest chance of success.

Feature engineering may be described by three successive processes [31]:

1. Feature extraction
2. Feature selection
3. Feature construction

3.2.1 Feature Extraction

Feature extraction is an automated method of working with observational data to build new, derived, informative and non-redundant features which lead to better interpretation of the overall data. Because many observations produce far too much data to be used directly in machine learning, feature extraction reduces dimensionality of the data while at the same time improving the success of the predictive models [2].

For audio data, waveform analysis can be employed. For tabular data, principal component analysis (PCA) and unsupervised clustering methods can be employed. For image data, filters familiar to graphic designers such as edge detection and quantitative analysis of a particular colour or bit-mapped image might be employed.

Key to feature extraction is that the methods are automatic (even if designed and constructed from simpler methods) and work to reduce high-dimensional data.

3.2.2 Feature Selection

Feature selection is the process of isolating the subset of relevant features which best describe the dataset as a whole, as some features may not be rich in discriminatory power. Feature selection works to reduce the dimensionality of the data, which allows the machine learning algorithm is more readily discern the patterns [2].

In a world in which data is generated often faster than it can be processed, and in far more dimensions than it can be visualised, machine learning algorithms may suffer from the *curse of dimensionality*, a condition in which too many parameters work to reduce, not improve the overall success of a given algorithm [23].

Feature selection can be a manual process, calling upon an expert's knowledge of the data. For example, a time-based feature should be removed if the event we desire to better understand is stochastic in nature and has no correlation to time. Or, if a single value is constant across *all* classes, it offers no benefit to distinguishing one class from another and should be removed.

For those features which do not immediately lend themselves to an obvious use or discard selection criteria, density estimation by means of histograms, boosting, and Principal Component Analysis (PCA) are frequently used feature selection tools, and are briefly introduced in the following.

A *histogram* is a graphical representation of the probability distribution of a continuous variable. A histogram is constructed by dividing a full range of data values into equally

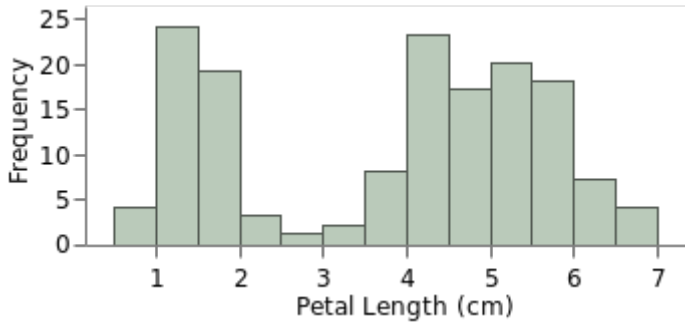


FIGURE 3.2: A histogram of the Iris data set [Daggerbox, Creative Commons].

spaced, consecutive, non-overlapping bins, then tabulate how many data points fall into each bin [32]. An example is given in Figure 3.2, the distribution of petal lengths from the Iris dataset (Section 5.4.3). While relatively simple in its function, a histogram provides a quick, visual means to conduct initial feature selection.

Boosting seeks relatively weak correlations between features, correlations just slightly better than random guessing, then employs them to improve the overall performance of a learning algorithm [33].

Principal Component Analysis (PCA) is the leading tool for modern data analysis, and a powerful means to conduct feature selection. PCA does not modify data, rather, it transforms its representation, as follows:

1. Select linear sub-spaces that are orthogonal to each other (eigenvectors).
2. Project the original data onto these sub-spaces.
3. Arrange the projected vectors in decreasing variance.

By engaging only those features which are strongly correlated, PCA retains the strength of your featureset while removing the weakly correlated or noisy features [34] [35].

Feature selection can also be conducted with Genetic Programming (Section 3.4.4.2). As discussed in "A Survey on evolutionary computation Approaches to Feature Selection" by Bing Xue, et al [36], Genetic Programming evolves functions in which features are tested for their potential contribution toward a given solution. Those which exhibit a beneficial contribution will be retained, while the others will be dropped from successive evolved functions and fall out of the solution space. As with the application of histograms, boosting, or PCA, the correlated features are more likely to contribute to the success of a machine learning algorithm. In Chapter 6 we will discover how multiple, successive generations of evolution retain some features, and drop others.

In addition, the arithmetic, trigonometric, or boolean function which defines the relationship between those retained features may itself be employed as a higher-order feature, as described in the following Section "Feature Construction".

3.2.3 Feature Construction

Feature construction is similar to feature extraction in that one builds new, derived, informative and non-redundant features. However, in place of an automated process it is conducted manually. There are no standard methods for feature construction, for it may differ with each dataset and call upon domain-specific expertise.

One fairly automated method of feature construction employs Genetic Programming to build multivariate expressions which are in and of themselves demonstrations of relationships between two or more data parameters. The following examples from Chapter 6 demonstrate how multivariate expressions evolved by Genetic Programming exhibit repeating structure.

Example, evolved expressions from an original pool of nine, arbitrary features:

$$az^2 * ea/ch - az * ea/t + az * t - ch * ea/vh + ch * t^3 * vh + 2 * ch + ch * vh/t - ea + 3 * t$$
$$az^2 * ea/ch - az * ea/t + az * t - ch * ea/yx + ch * t^3 * yx + 2 * ch + ch * yx/t + 2 * t$$
$$az^2 * ea/ch + az + ch * t^3 * ea - ch * ea/vh + 2 * ch + ch * vh/t - ea^2/t - ea + t + vh$$

Within these expressions we see find simple sub-functions, correlations which may serve as valuable, higher-order features:

$$az^2 * ea/ch$$
$$ch * t^3$$
$$ch * ea$$
$$2 * ch$$

This process is further described by Soha Ahmed, et al, in "Multiple feature construction for effective biomarker identification and classification using genetic programming" [37].

3.3 Approaches to Machine Learning

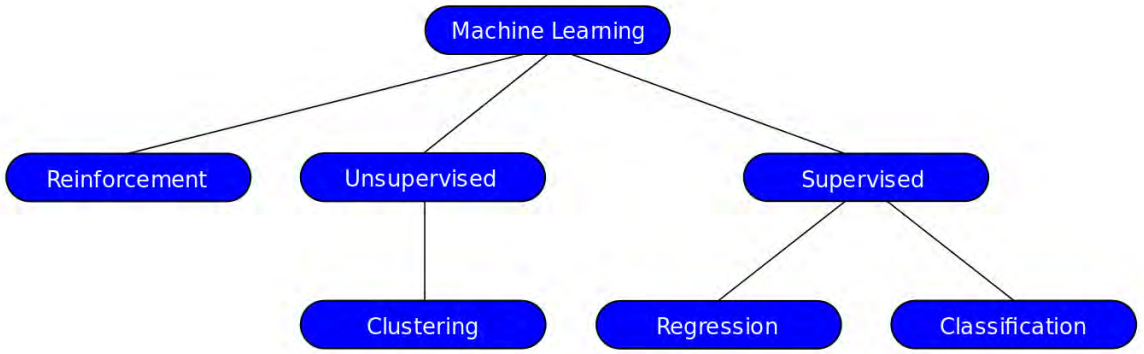


FIGURE 3.3: Primary approaches to machine learning.

As shown in Figure 3.3, there are three primary approaches to machine learning: reinforcement, unsupervised, and supervised. The decision as to which to engage depends upon what is known about the data, and how the algorithm is to be trained. One might engage reinforcement learning in an environment in which the data is gathered in real-time, while unsupervised and supervised data is typically presented to the machine learning algorithm as an established dataset, unlabelled or labelled accordingly.

While there are many applications of machine learning, in the body of this thesis *symbolic regression* and *classification* are discussed. Regression seeks relationships between features, such as solving for d, given values a, b, and c, while classification assigns one of n labels to a given data point which infers association to one of n classes [30].

3.3.1 Reinforcement Learning

Inspired by behavioural psychology, *reinforcement learning* is concerned with how computer programs operate as independent agents, making decisions in a given environment so as to maximise a reward. In this environment, the training occurs in real-time, in the real world [30].

For example, a machine learning algorithm which is learning to drive a vehicle will learn from the actions of a human driver (accelerator, brake, steering wheel) and from the input of sensors mounted on the vehicle while moving through an unknown, dynamic environment. Once the algorithm is given control of the vehicle, the human driver will correct the decisions of the trained system, providing real-time feedback for its actions. There is no stated goal, rather a continuous feedback process.

While reinforcement learning is not further addressed in the body of this research, it is important to understand its function as a principal approach to machine learning.

3.3.2 Unsupervised Learning

In highly complex, real-world systems, such as the transmission of pathogens across continents, fluctuations in the stock market; pattern recognition in social networking, airport security, and astronomy, unsupervised machine learning can yield rapid, highly accurate data mining results.

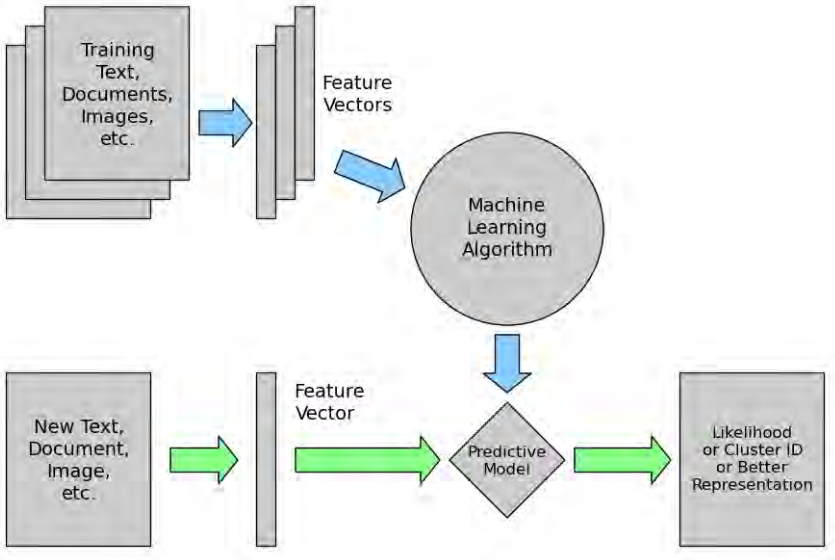


FIGURE 3.4: Unsupervised learning [SciKit Learn] during which training data is used to develop a number of hypotheses initiated by random seeds (blue arrows), one of which is then selected to be deployed in a real-world pipeline (green arrows).

Unsupervised learning seeks to find hidden (underlying) structure in data without the application of training labels[30]. Unsupervised learning does not provide *reward* or error feedback, rather, the resulting output, typically a measure in n dimensional space from each data point to the classification boundary or centroid, is compared to a prior analysis by the same algorithm to determine if the classification of the data has improved (Figure 3.4).

With each iterative pass, the algorithms employed are given *weighted* feedback to bias the boundary or centroid in order to improve performance in the subsequent run. One approach to unsupervised learning is clustering. Clustering works to assign class labels to data points so as to group them with other data points, a cluster more similar to each other than to those of one or more other clusters [38].

As with reinforcement learning, unsupervised learning will not be further addressed in this body of research, but it helps to understand this method in contrast to supervised learning, which is the method to be employed by all data analyses.

3.3.3 Supervised Learning

Supervised learning uses data labelled by a human prior to the training process[30]. This labelled data is either raw data, meaning the direct output of a measurement system, or features built from the raw data. Typically, supervised learning is used to conduct regression or classification.

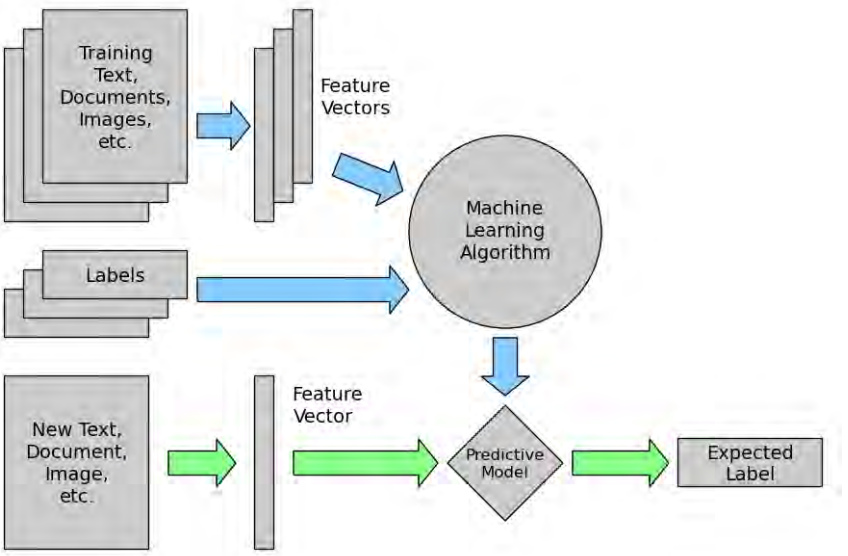


FIGURE 3.5: Supervised learning [SciKit Learn] during which training data associated with labels (solutions) is used to develop a number of hypotheses (blue arrows), one of which is then selected to be deployed in a real-world pipeline (green arrows).

In both cases, a supervised machine learning algorithm is trained upon labelled data (Figure 3.5). The derived, tested model is then employed against unlabelled, prior to unseen data in order to generate a prediction, a single value in the case of regression or a class label as in the case of classification [29].

Examples of types of supervised learning include decision trees, ensemble Bagging, Boosting, and Random forest; Naive Bayes, k-Nearest Neighbours (kNN), Support Vector Machine (SVM), evolutionary computation (e.g. Genetic Algorithms, Genetic Programming), and Neural Networks [39].

3.4 Machine Learning Algorithms

There are a number of machine learning algorithms which differ in their approach, ability to process particular kinds of data, associated computational overhead, and outcome. There is no one algorithm which works well with all kinds of data, nor one algorithm which satisfies all user needs. It is common for more than one algorithm to be used

together, sometimes one acting as a pre-processor to the other, or as a feedback mechanism, helping to improve accuracy of the total system.

While not core to this body of research, I will first describe Artificial Neural Networks in order to provide an understanding of one of the most popular algorithms employed in data mining today. I will then describe k-Nearest Neighbour (kNN) and Support Vector Machines (SVMs) in order to establish a recognised baseline against which I will compare the performance of Genetic Programming, which is core to this body of research.

3.4.1 Artificial Neural Network

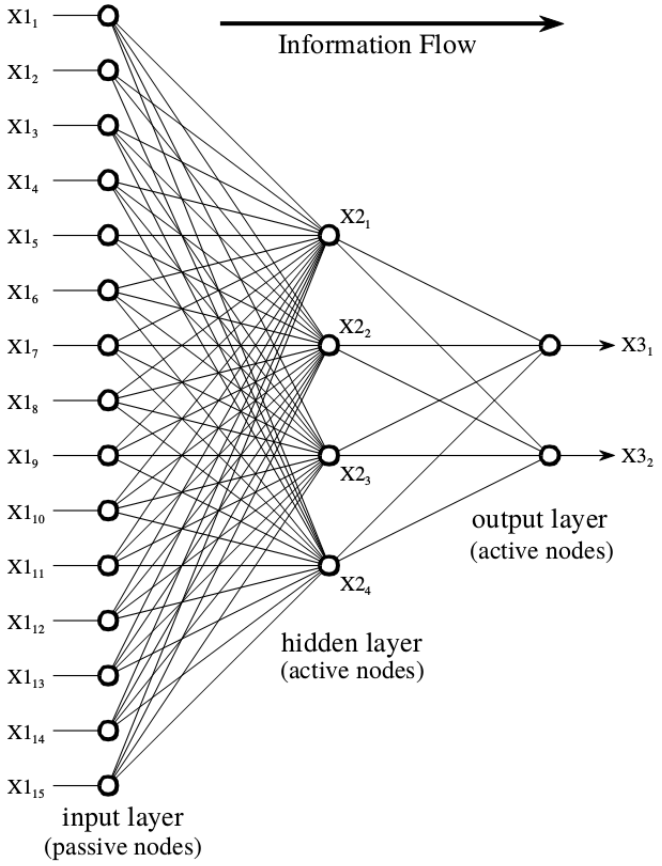


FIGURE 3.6: The most common Artificial Neural Network architecture is composed of three fully interconnected layers. The nodes of the input layer pass the data to the nodes of the hidden layer unaltered. The hidden and output layers modify the data with a weight applied at each node, terminating in two outputs [3].

An Artificial Neural Network (ANN or Neural Net) is an approach to machine learning inspired by biological neural networks, the organic equivalents of digital processors. In machine learning, Neural Nets are often used to approximate functions that encompass a large number of input parameters [3].

As shown in Figure 3.6, the most common Artificial Neural Network architecture is composed of three layers: *input*, *hidden*, and *output*. Each node is represented by circles, while the connecting pathways, analogous to neurons, are represented by lines. Note that each node of each layer is connected to all nodes of the next layer, in the direction of information flow. This Neural Network is therefore *fully interconnected*.

The nodes of the input layer $X1_1, X1_2 ... X1_{15}$ receive raw data or features from a prepared training dataset. The input layer does not modify the data it receives, rather it passes the data to each node of the middle, hidden layer. The input layer then is *passive*. The hidden and output layers are *active* and modify the data. As the data arrives at each node, the value of the data is multiplied by a *weight*, a predetermined value called from a set stored by the program.

As each node will receive input data from more than one node in the prior layer, the sum of all the weight adjusted data are added to produce a single number. Before being passed to the next layer, this single, summed value is further modified by a *sigmoid function*, a means of adjusting any value from -∞ to +∞ to lie between 0 and 1.

As shown, there are two output nodes in the final, active layer for this particular Neural Net [3].

With this example, the data flows only in one direction, from input to hidden to output layers. Other types of Neural Nets involve feedback paths which work to modify the weighted adjustments to guide the overall function of the Neural Net closer to the desired solution.

The kind of problems for which a Neural Net may be applied are as varied as the many applications of machine learning. A few examples include stock market predictions, voice recognition and image identification [3]. The data could be the valuations of stock for particular companies or exchange markets, waveform data and RGB pixel values.

Neural Nets are currently the most widely used machine learning algorithms in both research and commercially deployed systems for image processing, language comprehension, and control systems [30]. Neural Nets employ hundreds of internal connections, supporting complex classification routines. However, Neural Nets do not produce human readable functions. The relationships between the features, provided or internally developed, remain obscure.

In some environments it is beneficial for the human observer to understand the relationship between two or more features in a given dataset and the associated, real-world system. Genetic Programming generates human readable, multivariate expressions which are readily isolated and made deployable outside the system in which they evolved.

3.4.2 k-Nearest Neighbour

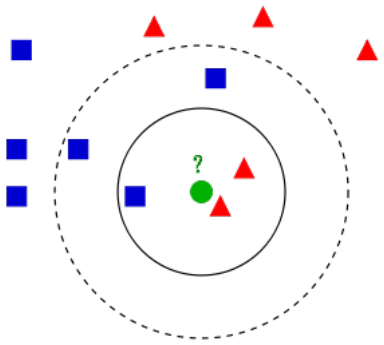


FIGURE 3.7: Classification by k-Nearest Neighbour [A. Anaj, Creative Commons] in which the green circle is a test sample that needs to be classified. Depending upon the numeric value associated with the green circle, its nearest neighbours will be either the two red triangles or three blue squares, as denoted by the solid and dashed circles, respectively.

k-Nearest Neighbour (kNN) is a supervised machine learning method through which data points in two or more dimensions are labelled according to closest proximity neighbours [30]. kNN works on the principal that data points of the same class will be described by features which are relatively close to each other in the n dimensional space, where n is the number of features.

The k-nearest neighbour algorithm does not impose assumptions about the distribution from which the sample is drawn, nor does it build an internal mathematical model. Rather, it engages a training set that contains both positive and negative correlations and conducts classification by means of a simple, data point-by-point vote. The distance metric works to minimise the distance between same-class data points and maximise the distance between dissimilar classes.

A prior to unseen, unlabelled data point is classified by calculating the distance to its nearest neighbouring training case. The class label of that nearest data point will determine its own classification (Figure 3.7).

The performance of the kNN algorithm is based upon three principal factors [39]:

- Distance measure (metric) used to locate the nearest neighbours
- Decision rule (kernel) used to derive a classification from kNN
- Number of neighbours k used to classify the new data point (training set)

The strength of kNN is its simplicity in training, and application to a broad range of classification and regression problems. The weakness of kNN is the computational overhead for the classification of new data points.

3.4.3 Support Vector Machines

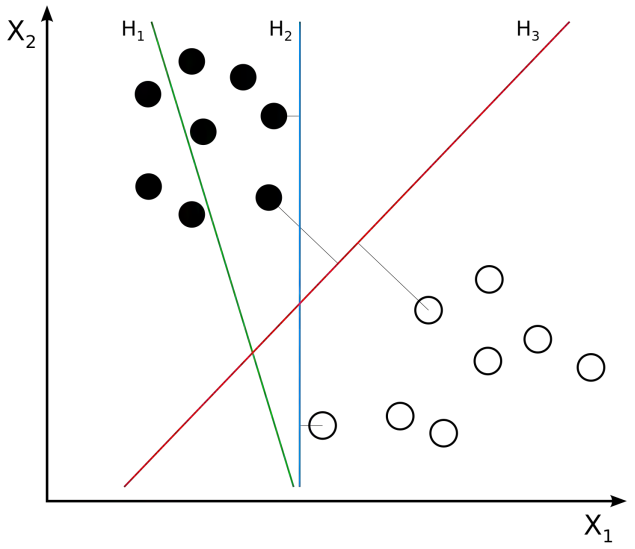


FIGURE 3.8: Classification by Support Vector Machine [(author unknown), Creative Commons] where vectors H1, H2, and H3 each attempt to separate the two classes of data. H1 fails to provide the desired separation. H2 succeeds, but with a narrow margin. H3 cleanly separates each class with the widest possible margin, which is the intended function.

Support Vector Machines (SVMs) are supervised machine learning methods which cluster, classify, and rank data, making it a non-probabilistic classifier [30]. SVMs were originally used to perform binary classifications and regression estimations. However, modern implementations work with multiclass (more than two) classification datasets.

There are a number of types of SVM kernels, including linear, polynomial, sigmoid, and radial basis function (RBF). Each invokes a unique algorithm employed to iteratively compare all data points in the given set. It is often the case that more than one kernel and its subsequent parameters will be used on a single dataset. The outcome of each run is compared to determine which is more likely to produce the highest quality output [40].

SVMs represent data as points in parameter space, mapped to fall into two or more unique categories divided by a gap that is as wide as possible (Figure 3.8). Once established by the training set, new test data is mapped into that same space and predicted to belong to one of the categories, according to the geometric mapping.

A Support Vector Machine constructs a hyperplane or set of hyperplanes in a high-dimensional *feature space*. The best separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class. This is the functional margin, for in general, the greater the margin the lower the error of the classifier [39].

While the original data features may be correlated in a finite-dimensional space so as to correctly apply the classification labels, it often happens that the data points are not linearly separable in that space. Therefore, the original finite-dimensional space is mapped into a much higher-dimensional feature space in which the separation is simplified.

As with ANNs, SVMs work well with a large numbers of features. However SVMs are are inherently binary classifiers.

3.4.4 Evolutionary Computation

In nature, genetic chromosomes give rise to variation in the individuals of a given population. Variation in the chromosomes is expressed as variation in the structure and behaviour of the individuals, for their given environment. Variation in that structure and behaviour is reflected by individual survival and reproduction. Individuals better able to perform a given task in their environment tend to survive and reproduce at a higher rate. Fewer, less fit individuals will survive and reproduce in the same environment. This is the foundation of survival of the fittest and natural selection, as described by Charles Darwin in *On the Origin of Species by Means of Natural Selection* [41].

Through the production of many subsequent generations, the population as a whole evolves to contain more individuals whose chromosomes are expressed as structures and behaviours that enable those individuals to survive and reproduce. This is key to biological evolution, and through this process of *natural selection*, structural change arises in response to physical fitness [42].

Evolutionary computation simulates the biological processes described by Darwin, working to improve computer programs or mathematical functions that describe real-world measurements. Evolutionary Algorithms (EA) are a type of evolutionary computation in which individual programs are randomly selected and tested for their fitness within the given environment. Inspired by biological evolution, a selection process then determines which individuals will be engaged in reproduction, mutation, and recombination to populate the subsequent generation of programs [42].

Each generation has the opportunity to improve upon the prior such that given enough generations, the EA might, but is not guaranteed to solve the given problem.

Two of the most popular applications of evolutionary computation are Genetic Algorithms and Genetic Programming.

3.4.4.1 Genetic Algorithms

In the 1975 publication *Adaptation in Natural and Artificial Systems*, John Holland provides a general framework for representing natural and artificial adaptive systems. He demonstrates the application of evolutionary processes to artificial systems, concluding that any problem which can be formulated in genetic terms can often be solved by what we now refer to as a "genetic algorithm" [43].

A Genetic Algorithm (GA) is evolutionary computation that simulates Darwinian evolutionary processes and naturally occurring genetic operations on chromosomes in order to discover previously unknown relationships between features in the data.

In the natural world, chromosomes consist of character strings built from four nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). A sequence of nucleotide bases constitutes a *chromosome string*. All the chromosomes which define an organism are the genome. For example, the human genome contains 2,870,000,000 nucleotide bases.

Genetic Algorithms are built from mathematical objects which are similar in structure to naturally occuring chromosomes. In classic GA, these chromosomes are binary (as compared to base-4 in DNA), each bit of a fixed-length string representing a feature of the real world. A simple problem can be described by a single character string, or chromosome. More complex, multifaceted problems can be described by a combination of two or more chromosomes, each of which describes a select feature. Each chromosome evolves and is then tested against a fitness function (Section 4.6). Through successive generations of evaluated chromosomes, the best, overall solution is found and the genome is defined.

Successive populations (generations) evolve using genetic operations patterned after the Darwinian principle of reproduction and survival of the fittest. Those which perform better are more likely to pass their code to the next generation, and therefore more accurately solve the given problem [42].

3.4.4.2 Genetic Programming

Genetic Programming (GP) is evolutionary computation that employs naturally occurring genetic operations, a fitness function (Section 4.6), and multiple generations of Darwinian evolution to resolve a user-defined task [44]. However, GP differs from GA in that it evolves a population of programs whereas GA evolves a population of chromosomes. GP directly traverses a solutions space (Figure 3.9) while GA traverses a search

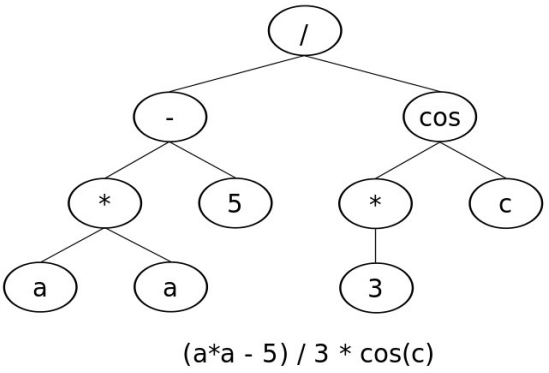


FIGURE 3.9: Tree-based Genetic Programming syntax tree.

space, its hypotheses tested outside of the algorithm. GP evolves the program itself in an effort to arrive to the desired solution.

In general, GP can be used to discover a mathematical relationship between features in data (regression) or to group data into categories (classification). Both of these employ a training (learning) and testing (validation) phase, after which the resultant, mathematical function may be applied to prior to unseen, online data [42].

GP will be the principal focus of this thesis and the associated body of research. As such, an in-depth study is provided in Chapter 4.

3.4.5 Evaluating a Hypothesis

All hypotheses developed by machine learning algorithms must be evaluated in order to determine how well they fit the given dataset, that is, how well they represent the real-world environment from which the data was acquired. With a 2- or 3-dimensional dataset it is relatively simple to plot the hypothesis $h(x)$ against the data in order to visualise how well the function separates the *classes* (or categories) to which data belongs [29]. However, for problems with more than three dimensions, it becomes difficult, even impossible, to visualise the hypothesised function.

The most popular means of evaluating a machine learning hypothesis is through training and subsequent testing. The implementation of a *data split*, the separation of data into two categories, *training* and *testing*, provides the machine learning algorithm with a unique dataset on which to train, and another on which to test. Two fairly standard ratios by which to split the dataset are 70%/30% and 80%/20%.

In general, and especially when working with ordered (time series) data, the entire dataset should be randomly re-sorted in order to reduce the potential of overfitting

(Section 3.4.6). When the time variable itself contains useful information, for example a sequential increase or decrease (cycle) of a particular variable, then the application of the machine learning algorithm to any given sequential subset will train the algorithm only on a narrow representation of the data, increasing the potential for global failure.

For example, within a times series dataset of 100 points, there may be particular behaviour in the first 50 points that differs from the final 50. A hypothesis derived from the first 70% of the data would likely do poorly when applied to the remaining 30% as it was not given opportunity to train on the potentially unique values found only in the final 30%. Therefore, the data in this time series should be randomly sorted prior to the data split, allowing for representatives from each region of the time series to be present in the training set [29].

In order to evaluate the effectiveness of an hypothesis, in the case of GP, a multivariate expression, we compute the training error as:

$$J_{train}(\theta) = \frac{1}{2m_{train}} \sum_{i=1}^{m} \left(h_\theta(x_{train}^{(i)}) - y_{train}^{(i)} \right)^2 \tag{3.1}$$

where parameters θ from the training set work to minimise the training error objective $J_{train}(\theta)$ which is 70% of the data and m is the total number of training samples. To avoid overfitting, the hypothesis with training parameters $\theta_0, \theta_1, \theta_2, ...\theta_n$ must be evaluated against the m_{test} set that contains the remaining, randomly sorted 30% of the total dataset.

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m} \left(h_\theta(x_{test}^{(i)}) - y_{test}^{(i)} \right)^2 \tag{3.2}$$

where x_{test}, y_{test} is the total dataset with x_{test}^1 and y_{test}^1 denoting samples of the test set. This is the average squared error as measured on the test set.

Application of an offline training/testing data split greatly improves development of a hypothesis which fits the total dataset and that of new data in an online, real-world environment.

3.4.6 Underfitting and Overfitting

Underfitting occurs when a machine learning algorithm has developed a function which is simpler than that required to describe the training set. For example, a linear function (degree-1 polynomial) would be underfit to a dataset which can only be described by an exponential function.

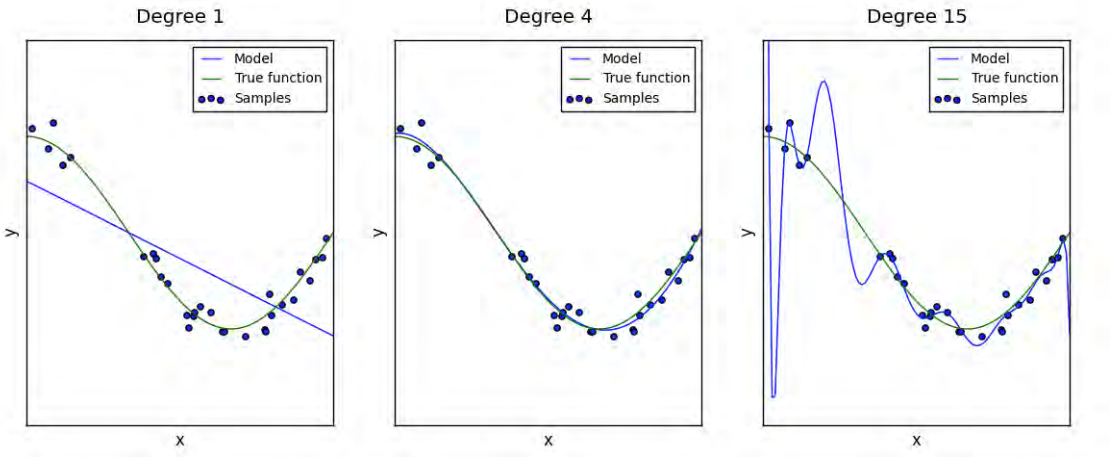


FIGURE 3.10: Underfitting, a well fit model, and overfitting a function to a dataset [SciKit Learn].

Overfitting occurs when a machine learning algorithm has developed a more complex function, such as a higher-order polynomial which matches the given training set *too* closely, resulting in the inability to generalise to a larger, real-world data pool.

In Figure 3.10, three polynomial functions have degrees 1, 4, and 15 respectively. At left, the linear function (polynomial with degree 1) is not adequate to fit the training data. At degree 15 the model *overfits* the training data, meaning it also includes the noise of the training data in its approximation. The polynomial of degree 4 fits the training data nearly perfectly.

The mean squared error (MSE) is used to evaluate the hypothesis for underfitting and overfitting [45] on the validation dataset. The higher the outcome, the less likely the model generalises to test and ultimately, online data.

3.4.7 Accuracy, Precision and Recall

Accuracy, Precision and Recall are key measures by which the performance of machine learning algorithms are qualified. Is is possible to have high accuracy and low precision or high precision and low accuracy (Figure 3.11). In a best case scenario, an algorithm classifies the data with both high accuracy and high precision.

Accuracy is a measure of the distance between the desired quantity and the generated result. The error then is the difference between the desired quantity and the result. Accuracy is measured as a percent, as [46]:

$$Accuracy = \left(\frac{\#\ of\ correctly\ classified\ examples}{\#\ of\ examples} \right) * 100 \tag{3.3}$$

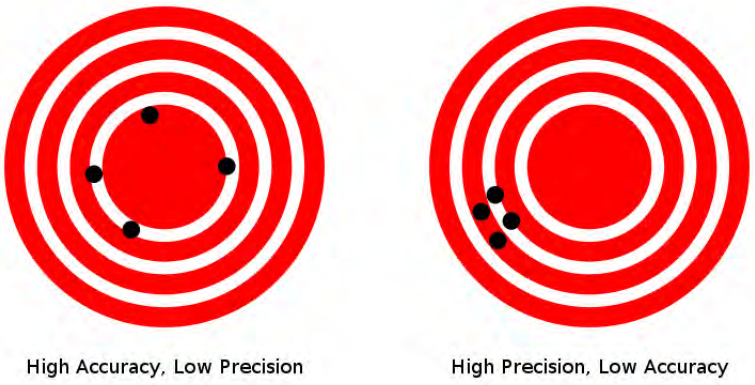


FIGURE 3.11: A visual demonstration of accuracy and precision [Zetawoof, Creative Commons], where accuracy measures distance from the desired goal and precision measures distance between multiple results.

Precision and Recall work with *true positives*, *false positives*, *true negatives*, and *false negatives*, and the relationship between them.

Precision (P) is the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p), and is measured as:

$$P = \frac{T_p}{T_p + F_p} \tag{3.4}$$

Recall (R) is the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n), and is measured as:

$$R = \frac{T_p}{T_p + F_n} \tag{3.5}$$

These quantities are related to the F1 score, which is the harmonic mean (weighted average) of Precision and Recall[47]:

$$F1 = 2\left(\frac{P \times R}{P + R}\right) \tag{3.6}$$

A high precision relates to a low rate of false positives while a high recall relates to a low rate of false negatives. High scores for both precision and recall demonstrates the classifier is running well. At its best, the F1 score reaches 1; at its worst 0. Both Precision and Recall contribute to the F1 score equally.

The application of Accuracy or Precision and Recall depends upon the type of data analysed and the desired understanding of the results. When used alone, accuracy is a simple metric which presents a one-dimensional understanding of the data analysis:

How close to the known value did the generated result land? For regression analysis, this may be adequate.

For classification however, Precision and Recall offer an improved understanding of how well the hypothesis fits the data in the context of the given problem. Not only is it made clear what percentage of the results match the known outcome, but for those that were incorrect, how they missed. *Were they incorrectly labelled as positive, or labelled as negative when they should have been positive? And what is the ratio of types of missed fits?*

No matter the machine learning algorithm employed, evaluation of the hypothesis is crucial to validation, for it gives the researcher a means to learn from current effort, reconsider the approach, and improve. Whether we are working with an Artificial Neural Network, Nearest Neighbour, or an evolutionary algorithm, the output must be human readable in some form, as the hypothesis itself, or as the classified data. kNN does not build an internal mathematical model. While Neural Nets form highly complex, powerful models, they remain black box solutions. The output of both is the labelling of the data. Genetic Programming is unique in its provision of a human readable, multivariate expression.

Chapter 4

Genetic Programming

4.1 Survival of the Fittest

According to the 18th century naturalist Charles Darwin, evolution by natural selection is a process inferred from three facts about populations: "1) more offspring are produced than can possibly survive, 2) traits vary among individuals, leading to different rates of survival and reproduction, and 3) trait differences are heritable." [48]

As one of the many varied approaches to machine learning, GP is an evolutionary computation algorithm that has the capacity to solve problems without being given the solutions in advance. As its name implies, Genetic Programming is an analogue to biological evolution in that it evolves a population of individual programs over time. Each generation contains individual programs which differ from those of the previous generation, transformed through the application of genetic operators mutation and sexual reproduction. GP does not require continuous human supervision or intervention, rather, the random alterations of individuals in the population evolve, with the best suited program or programs for the given environment retained [42].

GP searches a *program space*, meaning the alteration of the program itself in order to evolve a hypothesis which takes the form of a mathematical, multivariate expression. GP employs an offline training (learning) and test (validation) phase to produce the hypotheses. Once evaluated, the hypothesis can be employed as a portable model applied to online data.

In biological natural selection, testing for survivability preserves traits best suited for the functional role each trait performs [41]. In a similar respect, GP learns how well a program functions by comparing the output of its multivariate expression to a specified, qualified outcome. This comparison is quantified to generate a numeric value a *fitness*

score (Section 4.6). Those programs that demonstrate a high fitness score are more likely, but not guaranteed, to be selected for the next generation. Thus, the subsequent generation of individual programs are more likely, but not guaranteed to solve the given problem. As with natural selection, GP is a stochastic process that never guarantees the desired results [44].

4.2 Brief History

In the 1950s Alan Turing was likely the first to propose programs that evolve [49]. It was more than thirty years before Richard Forsyth successfully demonstrated evolutionary computation as a means to classify crime scene data in a United Kingdom investigation [50].

John Holland, professor to John Koza, was instrumental in carrying forward evolutionary computation in the 1980s, including orchestration of the first conference on genetic algorithms.

John Koza authored more than 200 publications on "Genetic Programming", a name coined by David Goldberg, a student of John Holland in 1983 [51]. GP was established in the early 1990s with the publication of four books by Koza [42], and the subsequent, rapid proliferation of use of GP in academic research. There are now more than 11,000 entries contained in the Genetic Programming Bibliography [52] with 77 uses of GP which were human competitive noted by John Koza as of 2010 [53].

GP has been applied to a diversity of problems, including software synthesis and repair, predictive modelling, data mining, financial modelling, product design, image processing, and more recently, astro-particle physics. More examples of real-world applications and success with GP are shared in "A Field Guide to Genetic Programming" [44] and in "Genetic Programming, An Introduction—On the Automatic Evolution of Computer Programs and its Applications" [54].

Active international conferences which feature GP include the Genetic and evolutionary computation Conference (GECCO), EuroGP, the IEEE World Conference on Computational Intelligence, the IEEE Congress on evolutionary computation, and the International Conference on Parallel Problem Solving from Nature.

4.3 The Generational GP Algorithm

According to "A Field Guide to Genetic Programming" [44], there are three basic steps to generational GP:

1. Generate an initial, stochastic population.
2. Iteratively perform selection, genetic operation, and evaluation:
   (a) Evaluate each program (hypothesis) in the current population against the given dataset and determine how well it performed, the value recorded as a fitness score.
   (b) Randomly select programs and compare their fitness scores.
   (c) Apply one of three genetic operators to a copy of the leading program:
      i. reproduction
      ii. mutation
      iii. crossover

      then move the program into the subsequent generation.
   (d) Evaluate all programs (hypotheses) in each new generation.
3. Repeat until the user-defined termination criteria are met.

In GP, each generation is composed of a population of individual programs. Each program is a mathematical function that when executed against the given data, produces a value.

The initial population (generation 0) is composed of randomly constructed programs built upon user-defined arithmetic, trigonometric, or boolean operators, and user-defined operands, variables which represent specific data parameters. The quantity of programs in the initial and all subsequent populations is defined by the user, and remains constant throughout the entire run.

The entire initial population is evaluated, meaning each individual program is executed against the given dataset, producing a single outcome per program. This outcome is measured against the known solution, and recorded as the fitness score.

From this initial population, individuals programs are randomly selected for a tournament in which the program with the best fitness score is copied into the subsequent generation, following application of a *genetic operator*:

1. With *reproduction*, the program is copied without modification.
2. With *mutation*, a portion of the program is randomly modified.
3. With *crossover*, two programs are selected to contribute a portion of their own code to a new program.

Each mutation and crossover might result in a reduction or improvement in the fitness score for each individual program. As with the initial population, all programs in each new generation are evaluated, randomly selected for a tournament, and then evolved for the subsequent generation. Because those programs selected exhibit the highest fitness score, their contribution to the next generation enables an overall improvement.

This process continues until the user-defined termination criteria are met.

Each of these noted stages are described in detail in the following section.

4.4 Tree-based Genetic Programming

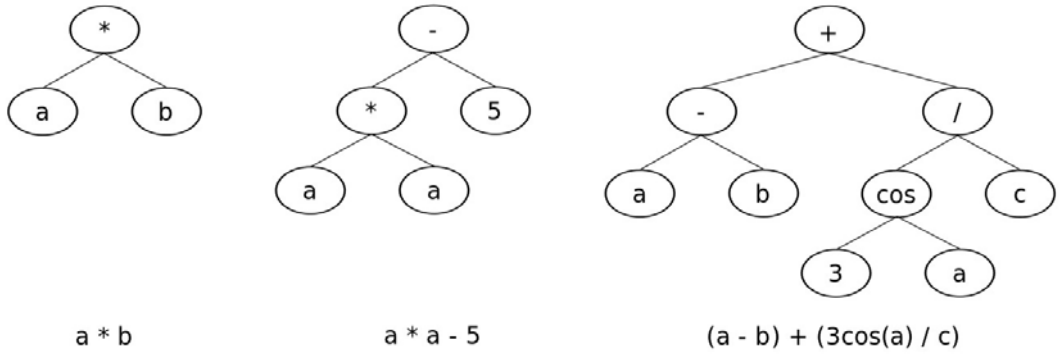


FIGURE 4.1: Three examples of GP syntax trees, each of which produces a unique expression when evaluated.

The concept of GP, as developed by John Koza in the mid '90s, has been modified and extended to include a number of variations, including but not limited to Strongly Typed GP, Cartesian GP, Semantic Programming, GP based Program Synthesis; Linear and Graph GP, Probabilistic GP, Multi-objective GP, and Fast and Distributed GP [44].

This present body of research employs tree-based GP wherein the program is evolved and expressed as a *syntax tree* (Figure 4.1). The components of a GP syntax tree are traditionally defined as *nodes* and *leaves*, or *Functions* and *Terminals* respectively. Combined, the nodes and leaves form a *primitive set*. Nodes are any segment of a syntax tree branch that is not the bottom, where leaves are the bottom of each branch.

However, it is now common to use the term *node* to describe either primitive, node or leaf. Therefore, the term *nodes* will be used throughout this document in reference to either a Function node or Terminal node, or both collectively.

Depending on a given problem, GP may employ Boolean, integer, real, complex, vector, or symbolic values, or a mixture of any of these [42]. As noted in Figure 4.1, evaluation of a GP syntax tree begins with the lower-left node, moves up through the root node, then again to the lower-left of the right branch through the bottom-right node. This is the foundation for syntax trees, tree-based GP, and the resultant expression.

In an analogue to biological natural selection, some nodes of a given GP syntax tree branch are, through the random processes of mutation and reproduction, added, altered, or lost as the offspring of each generation evolve from the prior. Those individual trees that benefit from the evolutionary operations within the context of the selected environment, are better adapted and are therefore more likely to reproduce. Those that fare poorly are less likely to survive and as such, less likely to contribute their code to subsequent generations [44].

4.4.1 Function Set

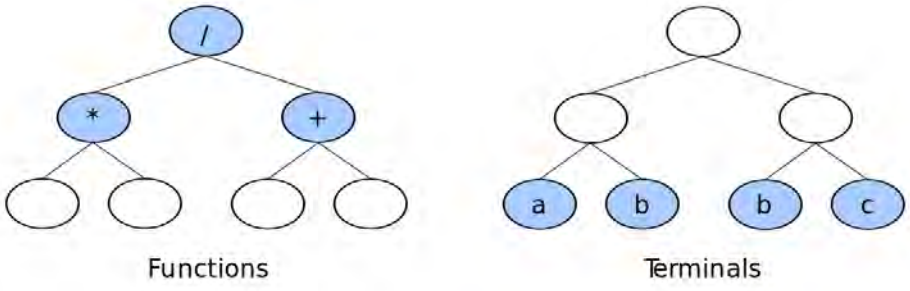


FIGURE 4.2: GP Functions and Terminals.

As shown in Figure 4.2, the Functions employed by a Genetic Program are typically user-designated. Functions may be arithmetic, logical, trigonometric, or domain-specific *operators*. The most often used are $[+, -, *, /, **, sqrt, cos, sin, AND, OR]$. The sum of all Functions employed in a single program space is called the *Function set* [44].

Each Function is associated with an *arity*, that is, the number of operands the operator expects to work with. As with grammar in which a verb anticipates a noun and adverb, the operator $*$ anticipates a variable both before and after itself, meaning it has an arity of 2, as in the simple expression: $a * b$ (Figure 4.3).

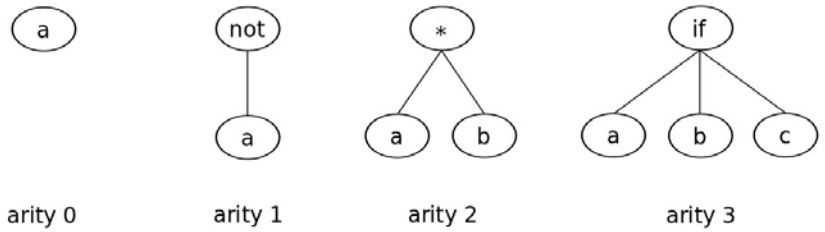


FIGURE 4.3: GP syntax tree arity.

Nearly all operators have an arity of 2, with the exception of the Boolean operator *NOT* that has an arity of 1 and, *IF* that has an arity of 3, as in the expression: $IF\ a\ THEN\ b,\ ELSE\ c$.

4.4.2 Terminal Set

Terminals are typically user-defined *operands*, variables that represent the data features ($[a, b, c, ...n]$), or coefficients ($[1, 2, 3, ...m]$). In a syntax tree representation (Figure 4.1), the Terminals of a program always reside at the end of the branches. The sum of all Terminals employed in a single program space is called the *Terminal set* [44].

An example of a Terminal set is given in Table 4.1 where the SAAO weather data (discussed in Appendix D) offers the Terminal set as variable headers across the top of each column. The variable names were defined by the user.

jd	**sat**	**saz**	**ap**	**rh**	**tmp**	**wm**	**wd**
2454578.03484	35	318	83.09	25.34	11.53	3.8	43.58
2454578.04875	32	314	83.09	24.2	11.86	4.03	58.46
2454578.06264	29	309	83.09	22.25	11.9	3.85	45.9
2454578.07654	26	305	83.09	22.58	12.05	2.68	53.35
2454578.09044	22	302	83.09	22.84	12.12	2.79	55.24

TABLE 4.1: SAAO weather data provides an example of a Terminal set composed of the variables: $[jd, sat, saz, ap, rh, tmp, wm, wd]$

Constants and coefficients to the variables are either implied through the addition of multiple instances of any given variable, as in $a + b + a$ simplified to $2a + b$, or through the random introduction of constants into the program space, at launch of the GP application.

4.4.3 Maximum Tree Depth

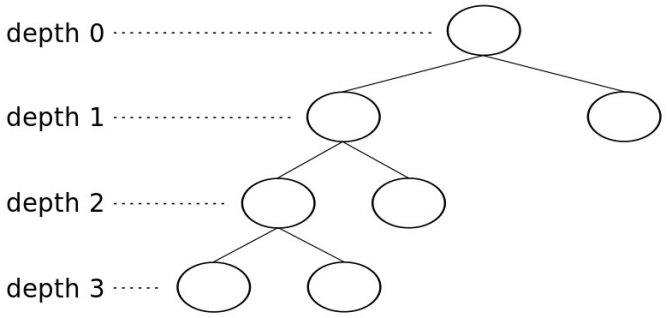


FIGURE 4.4: GP tree depth.

The depth of a tree (Figure 4.4) is determined by the number of edges traversed from the root (depth *0*) to the deepest Terminal (depth *n*). As GP trees tend to grow beyond the size required to solve the problem, a situation referred to as *bloat*, there are various approaches to managing the maximum depth of a GP tree.

In one method the maximum depth of the GP tree is set (often by a user-defined parameter) so as to limit the upper boundary for the number of edges defined in the GP syntax tree. While this limits the search space and potentially prevents the optimal solution from being discovered, it keeps the evolutionary process from experiencing the otherwise inhibiting bloat [42].

In "Resource-Limited Genetic Programming: Replacing Tree Depth Limits" Silva et al. [55] propose controlling total tree size not through a limitation applied to individual tree depth, rather a limit to the total number of nodes per population. In this manner, some trees may be allowed to grow quite large, but others will subsequently be limited in their total growth. This management of total resource allocation allows some trees to explore a complex solution space without the entire population being inhibited by bloat.

As a means of establishing quick estimation, assuming an arity of 2 for all operators, a formula used to derive the maximum possible number of nodes (both Function and Terminal nodes) for GP trees is:

$$nodes = 2^{(d+1)} - 1 \tag{4.1}$$

Where 'd' is the tree depth. The maximum possible number of Function nodes is one less than the number of Terminal nodes, which is one half the total, as given by:

$$nodes = \frac{2^{(d+1)}}{2} \tag{4.2}$$

For example, the maximum number of Functions and Terminals for GP tree depths 1-5 are:

Tree Depth Max 1 = 3 total nodes: 1 Functions, 2 Terminals
Tree Depth Max 2 = 7 total nodes: 3 Functions, 4 Terminals
Tree Depth Max 3 = 15 total nodes: 7 Functions, 8 Terminals
Tree Depth Max 4 = 31 total nodes: 15 Functions, 16 Terminals
Tree Depth Max 5 = 63 total nodes: 31 Functions, 32 Terminals

Having a sense of the potential number of Terminal nodes in a given tree helps estimate the required tree depth in relation to the number of features in a dataset, as deeper trees (more Terminals) are able to search the solution space more rapidly with support for more complex hypotheses. However, more complex hypotheses do not always offer more than subtle improvements in total fitness while introducing potential overfitting and the increased burden of computational overhead. As such, it is important to explore a variety of depths to learn which evolves the overall best hypothesis.

4.4.4 Types of GP Trees

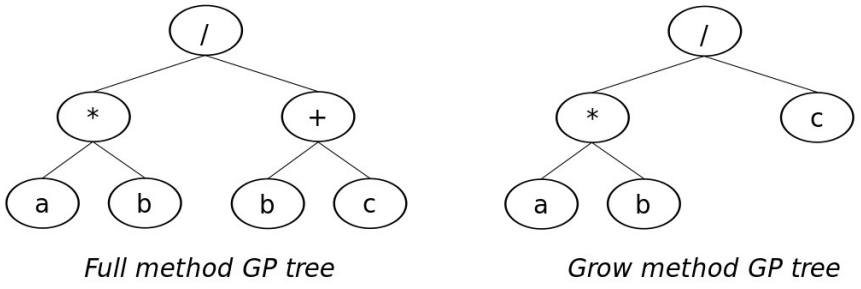


FIGURE 4.5: Full and Grow method GP Trees.

There are two types of GP trees, Full and Grow (Figure 4.5). With *Full* tree, all branches are balanced, meaning each branch reaches the same, user-defined maximum depth. *Grow* trees have unbalanced branches, meaning some branches might reach the maximum depth while others do not. Grow trees are not required to reach the maximum depth with any of their branches.

Grow trees are highly sensitive to the relative sizes of the Function and Terminal sets, the user-defined operators and operands, respectively. For example, if there are significantly more Terminals than Functions, the Grow method will generate shorter trees independent of the user-defined maximum depth. If, however, there are more Functions than Terminals, the Grow method will generate much larger trees that approach the size and shape of those generated by the Full method [44].

In summary, the Grow method generates trees of a greater diversity of shapes and sizes than does the Full method. As such, Grow trees are more likely to find simpler solutions faster [44].

For example, if the desired solution is $a + b + c$ for a total of five elements, there is a probability that a Grow tree might evolve to satisfy this exact expression, element for element. However, a Full tree confined, for example, to depth 3 for a total of 15 elements, will require that some elements are forced to cancel each other in order to arrive at the desired solution. For example, the expression $a - a + a + 1 * b + 2 * c - c$ has 15 elements that simplify to the desired 5-element expression $a+b+c$. Other more complex problems might not be achieved, given the same tree configuration, and the evolutionary algorithm would fail. While less likely to produce the same diversity of solutions as Grow trees, Full trees *do* contribute beneficially to the overall evolutionary process by contributing a larger code base for the initial runs, analogous to the contribution of genetic code in the biological world.

Grow trees alone might produce insufficient diversity of code for a complex problem and also fail to produce the desired hypothesis, or require extended computational time to solve. As such, a combination of both Full and Grow trees, known as *Ramped Half/Half* initialises the first population with 50% Full and 50% Grow, and increments from a minimum to the maximum depth defined by the user, and then applies the Grow method to each population for all subsequent generations. This provides a greater diversity of code from the Full method for the initial populations while taking advantage of the genetic diversity of the Grow method for subsequent, multi-generational evolution. Ramped Half/Half is the most popular method among tree-based GP research [44].

4.4.5 Initial and Subsequent Population Sizes

The creation of the first random population is a blind, stochastic search of the solution space. The user defines the size of the initial population and the number of individual trees. The size of the initial population determines the size of each subsequent population, generation to generation, as maintained throughout an entire GP run.

Both the number of trees and type of trees affect the total diversity of the populations. A greater number of trees offers a greater diversity of potential hypotheses to draw from. However, larger populations increase computational overhead (the number of CPU cycles) thus slowing overall GP runs. Smaller populations decrease diversity and reduce the ability to search a complex solution space.

4.5 Selection

Following the application of genetic operators (reproduction, mutation, crossover), individuals in GP are probabilistically selected based upon their resultant fitness score. That is, more fit individual programs are more likely to provide child programs for the next generation than inferior individual programs. There are several methods for applying selection, including fitness-proportionate, tournament, and rank selection. The most commonly employed selection method is *tournament selection* [44].

4.5.1 Tournament Selection

In tournament selection, the *tournament size* is user-defined and determines the quantity of individual programs chosen at random, per tournament, from the total population of trees. This is typically a small fraction, less than 10%. One tournament is engaged for the selection of each individual program to be copied to the next generation. If only one individual is involved, as with the genetic operator mutation, then only one tournament is conducted for the selection of each program copied to the next generation. If two individuals are involved, as with the genetic operator crossover, then one tournament is conducted for the selection of each program.

For example, if each generation contains 100 programs, and the tournament size is set to 10, then a minimum of 100 programs will have entered the tournament. Each program may be selected more than once in the process of creating the next generation.

Because tournament selection seeks one program from those randomly selected which exhibits the highest fitness score, the degree of improvement is not important. Thereby the fitness is automatically rescaled in order that *selection pressure* on the total population remains constant.

In this manner, *elitism* is avoided, meaning, a single, high scoring individual tree does *not* dominate early generations with its genetic code. This would lead to a loss of diversity and flexibility in the solution space. Rather, tournament selection is meant to amplify small differences in fitness scores in order to give a diversity of programs opportunity to participate in the evolutionary process.

Due to the random selection process there is an inherent level of noise (as defined by Poli and Koza [44]) in tournament selection. While preferring the best programs over the course of an entire run, tournament selection ensures that even average-scoring, individual programs have opportunity to participate in the multi-generational evolution toward the desired solution.

For a tournament selection in Karoo GP in which a maximisation function is anticipated, the selection process seeks the *highest* overall fitness score for the individual trees selected for a given tournament (e.g. 10 trees out of 100):

Simplified, sample code (see Appendix B for the full code) follows:

```
if fitness > tourn_test: # if current fitness > than prior
  tourn_lead = tree_id
  tourn_test = fitness

elif fitness == tourn_test: # if current fitness = prior
  tourn_lead = tree_id
  # the leader transitions to the next tree with the same fitness score

elif fitness < tourn_test: # if current fitness < prior
  # no adjustment made to the leader
  # no adjustment made to the leading fitness score
```

In this example, note the code programmer's comment "no adjustment made to the leading fitness score" for the condition in which the current tree's fitness score is equal to that of one before it. In this particular tournament selection code, the prior tree remains the leader. Another method would be to replace the prior with the new when the scores match.

4.5.2 Parsimony

A tournament selection process could employ a secondary criterion such as *parsimony*. Parsimony works to find the least complicated solution by evaluating the complexity of the current hypothesis. When two non-similar trees exhibit identical fitness scores, the tree with the least number of nodes would be chosen over its contender in order to discourage GP tree bloat and encourage minimal computational overhead [44].

4.6 Fitness Function

Genetic Programming discovers how well a program performs by evaluating its hypothesis through a quantitative analysis and comparing the output to an established ideal, the known solution (symbolic regression) or label (classification). This evaluation process is the *fitness function*, and the outcome is the fitness score, or fitness. The fitness can be measured in several ways:

- Amount of error between the hypothesis' output and the known solution
- Quantity of resource (computational time, cost) required to evolve the hypothesis to a desired, target state
- Accuracy of a hypothesis' ability to recognise patterns or classify objects
- Payoff produced by a game-playing program (model)
- Compliance of a resultant structure to the user specified design criteria

GP fitness functions are unique in evolutionary computation because what evolves is the computer program itself. Therefore, every program must be evaluated for its unique fitness.

Using the Karoo GP platform built in conjunction with this body of research, there were three principal fitness functions and associated tournament selection methods developed and employed: *Matching*, *Classification*, and *Absolute Value Difference*.

4.6.1 Matching Fitness Function Kernel

As described in greater detail in Chapter 5, the *matching* fitness function is a maximisation function which works to discover an exact match between the hypothesis output and the desired solution. This is the simplest of the fitness functions included with Karoo GP.

Simplified, sample code follows:

```
if result == solution: fitness = 1
else: fitness = 0
```

where *solution* is the given target and *result* is the outcome of the evaluated hypothesis. This returns 1 for a 100% match, or 0 for anything less.

4.6.2 Regression Fitness Function Kernel

As described in greater detail in Chapter 5, *regression* is a minimisation function, meaning it seeks a fitness score with the least value.

Simplified, sample code follows:

```
fitness = abs(result - solution)
```

where *solution* is the given target and *result* is the outcome of the evaluated hypothesis. In this case, the desired outcome is the least cumulative fitness score such that the regression algorithm and associated multivariate expression come as close as is possible to the desired outcome, but never likely produce an exact match.

Many other, far more complex fitness functions are possible. Fitness functions are at the heart of all evolutionary algorithms, as they translate the desired outcome into a mathematical solutions space [44].

4.6.3 Classification Fitness Function Kernel

As described in greater detail in Chapter 5, the multiclass Classification fitness function is a maximisation function which works to place data into defined classes. This is the most complex of the fitness functions included with Karoo GP, and the one used for the SKA-SA KAT-7 data evaluation (Appendix B).

Simplified, sample code follows:

```
if solution == 0 and result <= 0:
  fitness = 1 # check for first class

elif solution == class_labels - 1 and result > (solution - 1):
  fitness = 1 # check for last class

elif (solution - 1) < result <= solution:
  fitness = 1 # check between first and last

else: fitness = 0 # no class match
```

where *solution* is the given target and *result* is the outcome of the evaluated hypothesis. This returns 1 for three possible fitness cases, or 0 for a *no match*.

Both of these examples are *maximisation* fitness functions, meaning the fitness function of 1 or 0 for each row of data applied to each tree is added for a total, cumulative fitness score. Therefore, if the given dataset has 100 rows of data, each tree's hypothesis will be processed against all 100 data rows, the score for each row of 1 or 0 added for a cumulative, maximum potential fitness score of 100.

4.7 Genetic Operators

Three decades experience with GP has lead to a fairly consistent application of the genetic operators. Based upon the early work of Koza, configuration of *reproduction* (direct copy), *point mutation*, *branch mutation* (Full and Grow methods), and *crossover* (sexual reproduction) remain relatively consistent in all GP applications [42].

It is important to note that the evolutionary process is able to evolve a desired solution nearly independently of the ratio of the genetic operators applied, with the exception of 100% reproduction which would require that the solution be present in the initial, randomly generated population.

The speed of evolution, that is, the number of generations required to converge on a desired solution is affected by the ratio of the genetic operators, with higher crossover typically resulting in a faster, total process [42].

These four genetic operators are integrated into Karoo GP, and implemented as follows.

4.7.1 Reproduction

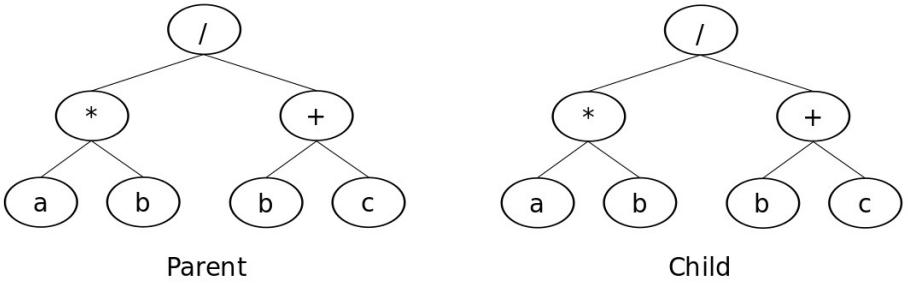


FIGURE 4.6: Genetic operator: reproduction

Through tournament selection, a single tree from the current generation is copied without alternation to the next generation (Figure 4.6). In the biological world, this is analogous to a member of a prior generation directly entering the gene pool of the subsequent (younger) generation.

As each generation is no more than a collection of individual programs, each with its own, potentially unique code, some individuals with desirable traits (and a higher fitness score) must be allowed to propagate unaltered to subsequent generations [44].

4.7.2 Point Mutation

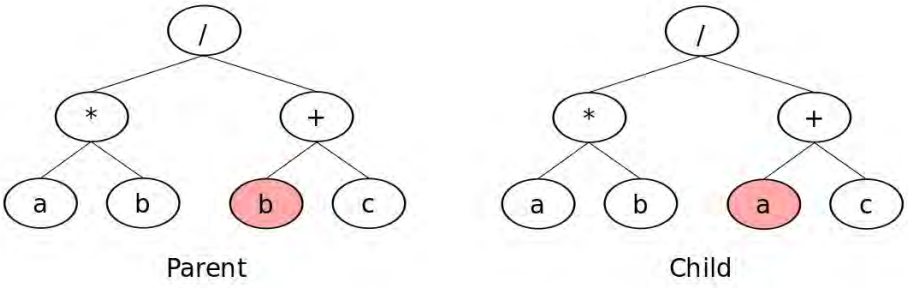


FIGURE 4.7: Genetic operator: point mutation

Through tournament selection, a copy of a program from the current generation mutates before being added to the next generation (Figure 4.7). In the biological world, this is analogous to asexual reproduction. In this method, a single point is selected for mutation while Functions are maintained as Functions and Terminals are maintained as Terminals.Therefore, the size and shape of the tree remains identical [44].

4.7.3 Branch Mutation

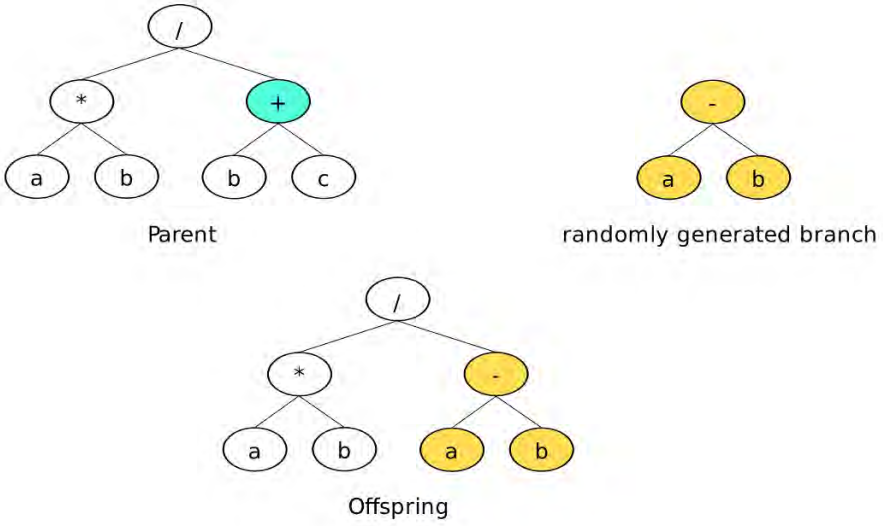


FIGURE 4.8: Genetic operator: branch mutation

As with point mutation, a copy of a tree from the current generation through tournament selection mutates before being added to the next generation. Unlike point mutation, in this method an entire branch is selected (Figure 4.8). If the evolutionary run is designated as Full, the size and shape of the tree remains identical, each node mutated sequentially, where Functions remain Functions and Terminals remain Terminals. If the evolutionary run is designated as Grow or Ramped Half/Half, the size and shape of the

tree might grow smaller or larger, but it may not exceed the maximum depth defined by the user [44].

However, as implemented in Karoo GP, branch mutation often performs in a similar fashion as point mutation due to the greater than 50% probability that the random selection of a single node in the tournament winning tree might be a Terminal. If that Terminal is already at the maximum depth, it can only perform as a point mutation [44].

4.7.4 Crossover

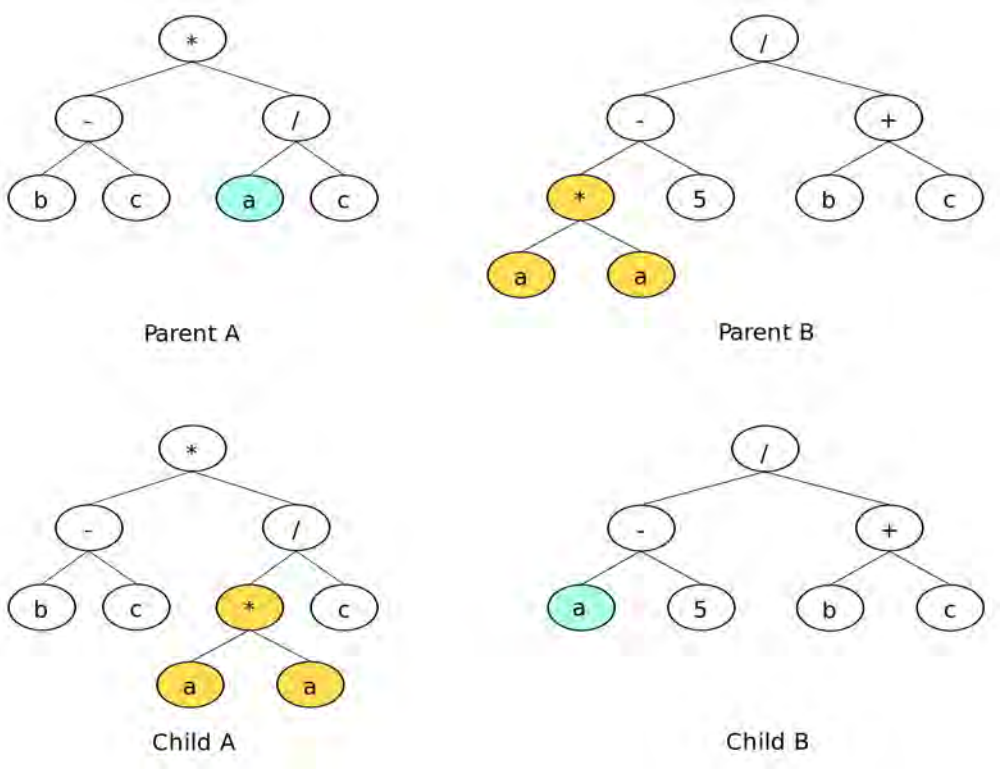


FIGURE 4.9: Genetic operator: crossover

The most often used form of crossover is *subtree crossover*, referred to in this document as simply *crossover*. Through tournament selection, two trees are selected as parents. Within each parent tree a branch is selected (Figure 4.9). Parent A is copied, with its selected branch deleted. Parent B's branch is then copied to the former location of Parent A's branch, and inserted (grafted). The size and shape of the offspring might be smaller or larger than either of the parents, but may not exceed the maximum depth defined by the user [44].

This process combines genetic code from two trees, both of which were chosen by the tournament process as having a higher fitness than the average population. Therefore,

there is a chance their offspring will provide an improved, overall fitness to the next generation.

According to Koza, there are three methods for selecting the trees for the crossover operation [42].

1. The first method uses probability, where $f(S_i(t))$ is the fitness of the solutions S_i and:
$$\sum_{j=1}^{m} f(S_j(t)) \tag{4.3}$$
is the sum m of all the members of the population. The probability that the solution S_i will be copied to the next generation is:
$$\frac{f(S_i(t))}{\sum_{j=1}^{m} f(S_j(t))} \tag{4.4}$$

2. The second method invokes the use of tournament selection wherein the genetic program selects two solutions at random. These solutions *compete* for the best output where the solutions with the highest fitness is retained, the other discarded. This method simulates real-world biological mating in which two members of the same sex compete to mate with a third partner of the opposite sex.

3. The third method simply selects those solutions based upon the numerical ranking of the fitness of a number of solutions.

4.7.5 Other Genetic Operators

Additional genetic operators not addressed in this research include gene duplication, gene dilution, permutation, editing, encapsulation, and decimation [42].

4.8 Termination

Termination is a user-defined parameter typically established in one of three ways:

1. By the total number of elapsed generations;

2. By the elapsed time for a GP run;

3. By the evolution of a hypothesis which demonstrates a quantitative fitness function [44].

Early termination can result in a hypothesis with a low, overall fitness score and inherent inability to model the data accurately. Prolonged runs can lead to program bloat and/or overfitting.

4.9 Preparing for a Genetic Programming Run

There are five steps to preparing for the execution of a GP run, as conducted by the researcher:

1. Define the Terminal set (operands).
2. Define the Function set (operators).
3. Develop the fitness function (kernel).
4. Determine the parameters used for controlling the run (type and quantity of trees, tree depth).
5. Determine the termination criterion (number of generations, duration, or a particular function), and the designated result of the run (regression or classification).

Chapter 5

Karoo GP

5.1 Genetic Programming in Python

Written entirely within the body of this MSc research, *Karoo GP* is an evolutionary algorithm, a Genetic Programming application suite for data analysis. Written in the programming language Python, Karoo GP owes its foundation to the "Field Guide to Genetic Programming" by Poli, et al [44].

Karoo GP is scalable, with multicore support, and designed to readily work with real-world data. As a teaching tool, it enables instructors to share step-by-step how an evolutionary algorithm arrives to its solution. As a hands-on learning tool, Karoo GP supports rapid, repeatable experimentation.

Karoo GP includes a desktop application with an intuitive user interface and a fully scriptable server application with user-defined, default parameters and command-line arguments. The included User Guide (Appendix A) offers system requirements, a crash-course in GP, and use of Karoo GP for both the novice and advanced user.

5.1.1 Data Format

As with all data analysis tools, Karoo GP anticipates the dataset to be structured in a particular manner. Each row is a data point, a recording of information from the real world. Each column holds features, parameters which describe the data point in two or more ways. There is no theoretical limit to the number of features nor rows, however, large datasets, either many rows or many columns, or both, will be computationally expensive.

Karoo GP has been run extensively on datasets with 10,000 rows and a dozen columns. But in machine learning, it is not uncommon to have hundreds, even thousands of features. The rows do not require headings, that is, unique identifiers. However, time series data can be identified by the unique time signature, often presented in the leftmost column.

Constants are added to the dataset as independent columns, randomly selected during run-time as with any of the features.

The top of each column is given a label, a variable name, and integer or decimal value in the case of constants. The rightmost column holds the solutions. Karoo GP develops hypotheses which solve for those values, row by row, through the entire dataset. The solutions column must be labelled with the lower case letter 's'.

5.2 Using Karoo GP

At launch, Karoo loads one of the three datasets associated with one of the three built-in toy models, or a user-defined dataset as set by command-line argument. In addition, the user-defined Functions (operators) and Terminals (operands) are loaded from external .csv files.

Karoo GP generates the first population of trees, evaluates, selects the most fit trees, and then evolves copies for each subsequent generation. This continues until either the maximum number of generations or a particular solution is reached.

In the following, I describe the use of the desktop application, where identical configuration parameters may be invoked in the server application as command line arguments.

5.2.1 Desktop User Interface

When the desktop application is launched, the user will be requested to set a number of run-time parameters:

```
Select (r)egression, (c)lassification, (m)atching, (p)lay (default m):
Select (f)ull, (g)row, or (r)amped 50/50 method (default r):
Enter depth of the initial population of Trees (default 3):
Enter maximum Tree depth (default matches initial):
Enter minimum number of nodes for any given Tree (default 3):
Enter number of Trees in each population (default 100):
```

```
Enter max number of generations (default 10):
Display (i)nteractive, (g)eneration, (m)iminal, (s)ilent (default m):

Type ? at any (pause) to review your options, or ENTER to continue.
```

The Karoo GP interface provides a HELP menu, with a number of user-defined parameters, including the ability to modify the tournament size, minimum number of nodes, balance of genetic operators, and the number of engaged CPU cores. The user may at the end of any generation display the generation ID, the full population of trees and their associated multivariate expressions, or the trees with the best fitness scores for that particular generation. The user may also evaluate any given tree against the test data, continue the run by adding more generations, or save a population to disk.

The User Guide covers each of the parameters described above in detail.

5.2.2 Server Configuration

With the Karoo GP server application, the user can configures the run from the command line, appending a series of arguments which override the default values, as follows:

```
-ker [r,c,m] fitness function: (r)egression, (c)lassification, (m)atching
-typ [f,g,r] Tree type: (f)ull, (g)row, (r)amped half/half
-bas [3...10] maximum Tree depth for the initial population
-max [3...10] maximum Tree depth for the entire run
-min [3...100] minimum number of nodes
-pop [10...1000] maximum population
-gen [1...100] number of generations
```

A sample execution script might appear as follows:

```
$ python karoo_gp_server.py -ker c -typ r -bas 4 -fil [filename].csv
```

5.3 Novel Features of Karoo GP

Karoo GP includes a number of novel features in the form of unique code algorithms, particular functions, and the user interface. A few of these are explained, as follows:

5.3.1 Minimum Number of Nodes

The user-defined *Min Node Count* defines the least number of Functions and Terminals allowed in any given tree. This parameter was introduced to Karoo GP in order to shape but not force the evolutionary process to search a particular solution space by establishing a lower bounds on the evolutionary landscape.

The greater the minimum number of Function (operators) and Terminal (operands) nodes, the higher degree of complexity to be found in the expressions generated by the GP as the gene pool, from which the tournament selection operates, is derived from those trees that meet the minimum number of nodes criterion.

Experiments in modifying the Min Node Count parameter between each generation resulted in a rapidly evolving landscape of hypotheses. By raising the minimum number of nodes value slowly, subsequent generations are quick to adapt, providing increasingly complex trees while rapidly recovering from an otherwise reduced gene pool.

A rapid increase in minimum number of nodes, say from node count 3 to 13 against a tree depth of 3 (of maximum 15 possible nodes) sometimes results in a population crash, termination of all trees, and the end of the run. However, just one or two trees might survive and in less than a half dozen generations, a substantial diversity of hypotheses again will prevail.

5.3.2 Population Save, Modify, and Reload

At the end of each generation, the current generation is appended to a .csv file. A 10 generation run of 100 trees will result in a .csv file with 1000 trees total. The final population is automatically written to disk and named accordingly. The user may then copy these files to an archive for future review and/or reintroduction to Karoo for further evolution.

Saved populations can be opened in a spreadsheet, manipulated, and re-loaded into Karoo. This enables launching Karoo with a *seed population*, rather than a randomly generated population, which is done when key correlations between certain data parameters are already known and introduced into the evolutionary process as a starting point.

5.3.3 Five levels of run-time monitoring

Karoo GP is unique in its offering of a simple, intuitive, yet powerful text-based user interface. The Karoo GP interface offers 5 levels of visual feedback during any given data run, as follows:

- *timer* - provides the user with the elapsed time required for each round of total tree population evaluation, the most CPU intensive part of Karoo GP. This is used strictly for the purpose of determining the most efficient quantity of CPU cores to engage on a given server.
- *debug* - provides the user with a detailed analysis of each execution of the genetic operators point mutation, brand mutation, and crossover. As every tree in Karoo GP is held in a Numpy array, the mutation of the array itself is observed. This is a means by which every genetic operator may be validated. An example of the output of this mode is given in *A system for automated array addressing*, below.
- *interactive* - provides on-screen summaries of the application of the genetic operators, construction of the gene pool, and tournament selection, including identification of each node of each tree that undergoes reproduction, mutation or cross over.
- *generation* - pauses at the completion of each generation in order to give the user opportunity to review the process, adjust the parameters, and then terminate the run or continue.
- *miminal* - moves through the entire data run without pause, but does display the expression derived from each tree.
- *silent* - designed for remote servers, this mode requires the least overhead as it displays the summary of each generation only.

5.3.4 User-defined Fitness Functions

The matching, regression, and classification fitness functions built-in to Karoo GP are generic in that they are not tailored to any particular problem. Advanced applications of Karoo GP may require a custom fitness function which seeks a particular outcome. Developing custom fitness functions to match custom problem types in GP software applications is typically a cumbersome task, involving the development of custom methods that match the framework-specific code. Users often must develop their own boilerplate

(template), which is time wasted on what would otherwise be productive problem solving [56].

Karoo GP is unique in the simplicity of application of a new fitness function, in that code need only be introduced in the tournament and test methods. Karoo GP offers a template for each, with further details provided in the User Guide.

5.3.5 A System for Managing Elements in Scaling Arrays

Karoo GP uses *axis-1* Numpy arrays, arrays which are turned 90 degrees such that they expand horizontally rather than vertically. There are 13 rows assigned to each array, which when combined describe all facets of any given tree: the sequential identification number (ID) for each tree within a given generation, the tree type (Full or Grow), the initial tree depth, the ID for each node within any given tree, the depth of each node within its host tree, the node label (root, Function, Terminal), the parent to any given node, the arity of any given node, and whether that node has one, two, or three children, their unique IDs, and the fitness for each tree as generated during its evaluation against the fitness function.

If these trees were without change, that is, if they were unaffected by genetic operators, then they would be generated once and remain static throughout the entire Karoo GP run. However, the intent of evolutionary computation is to modify a select few of these trees in each generation. The change can be as simple as the mutation of Function $+$ to Function $-$ or Terminal a to Terminal b. In either of these cases the structure of the tree, and subsequently the array, remains identical.

However, if a Function mutates to a Terminal, all nodes below that Function are lost. If a Terminal mutates to a Function, a new set of nodes will be appended. With the genetic operator crossover, entire sections of trees are swapped. In each of these cases, the size and shape of the tree might be affected and, thereby, the size of the array which defines that tree might also be modified.

If all modifications were made at the end of the array, then this would be a relatively simple process. However, the mutation and crossover processes will often affect the middle of a tree, thereby the middle of its array, and the recursive chase through the tree which uses each node ID and parent-child relationship to generate its inherent multivariate expression.

For example, inserting one additional node in the middle of an array will cause all nodes to the right of that new node to be shifted by one space to the right. Therefore, their unique node IDs would no longer match their position in the array, causing parent-child

relationships to be lost, and the multivariate expression derived from that tree would be incorrect or fail completely.

Therefore, in the development of Karoo GP it was required that an algorithm be developed capable of tracking the position of each node according to its position in the axis-1 array and, at the same time, its child relationships no matter how the array is modified.

A single algorithm was developed to determine the correct position of a child node, already in place or to be inserted, no matter the depth nor complexity of the tree, nor position in which a branch is inserted or removed. The Karoo GP method *fx_evo_c_buffer* in the *karoo_gp_base_class.py* library uses this formula to renumber all nodes in any tree which has been modified.

$$child = node_id + parent\_arity\_sum + prior\_sibling\_arity - prior\_siblings \tag{5.1}$$

This algorithm is implemented as described in the following pseudo-code, where the value *node* (Terminal or Function, but not root) is the integer identification for any node in the given tree, and is passed to this method:

```
for n in range(1, tree_depth):

  if node_depth[n] == parent_node_depth:
    if node_arity[n] != '': parent_arity_sum =+ node_arity

  if node_depth[n] == current_node_depth and node_id[n] < current_node_id:
    if node_arity[n] != '': prior_sibling_arity =+ node_arity[n]
      prior_siblings = prior_siblings + 1

child_id = node + parent_arity_sum + prior_sibling_arity - prior_siblings
```

If one enters the *debug* mode of Karoo GP, a representation of the genetic operators applied is displayed. The following is a simplified version of a branch mutation, where only the initial, unaltered tournament winner, randomly generated branch, and final, new tree are presented. In the real workings of Karoo GP, with each addition of a node from the randomly generated branch into a copy of the tournament winner, the resulting tree is displayed.

This is the unaltered tournament winner, of which a copy is made:

```
[['TREE_ID' '34' '' '']
```

```
 ['tree_type' 'g' '' '']
 ['tree_depth_base' '2' '' '']
 ['NODE_ID' '1' '2' '3']
 ['node_depth' '0' '1' '1']
 ['node_type' 'root' 'term' 'term']
 ['node_label' '+' 'c' 'c']
 ['node_parent' '' '1' '1']
 ['node_arity' '2' '0' '0']
 ['node_c1' '2' '' '']
 ['node_c2' '3' '' '']
 ['node_c3' '' '' '']
 ['fitness' '1.0' '' '']]
```

When this Numpy array is flattened, its expression is: $c + c$

This is the randomly generated branch to be inserted at node 2 (operand c) of the copy of the tournament winner:

```
[['TREE_ID' 'mutant' '' '' '' '']
 ['tree_type' 'g' '' '' '' '']
 ['tree_depth_base' '2' '' '' '' '']
 ['NODE_ID' '1' '2' '3' '4' '5']
 ['node_depth' '0' '1' '1' '2' '2']
 ['node_type' 'root' 'func' 'term' 'term' 'term']
 ['node_label' '-' '-' 'c' 'a' 'b']
 ['node_parent' '' '1' '1' '2' '2']
 ['node_arity' '2' '2' '0' '0' '0']
 ['node_c1' '2' '4' '' '' '']
 ['node_c2' '3' '5' '' '' '']
 ['node_c3' '' '' '' '' '']
 ['fitness' '' '' '' '' '']]
```

When this Numpy array is flattened, the expression is: $a - b - c$

This is the copy of the tournament winner after the new branch is inserted:

```
[['TREE_ID' 'new' '' '' '' '' '' '']
 ['tree_type' 'g' '' '' '' '' '' '']
 ['tree_depth_base' '2' '' '' '' '' '' '']
 ['NODE_ID' '1' '2' '3' '4' '5' '6' '7']
```

```
['node_depth' '0' '1' '1' '2' '2' '3' '3']
['node_type' 'root' 'func' 'term' 'func' 'term' 'term' 'term']
['node_label' '+' '-' 'c' '-' 'c' 'a' 'b']
['node_parent' '' '1' '1' '2' '2' '4' '4']
['node_arity' '2' '2' '0' '2' '0' '0' '0']
['node_c1' '2' '4' '' '6' '' '' '']
['node_c2' '3' '5' '' '7' '' '' '']
['node_c3' '' '' '' '' '' '' '']
['fitness' '' '' '' '' '' '' '']]
```

When this Numpy array is flattened, its new expression is: $a - b - c + c$

This branch mutation process may be represented by a syntax tree, as in Figure 5.1.

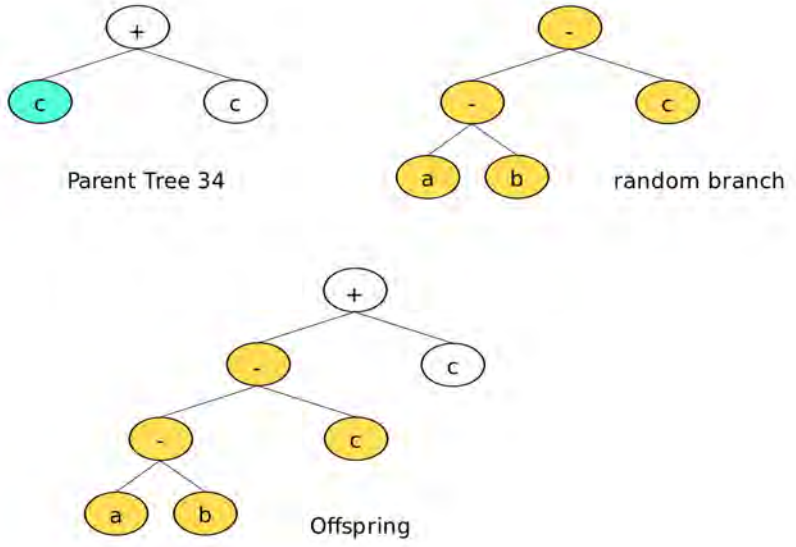


FIGURE 5.1: Genetic operator branch mutation demonstrated as a copy of Tree 34 receives the addition of a randomly generated branch, producing a unique offspring for the next generation.

The mutated copy of Tree 34 yields $(a) - (b) - (c) + (c)$ which is simplified to $a - b$.

5.4 Validation of Karoo GP

As Karoo GP was written from the ground-up for this body of research, it required validation against classic problems whose solutions are well established and easily reproducible prior to application to the SKA-SA KAT-7 dataset. Karoo GP was therefore tested against three unique problems in order to validate three unique fitness kernels: a

simple *matching problem*, the classic symbolic regression problem *Kepler's Third Law of Planetary Motion*, and the popular *Iris Dataset* classification problem.

These tests and associated results are described in the following sections.

5.4.1 A Matching Problem

The *matching* fitness function and associated, flexible, hand-crafted dataset incorporated in Karoo GP were given foundation as an early-stage, baseline functionality test. The matching kernel remains an easily understood demonstration of GP, and a quick test of overall system functionality, as it invokes initial population generation and subsequent multi-generational evolution, engaging the majority of Karoo GP's code foundation.

With the matching fitness function, the evolutionary process must generate an expression that contains *all* data points in a given dataset and the relationship they maintain with each other. Therefore, an exact match for *all* data rows is required in order to achieve 100% accuracy across the entire dataset, or the hypothesis fails.

However, there will be expressions which resolve one or two data rows, providing a local solution but fail the full dataset and therefore lack a global solution. For example, if we take:

$$s = a + b + c \tag{5.2}$$

And set $a = 0, b = 1, c = 2$ we will find the solution s = 3 by means of the simplified, binomial expression:

$$s = b + c \tag{5.3}$$

Yet, clearly, this function does not work when we set the variable a to something other than zero, as demonstrated in rows 2-5 in Table 5.1:

	a	**b**	**c**	s
row 1:	0	1	2	3
row 2:	1	2	3	6
row 3:	2	3	4	9
row 4:	3	4	5	12
row 5:	4	5	6	15

TABLE 5.1: A matching fitness function used by Karoo GP

There are other potential local minima which could easily confuse the evolutionary process:

row 2: $b * c$
row 3: b^a
row 4: $a * b$

With the addition of floating points, coefficients, and trigonometric Functions the number of unique solutions for both local minima and a working, global hypothesis is diverse and exciting. For example, while the anticipated solution to the above dataset is $s = a + b + c$, given the operators $+, -, *, /$ the evolutionary process often presents more creative solutions, some more, some less computationally expensive than others, as follows:

3*b
a + 2*b + 1
2*a + c + 1
-a*b + a*c + b + c

All of these solutions are 100% accurate in their results, against all data rows in Table 5.1. In this particular run of 100 trees per generation, an accuracy test built-in to Karoo GP was applied to two trees. Tree #92 was demonstrated to have produced a 100% accurate solution across the entire dataset, as follows:

Tree 92 yields (raw): $(b) + (a) + (c)$
Tree 92 yields (sym): $a + b + c$
data row 0 yields: 3.0
data row 1 yields: 6.0
data row 2 yields: 9.0
data row 3 yields: 12.0
data row 4 yields: 15.0
Tree 92 has an accuracy of: **100.0**

While the very next tree (#93) did not score as well:

Tree 93 yields (raw): $(c) + (c)$
Tree 93 yields (sym): $2 * c$
data row 0 yields: 4.0
data row 1 yields: 6.0
data row 2 yields: 8.0
data row 3 yields: 10.0
data row 4 yields: 12.0
Tree 93 has an accuracy of: **20.0**

Clearly, this simple fitness function both demonstrates the functionality of Karoo GP while enabling the user to better understand GP through a simple, easily executed operation.

5.4.2 A Regression Problem

In the early 1600s Johannes Kepler proposed three laws of planetary motion derived from data obtained from his mentor Tycho Brahe. While Kepler's explanation for the underlying reasons for the motions of the planets was later proved invalid, the three laws remain accurate descriptions of the motions of all planets in our solar system and any gravitationally bound satellite in orbit around any body.

Kepler's three laws of planetary motion [57] can be described (verbatim) as follows:

- *Law of Ellipses* - "The path of the planets about the sun is elliptical in shape, with the center of the sun being located at one focus."
- *Law of Equal Areas* - "An imaginary line drawn from the center of the sun to the center of the planet will sweep equal areas in equal intervals of time."
- *Law of Harmonies* - "The ratio of the squares of the periods of any two planets is equal to the ratio of the cubes of their average distances from the sun."

Unlike Kepler's first two laws that describe the motion of a single planet, Kepler's third law of planetary motion, the Law of Harmonies compares the orbital period p to the average orbital radius r of one planet to the orbital period and average orbital radius of one or more other planets, all in orbit around the Sun. As is presented in Table 5.2, the relationship established is the square of the periods to the cube of the average orbital radius from the sun.

Kepler's original work compared only Mars to Earth, as follows:

Planet	**Period (s)**	**Avg. Radius (m)**	s^2/m^3
Earth	$3.156 * 10^7 s$	$1.4957 * 10^{11} m$	$2.977 * 10^{-19}$
Mars	$5.93 * 10^7 s$	$2.278 * 10^{11} m$	$2.975 * 10^{-19}$

TABLE 5.2: Orbital period s measured in seconds and average orbital radius m measured in meters from the sun, for Earth and Mars, as applied by Kepler

The ratio s^2/m^3 is the same for Earth and for Mars. If this is computed for the other planets (switching the unit for time from seconds to years, and distance from meters to astronomical units, the distance from the Earth to the Sun), the result is an identical

ratio for each planet, as shown in Table 5.3. The apparent perturbations are the result of working with only two decimal places in both measurement and computation [58].

Planet	**Period (yr)**	**Avg. Radius (au)**	$p^2/r^3(yr^2/au^3)$
Mercury	0.241	0.39	0.98
Venus	.615	0.72	1.01
Earth	1.00	1.00	1.00
Mars	1.88	1.52	1.01
Jupiter	11.8	5.20	0.99
Saturn	29.5	9.54	1.00
Uranus	84.0	19.18	1.00
Neptune	165.0	30.06	1.00
Pluto	248.0	39.44	1.00

TABLE 5.3: Orbital period p and average orbital radius r for all planets in our solar system

At the surface, this test appears very simple as the number of parameters is just 2: period (time) and radius (distance). Yet, Kepler's Third Law, which invokes a minimisation function in symbolic regression presents an interesting conundrum, as follows.

In some problem domains, evolutionary algorithms may tend to converge on a premature solution, a *local minimum*. This is true for GP applied to Kepler's Third Law where t/t (period divided by period) of course equals 1.0. While the numeric result of this simple function is a correct match for five of the planets, the correct solution for Mercury is 0.98, 0.99 for Jupiter, and 1.01 for Venus and Mars (as given in Table 5.3). Therefore, 1.0 does not represent the correct solution nor the related orbital dynamics between all the planets of our solar system.

While an earlier version of Karoo GP was able to solve Kepler's Third Law, it often required 30, 40, 50 or more generations for the evolutionary process to discover the additional properties required to pull the given trees out of the local minimum and toward a solution that would satisfy all nine planets, not just five.

One example of the output of a local minimum solution is r/r where r is the average orbital radius of the planet as measured in astronomical units, resulting in 1.

Running 50 generations for such a simple problem is not a long-term solution as the computational resources required are too high. By adjusting the user-defined parameter Min Node Count, the hypothesis is forced to work within the minimum number of nodes, for its evolved multivariate expression.

For example, the local minimum example r/r has just three elements: r, /, and r, where each operator and each operand count as one element. However, the correct solution $p * p/r * r * r$ has a total of nine elements.

In an effort to resolve this issue, the following options were considered:

1. If any given tree falls below the user-defined number of nodes (node count, not depth count), that tree is forced to mutate over and over again until it is at or above the prescribed node count. This was deemed to be convoluted (unnatural), as this is not how this occurs in the biological world.

2. Nudge the kernel-level fitness function (higher or lower, depending upon invocation of a maximising or minimising function) such that a tree whose node count is below the user-defined number is less likely to be selected in a Tournament. This was a complicated solution which was analogous to the alteration of the rules of engagement of evolution, rather than the environment in which biological organisms survive. This would required a fairly invasive alteration of every fitness function kernel, those currently in the code and all developed by future users. It is desirable to retain relatively simple fitness functions whose processes can be represented through $>$, $<$ and $=$ relationships.

3. Introduce a gene pool from which the tournament selection draws its trees, and then block any tree whose node count is lower than the user-defined Min Node Count from entering the gene pool. As all four of the genetic operators perform *only* on trees chosen by tournament selection, the filtering of trees prior to the Tournament automatically moves trees through *all* fitness functions which meet the Min Node Count criteria, with minimal processing overhead and no additional code development for future kernel developers. This process is analogous to the change in real-world environmental conditions, such as a change in soil water content or an unusually cold winter, that limits individuals in a current generation from being selected for the next. But it does not inhibit those traits from again appearing in the subsequent generation by means of point mutation, branch mutation, or crossover.

With the third approach, the Min Node Count parameter and subsequent filtration of the gene pool encourages but does not force a solution upon the next generation. As such, trees may evolve in subsequent populations which do have node counts lower than the user-defined minimum threshold. For example, if a user sets the Min Node Count to 7, all trees which enter the gene pool will have at least seven nodes (elements in an expression) but the next generation may again produce trees with node counts lower than seven.

In confirmation, the addition of the user-defined, Min Node Count parameter does *not* enable Karoo GP to automatically guarantee Kepler's Third Law in a finite quantity of generations. The user must yet experiment with Full, Grow, or Ramped Half/Half trees and Maximum Depth in order to arrive to the correct p^2/r^3 solution efficiently.

A series of test runs were conducted with the following Karoo GP configuration parameters, values in set notation [...] referring to a run for each possible combination:

kernel = a
tree_type = r
tree_depth_max = [3, 4, 5]
tree_depth_min = [3, 4, 5 ... 9]
tree_pop_max = 100
generation_max = 30
tourn_size = 10
cores = 2
precision = 4

With the Maximum Tree Depth set to 3 and Min Node Count to 3, Karoo GP failed in 30 generations as it again converged on the solution $r/r = 1$. However, with a setting the Maximum Tree Depth to 5 for a possible $2^{d+1} - 1 = 63$ nodes and Min Node Count to 9, Karoo converged on the correct solution p^2/r^3 in less than eight generations (Table 5.4).

GP tree	**model**
1	p^2/r^3
2	p^2/r^3
3	p^2/r^3
...	
88	p^2/r^3
92	p^2/r^3
94	p^2/r^3

TABLE 5.4: First successful run of Karoo GP against the Kepler dataset

This confirms the introduction of the Min Node Count setting as a valid modification to the code in that its assists the evolutionary process in avoiding the local minimum, while retaining the essential, stochastic nature of evolutionary computation.

5.4.3 A Classification Problem

In the 1920's, botanists collected measurements on the sepal length, sepal width, petal length, and petal width of 150 iris specimens, 50 from each of three species: Iris Setosa,

Iris Versicolour, and Iris Virginica. In 1936 R.A. Fisher hand-derived a function which separated these species [59].

As noted at the University of California's *Machine Learning Repository*, this is now known as the classic *Iris multivariate dataset* and is a must-solve problem for all developers of classification algorithms.

5.4.3.1 Classification by Hand

180 MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS

Table I

Iris setosa				*Iris versicolor*				*Iris virginica*			
Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width
5·1	3·5	1·4	0·2	7·0	3·2	4·7	1·4	6·3	3·3	6·0	2·5
4·9	3·0	1·4	0·2	6·4	3·2	4·5	1·5	5·8	2·7	5·1	1·9
4·7	3·2	1·3	0·2	6·9	3·1	4·9	1·5	7·1	3·0	5·9	2·1
4·6	3·1	1·5	0·2	5·5	2·3	4·0	1·3	6·3	2·9	5·6	1·8
5·0	3·6	1·4	0·2	6·5	2·8	4·6	1·5	6·5	3·0	5·8	2·2
5·4	3·9	1·7	0·4	5·7	2·8	4·5	1·3	7·6	3·0	6·6	2·1
4·6	3·4	1·4	0·3	6·3	3·3	4·7	1·6	4·9	2·5	4·5	1·7
5·0	3·4	1·5	0·2	4·9	2·4	3·3	1·0	7·3	2·9	6·3	1·8
4·4	2·9	1·4	0·2	6·6	2·9	4·6	1·3	6·7	2·5	5·8	1·8
4·9	3·1	1·5	0·1	5·2	2·7	3·9	1·4	7·2	3·6	6·1	2·5
5·4	3·7	1·5	0·2	5·0	2·0	3·5	1·0	6·5	3·2	5·1	2·0
4·8	3·4	1·6	0·2	5·9	3·0	4·2	1·5	6·4	2·7	5·3	1·9

FIGURE 5.2: The original Iris dataset, 1936 [R.A. Fisher].

Taxonomists conduct precise measurements of flora and fauna in order to differentiate species. In his paper "The use of multiple measurements in taxonomic problems" [59], Fisher opens with "W H E N [sic] two or more populations have been measured in several characters, $x1, ..., x8$, special interest attaches to certain linear functions of the measurements by which the populations are best discriminated."

In the subsequent section II, **Arithmetical Procedure**, where his "Table I" refers to Figure 5.2, Fisher states [59]

> Table I shows measurements of the flowers of fifty plants each of the two species iris *setosa* and *I. versicolor*, found growing together in the same colony and measured by Dr E. Anderson, to whom I am indebted for the use of the data. Four flower measurements are given. We shall first consider the question: *What linear function of the four setosa [flower] measurements will maximize the ratio of the difference between the specific means to the standard deviations within species?*

$$X = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 + \lambda_4 x_4 \tag{5.4}$$

In section IV Fisher goes on to identify a means of seeking a mathematical function to correctly separate the iris species with the **Analogy of Partial Regression**.

> The analysis of Table VI suggests an analogy of some interest. If to each plant were assigned a value of a variate y, the same for all members of each species, the analysis of variance of y, between the portions accountable by linear regression on the measurements $xl, ..., x4$, and the residual variation after fitting such a regression, would be identical with Table VI, if y were given appropriate equal and opposite values for the two species.

Based upon the data meticulously gathered by his associate Dr. Anderson (Figure 5.2), Fisher was in 1936 hand-crafting regression algorithms which cleanly separated the three species of iris flowers. Eighty years later, we are now able to apply machine learning to derive similar functions through an evolutionary process which mimics the very biological foundation that Fisher sought to describe.

5.4.3.2 Classification with Genetic Programming

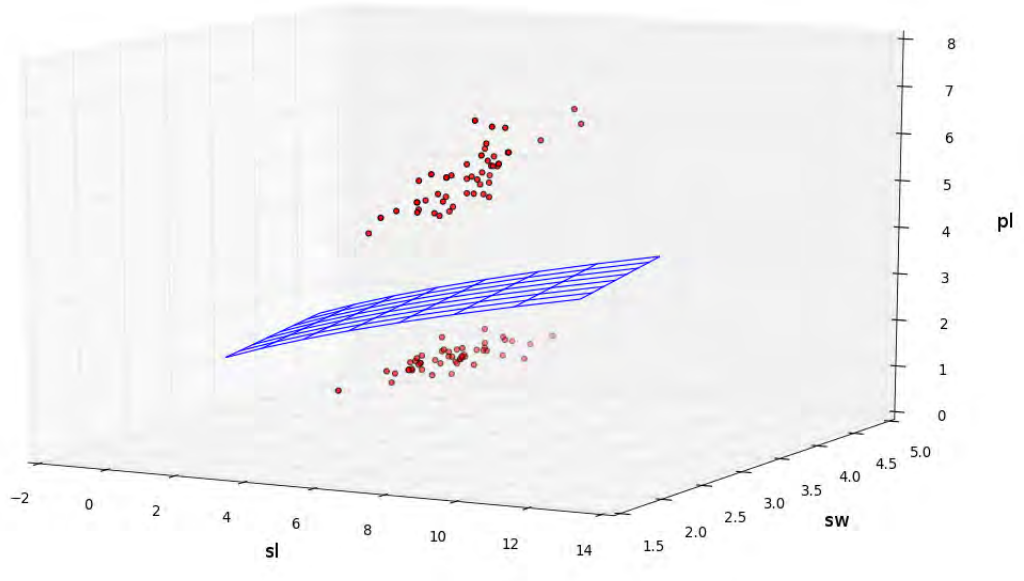


FIGURE 5.3: Plot of two Iris species against the GP evolved function $sl = -sw + pl^2$.

In working with the same dataset as did R.A. Fisher, we learn that one class is linearly separable from the other two, while the latter two are *not* linearly separable from each other. With the application of Genetic Programming as a tool for classification, there are two ways to approach this problem:

1. Compare only two species at a time, in a round-robin such that we work with sl/sw, sl/pl, and finally sw/pl; or

2. Seek a nonlinear function using a kernel which supports multiclass classification.

Given the four features [sl, sw, pl, pw], whatever relationship between these features exists in lower dimensions of comparison will be retained in higher dimensions. Therefore it is safe to test a lower dimension of the feature set as long as the features selected are decisive in the identification of the flower species [60]. Thus, column *pw* of the Iris dataset was removed from the dataset, reducing the dimensionality by one degree, in order to facilitate visualisation of the output on a 3-dimensional graph (Figure 5.3).

Working with just two Iris flower species (binary classification) and three features [sl, sw, pl], Karoo GP generated both linear and nonlinear functions that provide 100% accurate classifications, as is represented in the Figure 5.3 scatter plot.

When given all four Iris flower species (multiclass classification), in just 10 generations of 100 trees per population and a Max Tree Depth of 3, Karoo GP converged on the hypothesis $-sl/sw**2 - sw + c$ which provided an accuracy of 93.33%.

Table 5.5 provides a row-by-row evaluation of the output of the hypothesis $-sl/sw^2 - sw + pl$ against the test dataset, as executed *internal* to Karoo GP:

Tree 99 yields (raw): $(pl) - (sw) - (sl)/(sw)/(sw)$
Tree 99 yields (sym): $-sl/sw^2 - sw + pl$

data row 12 predicts class: 0 (0) as $-2.0683 <= 0$
data row 13 predicts class: 1 (1) as $0 < 0.4867 <= 1$
data row 14 predicts class: 0 (0) as $-2.1889 <= 0$
data row 15 predicts class: 0 (0) as $-2.3082 <= 0$
data row 16 predicts class: 2 (1) as $1.0238 > 1$
data row 17 predicts class: 2 (2) as $1.2092 > 1$
data row 18 predicts class: 2 (2) as $1.9444 > 1$
data row 19 predicts class: 2 (2) as $1.8028 > 1$
data row 20 predicts class: 2 (2) as $1.216 > 1$
data row 21 predicts class: 1 (1) as $0 < 0.639 <= 1$
data row 22 predicts class: 1 (1) as $0 < 0.675 <= 1$
data row 23 predicts class: 1 (1) as $0 < 0.4451 <= 1$
data row 24 predicts class: 2 (2) as $1.2652 > 1$
data row 25 predicts class: 0 (0) as $-1.9152 <= 0$
data row 26 predicts class: 2 (1) as $1.577 > 1$
data row 27 predicts class: 1 (1) as $0 < 0.8215 <= 1$
data row 28 predicts class: 2 (2) as $1.6044 > 1$

TABLE 5.5: The output of Karoo GP as it works to predict one of the three possible classes for each data point (row) in the *test* subset of the Iris flower dataset

In Table 5.4 data rows 16 and 26 failed to provide the correct classification. Subsequently, less than 100% accuracy was achieved. As GP is a stochastic process, it cannot guaranteed perfect results. The implementation of additional generations or execution of new runs will provide various results, some which fare better than others.

5.5 Conclusion

Given the results of the three tests built-in to Karoo GP: *A Matching Problem*, *A Regression Program*, and *A Classification Problem*, Karoo GP is demonstrated to having solved real-world problems efficiently and accurately. The infrastructure developed provides a flexible environment in which to develop a multitude of fitness functions and associated selection processes for a diversity of approaches to problem solving through GP.

There was an additional level of success with Karoo GP in that the evolutionary processes which unfolded were anticipated by the literature. For example, in early experimental runs against the *Boosted PC* dataset, Karoo GP was configured to pause at each increment of 10 generations in order to observe the current state of the evolved hypotheses. There was a noted reduction in the number of unique hypotheses through this progression. At 10 generations, nearly every tree offered its own, unique solution. But at 50 generations, a convergence to less than a half-dozen hypotheses was not uncommon, all of which generated identical Precision-Recall scores. This confirmed that GP was unfolding as intended, moving toward an evolved solution.

Concerning the degree of linearity of the evolved expressions, by adjusting the depth of the trees, the complexity of the expressions varied as anticipated. Higher-order multivariate expressions were generated with the application of higher tree depths, while lower-order, sometimes linear functions were generated with the application of lower tree depths. This demonstrates the function of user-defined evolutionary constraints within Karoo GP.

Given an understanding of the data structure, the user is able to anticipate the appropriate input parameters and configure Karoo GP appropriately. Thus, Karoo GP is demonstrated to be a flexible, scalable tool for a diversity of data types and depths.

Chapter 6

Machine Learning Applied to KAT-7 Data

6.1 Introduction

In this chapter I investigate the use of both raw data and engineered features in machine learning, and the application of algorithms Karoo GP, kNN, and SVM in a performance comparison for the automated flagging of RFI in SKA-SA KAT-7 data. All three machine learning algorithms were employed at the Pinelands office of the Square Kilometre Array, South Africa, on a 40-core Intel server with 256 GB RAM, running the Linux operating system.

The means by which the data was prepared and the configuration of the machine learning algorithms are discussed in the following section.

6.2 The KAT-7 Data

For each integration time, analogous to an optical telescope's exposure of the sky, each radio telescope captures data which is processed at sequential stages, including preliminary flagging at the *ingester* followed by iterative flagging prior to imaging by the astronomer.

For each integration there are six basic indices, six configuration parameters, and 16 additional polarisation parameters recorded. In addition, the baseline of any given interferometer pair and the instantaneous bandwidth are recorded, information necessary to correctly process the data between any two antennae in the array. The baseline ranges

from 26 to 185 meters, the minimum and maximum possible distance between any two antennae in the KAT-7 array. The instantaneous bandwidth is a designated 256 MHz subset of the total frequency range of 1200-1950 MHz. Within the 256 MHz the array can be configured to a range of discreet, sequential channels, each of which having its own frequency range. The frequency associated with each channel is found by dividing the instantaneous bandwidth by the total number of channels where the sum of all channels equals 256 MHz.

With the particular dataset used in the body of this research, each integration is 0.5 seconds in duration and represented by a single data point, a row in a data file. Each data point is described by 45 parameters (columns), as follows.

- Six basic indices:
  amplitude
  time
  4 polarisations (2 per antenna): *HH*, *VV*, *HV* and *VH*
- Six configuration parameters:
  source (an identification for the object observed)
  antenna1
  antenna2
  azimuth
  elevation angle
  channel
- 16 polarisation parameters recorded for each integration:
  HH_amp, VV_amp, HV_amp, VH_amp
  HH_std_Mean, VV_std_Mean, HV_std_Mean, VH_std_Mean
  HH_cos, VV_cos, HV_cos, VH_cos
  HH_sin, VV_sin, HV_sin, VH_sin
- In addition, 16 stokes parameters, unique components of an electric field which propagate independently [61], are derived from the four polarisations:
  I_amp, Q_amp, U_amp, V_amp
  I_std_Mean, Q_std_Mean, U_std_Mean, V_std_Mean
  I_cos, Q_cos, U_cos, V_cos
  I_sin, Q_sin, U_sin, V_sin
- RFI flag: 1 (RFI) or 0 (source)

It is important to note that the above parameters are raw output, not features in the machine learning meaning as they do not offer deeper insight to the data itself.

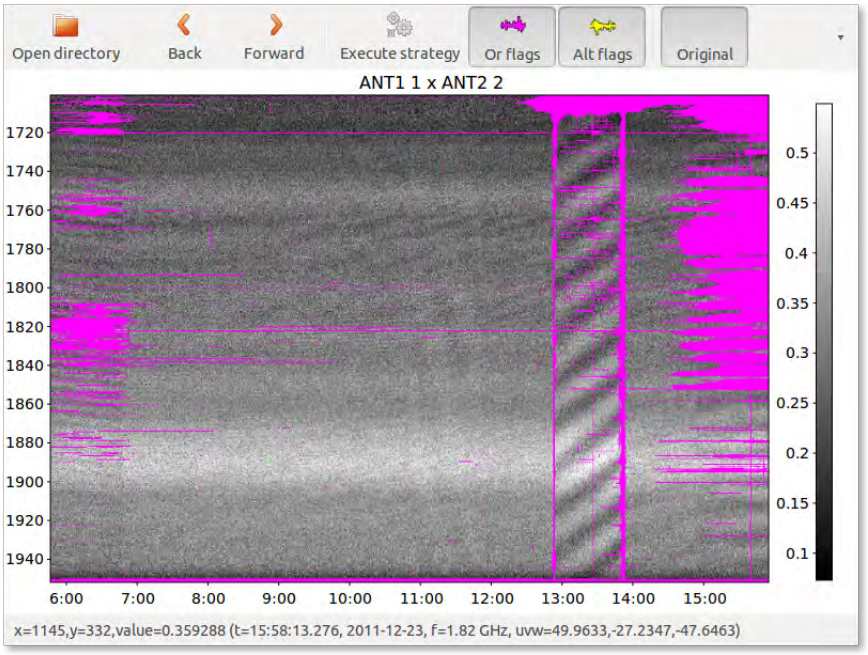


FIGURE 6.1: Software application AOFlagger flagging RFI in KAT-7 data [T. Mauch, SKA-SA], where the colour pink denotes man-made noise and the remaining grey, the desired signal.

RFI flagging of this KAT-7 observation run was conducted by means of both an automated and manual application of the software package *AOFlagger* [17] (Figure 6.1) by SKA-SA researchers Laura Richter, Lunga Mxhalisa, and Lerato Sebokolodi. Therefore, AOFlagger was used to apply the last of the parameters (above), a label of 1 for *RFI* or 0 for *non-RFI* for each data point.

The full dataset provided by the SKA-SA contains 13,000,000 data points of which 260,000 are flagged as RFI.

6.2.1 The Pixel Classifier (PC)

Working under supervisor Dr. Bruce Bassett, in the same research group as myself, UCT MSc candidate Gilad Amar developed a Python script referred to as the *Pixel Classifier* which extracts parameters from the KAT-7 data files.

The Pixel Classifier was used to generate a dataset which contains all possible parameters (as listed above) from which two subsets were generated: a dataset called *Boosted PC*, and a featureset called *Time Averaged PC*. Both are described below, followed by how they were employed by one or more machine learning algorithms.

6.2.2 Dataset: Boosted PC

This subset of the Pixel Classifier dataset was constructed by Arun Aniyan, postdoc at the Square Kilometre Array. Arun did not generate new features, rather, he used the feature selection method *boosting* as a means of determine which features are important and which demonstrate little or no correlation to the desired outcome, thus reducing the dimensionality from the original 45 parameters to just 9 which were employed by the machine learning algorithms.

The 9 data parameters retained are as follows:

U: stokes parameter
V: stokes parameter
W: stokes parameter
V_sin: stokes parameter
vh: vertical-horizontal polarisation
t: time at which the data was captured
ea: antennae elevation
az: antennae azimuth
ch: channel (frequency) to which the antennae is configured

As the original 13,000,000 rows of data contained only 260,000 rows of RFI, the 98%/2% split was far too biased in favour of non-RFI to enable a machine learning algorithm to effectively train. Therefore, a more manageable and effective 10,000 data points were prepared from a randomly selected set of 5,000 RFI (50%) and 5,000 non-RFI (50%) data points, using the following Python code:

```
data = np.loadtxt(filename, skiprows = 1, delimiter = ',', dtype = float)

# find indices where final column = 0 or 1 and record the row num
data_0_list = np.where(data[:,-1] == 0)
data_1_list = np.where(data[:,-1] == 1)

# select 5,000 unique data points for non-RFI (0) and RFI (1)
data_0 = np.random.choice(data_0_list[0], 5000, replace = False)
data_1 = np.random.choice(data_1_list[0], 5000, replace = False)

data_0_new = data[data_0]
data_1_new = data[data_1]

data_new = np.vstack((data_0_new, data_1_new))
```

With each execution of Karoo GP, the SciKit Learn Python cross-validation library (sklearn.cross_validation) splits the prepared dataset into two subsets, one for training Karoo GP and the other for testing the expressions evolved by Karoo GP. The ratio of data points contained in the training/testing subsets is 80%/20% for 8,000/2,000 respectively. These subsets are randomly generated with each execution of Karoo GP, and are therefore unique to each data run.

It is important to note that the representation of each class is not guaranteed to be identical to the total dataset, such that there may be a few more class 0 or class 1 data points in any given cross-validation sort, but not so large an offset as to adversely affect the quality of training. The number of data points for each class are presented as the Support numbers for each Precision-Recall summary.

6.2.3 Featureset: Time Averaged PC

This subset is built upon features, that is, meaningful relationships between two or more of the raw parameters extracted by the Pixel Classifier. This level of feature extraction is often associated with improved success in machine learning algorithms.

The *Time Averaged PC* featureset is built from nine features extracted from multiple, eight-second segments of the Pixel Classifier dataset, both those that contain RFI and those that do not. Each feature describes various amplitude-related properties of the signals, as described in the following:

Mean λ: Numerical average of the amplitudes in 8 sec duration segments
Skew sk: Skewness of distribution of signal voltages in the segment
Kurtosis k: Approximate nature of gaussianity of the signal segment
Max sum σ_c: Maximum value of cumulative sum of signal voltage
Slope of signal sl: The slope of linear fit of the signal segment
Percentile values @ 25% p_{25}: Upper quartile of the values
Percentile values @ 75% p_{75}: Lower quartile of the values
Percentile values p_0: difference of the values
Sum values sum: Adding all integration amplitudes into a single value

We generated 2,180,000 samples, of which 588,339 were labelled as RFI and 1,591,661 as non-RFI. As with the dataset *Boosted PC*, a total of 10,000 data points, 5,000 RFI and 5,000 non-RFI, were randomly selected from the total feature set for the training of the machine learning algorithms. Internal to Karoo GP, the SciKit Learn cross-validation module was used to generate the 80%/20% training/testing split.

As with the *Boosted PC* dataset, the representation of each class is not guaranteed to be identical to the total dataset, such that there may be a few more class 0 or class 1 data points in any given cross-validation sort. Again, the number of data points for each class are presented as the Support numbers for each Precision-Recall summary.

6.3 Data Runs with Karoo GP

A total of 12 data runs were conducted with Karoo GP, across the *Boosted PC* dataset and *Time Averaged PC* featureset combined, as follows:

- *Boosted PC* dataset: data runs A-H (8)
- *Time Averaged PC* featureset: data runs I-L (4)

Each data run required, on average, 48 hours to complete. As noted with each run, the configuration of Karoo GP was held nearly constant with adjustments made only to the maximum tree depth and the number of generations. All Karoo GP configuration settings are presented as follows:

- Kernel: Classification
  We are attempting to *classify* each data point as RFI (1) or non-RFI (0).
- Tree Type: Ramped Half/Half
  Per Section 4.4.4, this is the best means of providing a diverse code base.
- Tree Depth Max: 3-5
  Enables a potential maximum of 8, 16, or 32 Terminals respectively, in which the features (variables) are given opportunity to be raised to a higher power, providing foundation for the evolution of nonlinear functions.
- Min Node Count: 3
  None of the data runs fell into a local minimum, therefore it was not necessary to raise this value.
- Tree Pop Max: 100-200
  100 trees per generation was found to support the need to explore a large, stochastic solution space (Section 4.4.5).
- Generation Max: 10-50
  Early, experimental data runs of Karoo GP applied to the raw KAT-7 data were conducted with 10 and 20 generations, yielding low scoring hypotheses. It was

discovered that convergence on higher scoring hypotheses began at 30 generations, with minimal evolutionary adaptation after 50 generations, meaning an improved hypothesis was not likely to be missed for termination at this stage. Therefore, no data run described herein was conducted with fewer than 30 or greater than 50 generations.

- Genetic Operators:
  reproduction: 10%
  point mutation: 10%
  branch mutation: 20%
  crossover: 60%

  These values are the result of a combination of early, experimental runs (Section 6.2.1) and the literature review (Section 4.7).

- Tournament size: 10
  Ten trees are 10% of a population of 100 trees (Section 4.5.1).

- CPU Cores: 40 cores
  In experimental data runs prior to engaging the A-L data, Karoo GP saw nearly linear performance improvement from 12 to 28 cores, and a tapering improvement to 40. This was established using the Timer function built-in to Karoo GP.

- Floating Point Precision: 4
  The SciKit Learn Precision-Recall module reports only two decimal places, therefore four were determined to be ample.

6.3.1 Data Runs A-H

A total of 8 data runs (A-H) were conducted with the *Boosted PC* dataset, with the following configuration parameters for Karoo GP:

kernel: *c*lassification
tree_type: *r*amped half/half
tree_depth_max: 5, 4, or 3 (see below)
min_node_count: 3
tree_pop_max: 100
generation_max: 30
tourn_size: 10
cores: 40
precision: 4

Data Run A Results

With Karoo GP set to a tree depth of 5:

Tree 1: $az^2*ea/ch-az*ea/t+az*t-ch*ea/vh+ch*t^3*vh+2*ch+ch*vh/t-ea+3*t$

Tree 4: $az^2*ea/ch-az*ea/t+az*t-ch*ea/yx+ch*t^3*yx+2*ch+ch*yx/t+2*t$

Tree 6: $az^2*ea/ch+az+ch*t^3*ea-ch*ea/vh+2*ch+ch*vh/t-ea^2/t-ea+t+vh$

Tree 9: $az^2 * ea/ch + az * t + ch * t^3 * ea - ch * ea/vh + 2 * ch + ch * vh/t - ea^2/t$
$-ea + 2 * t + vh$

P-R Classification Report for Tree 4, Data Run A

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.75	0.97	0.85	978
1 RFI	0.97	0.69	0.81	1022
avg / total	0.86	0.83	0.83	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	949	29
1 RFI	173	849

TABLE 6.1: Precision-Recall Classification report for Tree 4, Data Run A, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 949 true-positive class 0 and 849 true-positive class 1.

Data Run B evolved two unique hypotheses, both of which scored lower in the Precision-Recall Classification tests than those in Data Run A.

Data Run C was terminated with an ill-timed reboot of the server.

Data Runs D, E, and F were configured for a maximum tree depth of 4, but yielding no improvement over data runs A and B.

Data Run G was terminated with a server crash.

Data Run H Results

With Karoo GP set to a tree depth of 3:

Tree 1: $-ea + t * vh + t + ea/ch$

Tree 2: $-ea + t * vh + t + ea/ch$

Tree 3: $-ea + t * vh + t + ea/ch$

...

Tree 97: $-ea + t * vh + t + ea/ch$

Tree 98: $-ea + t * vh + t + ea/ch$

Tree 99: $-ea + t * vh + t + ea/ch$

P-R Classification Report for Tree 99, Data Run H

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.78	0.97	0.86	1018
1 RFI	0.96	0.71	0.82	982
avg / total	0.87	0.84	0.84	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	989	29
1 RFI	284	698

TABLE 6.2: Precision-Recall Classification report for Tree 99, Data Run H, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 989 true-positive class 0 and 698 true-positive class 1.

6.3.2 Discussion

With this initial set of runs against the dataset *Boosted PC*, the intent is to simply look for goodness of fit, structure, and repeating patterns in what has evolved.

While we started with nine parameters, in run A the four Stokes parameters dropped, leaving az, ea, ch, t, and vh.

In the evolved expressions, we see structure forming, combinations of parameters which repeat. For example, in run A (Table 6.1) there are four identical sub-functions contained in each multivariate expression: $az^2 * ea/ch$, $ch * t^3$, $ch * ea$, and $2 * ch$. While we will not attempt to trace these correlations back to the physical world, these elements are clearly foundational to the identical performance across all trees evolved in this run.

In general, the expressions evolved in data runs A and B intuitively feel too complex, and appear to be overfitting despite their weak performance in the Precision-Recall scores.

When we reduce the tree depth from 5 to 3, thereby reducing the maximum number of Terminals from 32 to 8, we alter the potential outcome with reduced opportunity for nonlinear functions to evolve. We need only compare the hypotheses evolved in runs A and B to those of H as evidence of this reduction in dimensionality.

It is important to note that in data run H (Table 6.2) we see total convergence to a single hypothesis, with only four of the original nine parameters *ea*, *t*, *vh*, and *ch* retained. Yet this run performed better overall, with an average F1 score of 84 versus 83 in the prior runs.

We then look to the Confusion Matrix, paired with the Precision-Recall test, as a means to readily identify the strengths and weaknesses of this particular evolutionary run. With Tree 99, data run H, we see that this hypothesis correctly identified 989 of 1018 non-RFI data points (class 0), with just 29 false-positives. However, it did quite poorly in identifying RFI data points (class 1), with just 698 out of 982 and 284 false-positives.

6.3.3 Data Runs I-L

A total of four data runs (I-L) were conducted with the *Time Averaged PC* dataset, with the following configuration parameters for Karoo GP:

kernel: *c*lassification
tree_type: *r*amped half/half
tree_depth_max: 5 or 4 (see below)
min_node_count: 3
tree_pop_max: 100
generation_max: 50
tourn_size: 10
cores: 40
precision: 4

Data Run I Results

With Karoo GP set to a tree depth of 5:

Tree 1: $\sigma_c^2 * \sigma_v - \sigma_c - k * \lambda^2/p_{25} + \lambda^2 * p_{25} - \lambda + p_0 + p_{25}/\sigma_v + p_{75} + sk/\sigma_v - \sigma_v/\lambda$
Tree 3: $\sigma_c^2 * \sigma_v - \sigma_c - \sigma_c/\lambda - k * \lambda^2/p_{25} + \lambda^2 * p_{75} + p_0 + p_{25}/\sigma_v + sk/\sigma_v$
Tree 9: $-\sigma_c - \sigma_c/\lambda - k * \lambda^2/p_{25} + p_0 + p_{25} * p_{75}^2 + p_{25}/\sigma_v + sk/\sigma_v + \sigma_v^3$
...
Tree 46: $\sigma_c^2 * \sigma_v - \sigma_c - \sigma_c/\lambda - k * \lambda^2/p_{25} + \lambda * p_{25}^2 + p_0 + p_{25}/\sigma_v + sk/\sigma_v$
Tree 67: $\sigma_c^2 * \sigma_v - \sigma_c - \sigma_c/\lambda - k * \lambda^2/p_{25} + \lambda^2 * p_{75} + p_0 + p_{25}/\sigma_v + sk/\sigma_v$

P-R Classification Report for Tree 46, Data Run I

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.62	0.99	0.76	965
1 RFI	0.98	0.43	0.60	1035
avg / total	0.81	0.70	0.68	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	957	8
1 RFI	591	444

TABLE 6.3: Precision-Recall Classification report for Tree 46, Data Run I, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 957 true-positive class 0 and 444 true-positive class 1.

P-R Classification Report for Tree 67, Data Run I

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.62	0.99	0.76	965
1 RFI	0.98	0.43	0.60	1035
avg / total	0.81	0.70	0.68	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	957	8
1 RFI	591	444

TABLE 6.4: Precision-Recall Classification report for Tree 67, Data Run I, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 957 true-positive class 0 and 444 true-positive class 1.

Data Run J Results

With Karoo GP set to a tree depth of 4:

Tree 1: $\sigma_c^3/p_{25} + \sigma_c + 4 * k + k/p_{25} + \lambda - k/\sigma_c^2$

Tree 50: $\sigma_c^3/p_{25} + \sigma_c + 4 * k + k/p_{25} + \lambda - k/(\sigma_c * \sigma_v)$

Tree 92: $\sigma_c^3/p_{25} + \sigma_c + 4 * k + k/p_{25} + \lambda - k/\sigma_c^2$

Tree 99: $\sigma_c^2 * \sigma_v/p_{25} + \sigma_c + 4 * k + k/p_{25} + \lambda - k/\sigma_c^2$

P-R Classification Report for Tree 92, Data Run J

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.83	0.98	0.90	989
1 RFI	0.98	0.80	0.88	1011
avg / total	0.90	0.89	0.89	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	969	20
1 RFI	201	810

TABLE 6.5: Precision-Recall Classification report for Tree 92, Data Run J, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 969 true-positive class 0 and 810 true-positive class 1.

P-R Classification Report for Tree 99, Data Run J

	Precision	**Recall**	**F1 score**	**Support**
0 non-RFI	0.83	0.98	0.90	989
1 RFI	0.98	0.80	0.88	1011
avg / total	0.90	0.89	0.89	2000

Confusion matrix

	non-RFI	RFI
0 non-RFI	969	20
1 RFI	201	810

TABLE 6.6: Precision-Recall Classification report for Tree 92, Data Run J, where *Support* is the total number of data points in each class, with the left-to-right diagonal of the Confusion Matrix yielding 969 true-positive class 0 and 810 true-positive class 1.

6.3.4 Discussion

In this set of runs against the dataset *Time Averaged PC*, we are working with features, not just raw data points. Therefore, we anticipated a higher quality outcome. However, when we start with a tree depth of 5 (Tables 6.3, 6.4 for run I) the overall performance, as ranked by the F1 score, is lower than that of the initial run A with the same tree depth. When we reduce to tree depth 4 (Tables 6.5, 6.6 for run J) GP performs better than all prior runs.

Without several more runs at each tree depth, it is difficult to state why we see a decrease in performance on the *Time Averaged PC* featureset for the same tree depth as the prior

Boosted PC dataset. GP may have stalled in a local minimum, or initiated run I with poor random seeds from which it did not recover.

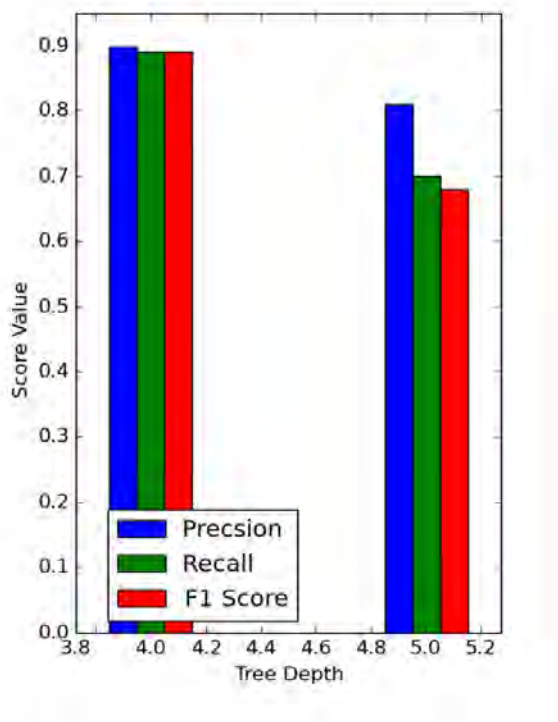


FIGURE 6.2: A comparison of the average Precision, Recall, and F1 scores for trees J and I with depth 4 and 5 respectively. In this case, the lower depth 4 performed better than the higher dimensional depth 5.

As with the depth 5 runs against the *Boosted PC* dataset, we see expressions that are intuitively too complex for the quantity of variables present with the depth 5 run against the *Time Averaged PC* featureset. As visualised in Figure 6.2, reducing the tree depth, and thereby the order of the multivariate expression, improves performance against this featureset, from an average F1 score of 68% in run I to 89% in run J, five points higher than with any tree in the prior runs.

We again look to the Confusion Matrix. With Tree 99, data run J (Tables 6.5, 6.6), we see that this hypothesis correctly identified 969 of 989 non-RFI data points (class 0), with just 20 false-positives. However, it performed sub-optimally in identifying RFI data points (class 1), with just 810 positives and 201 false-positives. This is an area for improvement.

Of the original 9 parameters, σ_c, p_{25}, k, σ_v, and λ are retained in run J. σ_c is repeatedly squared or cubed, while λ is sometimes squared. As with the runs against the *Boosted PC* dataset, we will not make claims as to any mechanical coupling in the real world, but when we see repeating structure across multiple trees in multiple runs, we begin to

recognise which parameters will likely be retained by the GP feature selection process, and to what degree (n^1, n^2, or n^3) they will be represented.

It is important to note that in runs I and J we have multiple hypotheses, per each run, that provide identical F1 scores, 68% in I and 89% in J respectively. This is an example of the evolutionary process taking two or more paths to arrive to the same solution. This provides some level of confidence, that each run converged on an optimal solution for the given data and GP configurations.

Data runs K and L will not be discussed as they did not provide improvement over runs I and J.

6.4 Data Runs with kNN and SVM

In the early stage of this research, Gilad Amar, Arun Aniyan, and I collaboratively developed a pipeline which explores the application of multiple machine learning algorithms to a single dataset.

This pipeline employs a *grid-search*, a means of systematically exploring candidate hypotheses generated by a machine learning algorithm where one or more parameters are incrementally adjusted with each run. Typically, the user defines the minimum and maximum input values, and the incremental steps in between [62].

The pipeline optimises the given parameter(s) for the given machine learning algorithm (in this case, a binary classifier), for accuracy, precision or recall as specified by the user. The grid-search tuning parameters for all included classifiers are user-specified in the main pipeline code, and auto-recognise when the classifier is optimised. Internal to the pipeline, the training/testing split is generated with each run. With each run, the full grid-search is applied. The hypothesis with the best set of scores is chosen for each specified classifier.

There are several means by which the distance between any two data points may be measured. Manhattan and Minkowski distance metrics are frequently employed. The grid search determined the best, real-value vector space metric to be the *Minkowski* distance [63], defined as:

$$p = [p_1, p_2, ..., p_n] \tag{6.1}$$

$$q = [q_1, q_2, ..., q_n] \tag{6.2}$$

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{6.3}$$

Where p is the first parameter, q the second, and d the distance between them as calculated by the square root of the sum of the square of the difference of each sequential, paired value.

6.4.1 Pipeline Configuration

The pipeline was configured to employ the k-Nearest Neighbour (kNN) and Linear Support Vector Machine (SVM) algorithms made available from SciKit Learn[1], against the *Time Averaged PC* featureset.

kNN Pipeline Configuration

For kNN, the pipeline was given the following parameter search space:

Runs: 10

Algorithms: KDTree, BallTree

k: 3 to 15, incrementing by odd numbers only

Dataset: *Time Averaged PC*

Kernels KDTree and BallTree were both employed to compare their relative performance. where KDTree improves upon a brute force approach to data point comparison, and BallTree improves upon KDTree when applied to higher dimensional data.

The nearest-neighbour parameter k was chosen in relation to the dimensionality of the *Time Averaged PC* featureset, where k was set to a minimum of 3 and maximum of 15 for the given nine features, as the feature histograms showed no clear indication of separation in lower dimensions, and lower values for k are subject to noise [63]. As this is a binary classification problem, we increment by odd numbers so as to avoid a potential tie in selecting the class for a test case.

kNN was optimised for *precision* (instead of accuracy or recall) and engaged in a minimum of 10 pipeline runs, each with a modified grid of search parameters. This means there were 10 unique training/testing datasets created, each applied against all possible combinations of kernels, k-neighbours, and leaf sizes. This resulted in *10 runs * 2 algorithms * 7 k-Neighbours* for a total of 140 unique runs.

kNN converged on 5 k-Neighbours, a leaf size of 45, and the algorithm $KDTree$.

SVM Pipeline Configuration

[1] www.scikit-learn.org

For Linear SVM, the pipeline was given the following parameter search space:

Runs: 10

Kernels: Linear

C Support: [1, 11, 21, 31, ... 101], in increments of 10

Loss function: L1, L2

Dataset: *Time Averaged PC*

In early test runs of SVM (Nov 2014 - Feb 2015) it was found that the kernels *polynomial* and *rbf* did not converge upon a single cross-validation dataset, not even when allowed to run for a week or more. Therefore, only the Linear kernel was engaged in the pipeline.

C support determines the margin of the hyperplanes, where large values of C invoke a smaller margin, and small values a larger margin. This parameter works to reduce the number of misclassifications.

As with kNN, SVM was optimised for *precision* and engaged in a minimum of 10 pipeline runs, each with a modified grid of search parameters. This means there were 10 unique training/testing datasets created, each applied against all possible combinations of available kernels, support, and loss functions. In the case of SVM, this resulted in *10 runs * 1 kernel * 11 support values * 2 loss functions* for a total of 220 unique runs.

SVM converged on a support value of 1 and Loss Function of $L2$.

6.4.2 kNN and SVM Results and Discussion

The best, overall pipeline results:

P-R Classification Report for kNN, Linear SVM

	Precision	**Recall**	**F1 score**
kNN avg	0.86	0.73	0.78
SVM avg	0.96	0.32	0.48

TABLE 6.7: Precision-Recall Classification report for the kNN and SVN runs on the *Time Averaged PC* featureset. Precision, Recall, and F1 scores are given. The *Support* values were not calculated, as the quantity of non-RFI and RFI in the test dataset were not provided by these classifiers.

As with the final four runs of Karoo GP, kNN and SVM were employed to engage the *Time Averaged PC* featureset. As shown in Table 6.7, kNN performed better than SVM for the F1 score for 48% versus 78%. However, SVM provided a higher Precision score than kNN at 96% versus 86% and kNN outperformed SVM for Recall 73% to 32%.

As kNN and SVM were each configured to search a dynamic parameter space across 140 and 220 runs respectively, it is safe to state that these results demonstrate the best that kNN and SVM are able to offer for this given featureset and the parameters made available.

6.5 Conclusion and Closing Discussion

Algorithm	**data type**	**Precision**	**Recall**	**F1 score**
Best of GP A-H	raw	0.87	0.84	0.84
Best of GP I-L	features	0.90	0.89	0.89
Best of kNN 140	features	0.86	0.73	0.78
Best of SVM 220	features	0.96	0.32	0.48

TABLE 6.8: Summary of all runs against the SKA-SA KAT-7 data in which Karoo GP, kNN, and SVM hypotheses are evaluated for their Precision, Recall, and F1 scores

As shown in Table 6.8, there were five groupings of data runs. Recall that run H employed the *Boosted PC* dataset while runs I, J, SVM and kNN employed the *Time Averaged PC* featureset, where both were generated from the same SKA-SA KAT-7 observation.

For the metric Precision, SVM provided the highest score of 95%. For the metric Recall, Karoo GP provided the highest score of 80%. For the metric F1 score, Karoo provided the highest score of 89%.

With Karoo GP and its four runs (I-L) against the *Time Averaged PC* featureset, the best P/R/F1 scores were 0.90/0.89/0.89 accordingly, a noteworthy but not significant improvement over Karoo GP against the raw dataset and a P/R/F1 of 0.87/0.84/0.84.

While this is by no means an exhaustive study of feature engineering nor machine learning applied to the KAT-7 data, it does lightly explore the role of features and engage a comparative performance of three machine learning algorithms Genetic Programming, k-Nearest Neighbour, and Support Vector Machines.

As the SKA-SA team in Pinelands, South Africa employs a dedicated team of researchers for the task of applying machine learning to automated RFI mitigation, progress will surely continue to be made. But "success" does not come easy with such complex datasets and an incredibly challenging problem at hand. Optimisation will continue even after the MeerKAT array is fully operational, in an effort to improve the quality of the data for the benefit of the radio astronomers.

At the time of this research, the SKA-SA was just beginning to investigate the application of machine learning. As such, Karoo GP was one of the first algorithms employed, alongside Artificial Neural Networks and the ongoing exploration of statistical modelling.

The KAT-7 dataset was the first real-world data employed by Karoo GP, a platform for evolutionary computation developed in the course of this research. Karoo GP readily scaled from its built-in toy models to a featureset of 10,000 rows and nine columns, running in parallel across 40 compute cores, for a dozen, 48 hour runs.

The hypotheses evolved by Karoo GP matched the anticipated behaviour of Genetic Programming. An increase in complexity in the multivariate expressions was observed as an increase in the upper boundary of the tree size, and decreased complexity with a lower boundary accordingly. In the body of this research, the classical machine learning cases of local minima, underfitting and overfitting were demonstrated. And an overall improvement in fitness scores were associated with the movement from a raw dataset to a featureset.

The results of Karoo GP applied to this single KAT-7 dataset were promising, opening possible further exploration in which additional, more strongly correlated features would be explored.

Appendix A

Karoo GP User Guide

This appendix provides the full body of the Karoo GP User Guide.

Karoo GP, the User Guide, additional data management tools and sample data are freely available via Github.

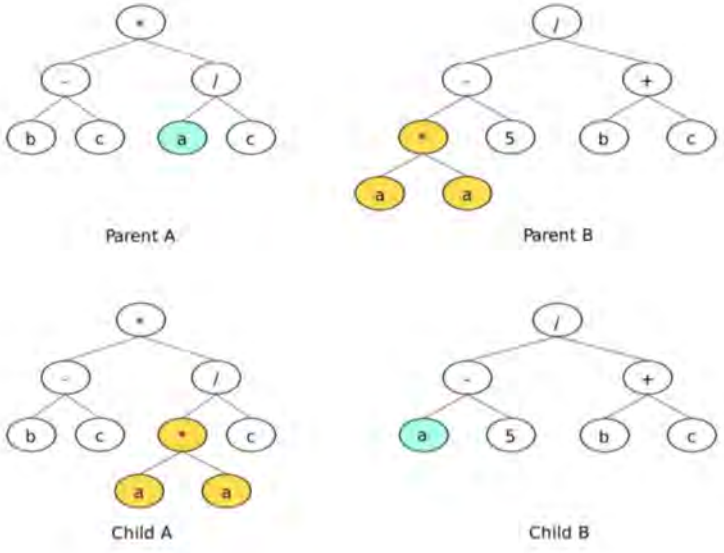
Karoo GP User Guide

Genetic Programming in Python

ver. 20160824

by Kai Staats

kstaats.github.io/karoo_gp/



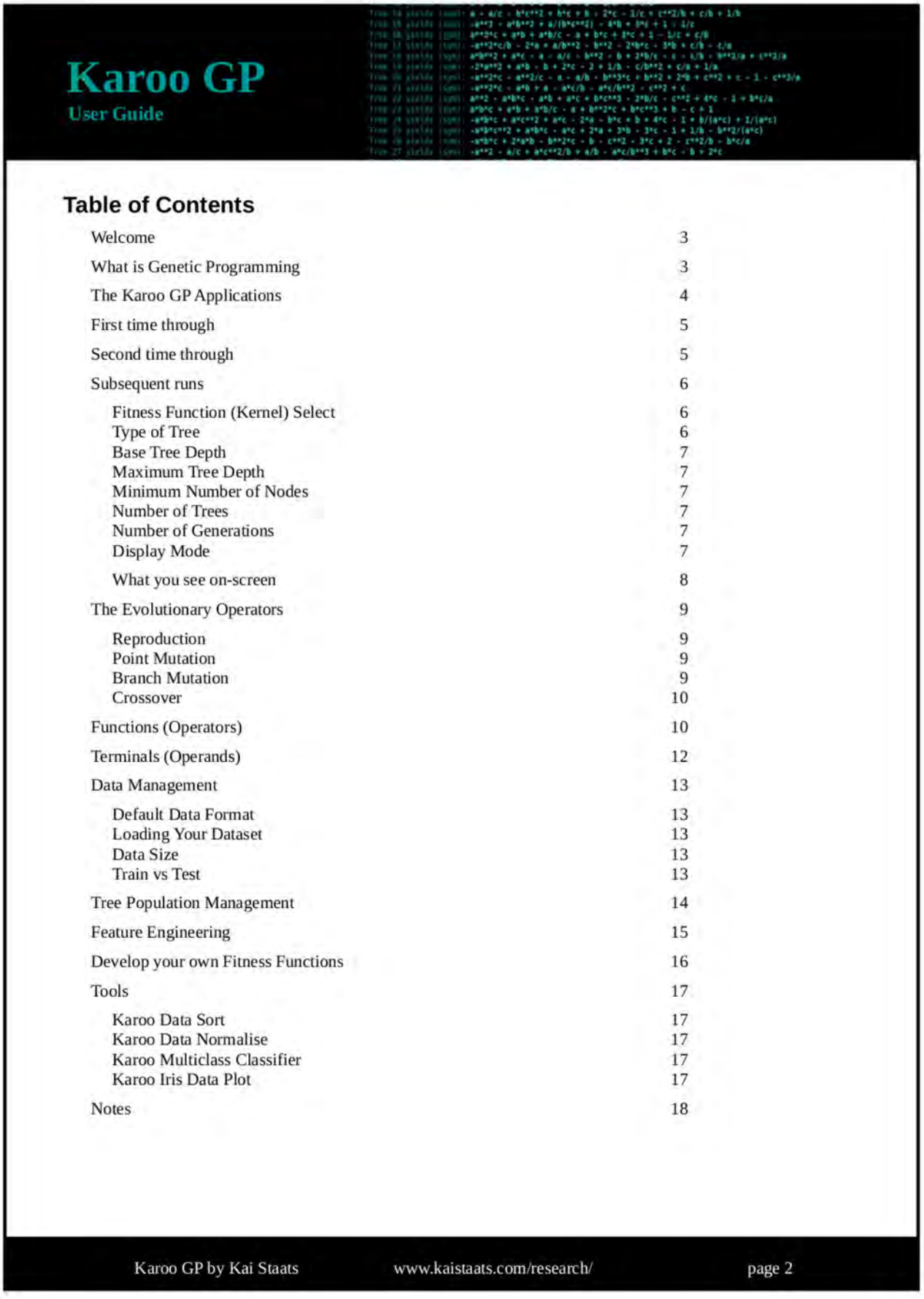


Table of Contents

Welcome to Karoo GP

Karoo GP is an evolutionary algorithm, a *genetic programming* application suite which provides both symbolic regression and classification analysis. Written in the programming language Python, Karoo GP owes its foundation to the "Field Guide to Genetic Programming" by Poli, Langdon, McPhee, and Koza.

Karoo GP is a scalable platform with multicore support, designed to readily work with realworld data. *No programming required.* As a teaching tool, it enables instructors to share step-by-step how an evolutionary algorithm arrives to its solution. As a hands-on learning tool, Karoo GP supports rapid, repeatable experimentation.

Karoo GP includes a Desktop application with an intuitive user interface, a fully scriptable Server application with user defined default parameters and command-line arguments; a stand-alone Python script which generates randomly sorted subsets of larger datasets and another which normalises datasets; and a toy model which shows the inner workings of multiclass classification.

This guide is designed to give you a crash course in tree-based Genetic Programming and a light introduction to preparing datasets for machine learning. Even if you are an expert in machine learning, you are encouraged to make time to read this document in order to anticipate all that Karoo GP has to offer.

Requirements:

- Python 2.7.6 [GCC 4.8.2] with libraries: NumPy 1.9.2, SymPy 0.7.6.1, pprocess 0.5.1, sklearn 0.16.1, matplotlib, csv, os, sys, argparse, and time

Tested systems:

- Ubuntu Linux 14.04 Desktop on Intel i5/i7 (4 core): no known issues
- Ubuntu Linux 14.04.3 LTS on Intel Xeon E5-2650 (40 core): no known issues
- Mac OS X on various hardware: *pprocess does not report the quantity of CPU cores. To resolve, manually set self.core_count to the number of cores on your system, at or near line 120 in karoo_gp_base_class.py*

What is Genetic Programming?

Genetic Programming (GP) is a type of evolutionary algorithm, a subset of machine learning. Inspired by biological evolution, GP is a computer program that uses random mutation, crossover, a fitness function, and multiple generations of evolution to resolve a user-defined task. At the foundation level, genetic programming discovers relationships between dataset features—correlations between measurements of the real world.

Long before the advent of gene sequencing or machine learning, taxonomists conducted precise measurements of physical plants and animals in order to differentiate species. In 1936, biologist R.A. Fisher worked to differentiate species of Iris flowers using data generated by Edgar Anderson. In his paper "The use of multiple measurements in taxonomic problems", Fisher states "When two or more populations have been measured in several characters, xl, ..., x8, special interest attaches to certain linear functions of the measurements by which the populations are best discriminated." Using Genetic Programming, we readily discover the mathematical functions that cleanly separate the 3 Iris species using just 50 data points each. This Iris multivariate dataset is a simple yet classic, must-solve problem for all Machine Learning developers of classification algorithms, and is included with Karoo GP as the default Classification dataset.

In general, genetic programming can be used to discover a relationship between features in data (symbolic regression) or to group data into categories (classification). Both of these employ a training (learning) and test (validation) phase, after which the resulting function (mathematical expression) may be employed in a live data stream for realtime regression or classification analysis, accordingly.

The Karoo GP Applications

Desktop

The Karoo GP Desktop application presents a functional interface for configuring, launching, and monitoring your Karoo GP runs. There are 5 choices for how you monitor the evolutionary process, from the *silent* mode in which you see only the outcome of each generation to a detailed, step-by-step debug mode in which every mutation and crossover process is presented to you, as both a validation and learning tool.

To launch the Desktop application from the shell:

```
$ python karoo_gp_main.py
```

If you include the path to an external dataset, it will load at launch:

```
$ python karoo_gp_main.py /[path]/[to_your]/[filename].csv
```

While this can be run on a remote server, you may find that once you get the hang of using Karoo, and are in production mode, karoo_gp_server.py provides both a powerful, scripted and command-line platform.

Server

Karoo GP Server is designed to support repeated, scripted, and command-line executed genetic programming runs. You can manually configure a routine to move through a series of tree depths or population sizes, or apply an external genetic algorithm to analyze the leading expressions from each generation and modify the configuration of each subsequent generation or full run accordingly, to help guide GP toward the optimal solution.

```
$ python karoo_gp_server.py
```

Without arguments, Karoo GP defaults to the datasets located in karoo_gp/files/ as determined by the fitness function (kernel) selected, eg: a Classification kernel loads data_CLASSIFY.csv. If you include *only* a path to your own dataset:

```
$ python karoo_gp_server.py /[path]/[to_your]/[filename].csv
```

If, however, you include any other arguments, then you must *also* include a flag to load your own dataset:

```
$ python karoo_gp_main.py -ker c -typ r -bas 4 -fil /[path]/[to_your]/[filename].csv
```

You can set your preferred default configuration parameters in the file karoo_gp_server.py such that without any command-line arguments, Karoo GP Server will repeat a run for comparison to priors. Command-line arguments then override these default settings, as follows:

argument	**options**	**description**
-ker	[r,c,m]	kernel: (r)egression, (c)lassification, or (m)atching
-typ	[f,g,r]	tree type: (f)ull, (g)row, or (r)amped half/half
-bas	[3...10]	maximum tree depth for the initial population
-max	[3...10]	maximum tree depth for the entire run
-min	[3...100]	minimum number of nodes
-pop	[10...1000]	maximum population
-gen	[1...100]	number of generations
-fil	/[path]/[to_your]/[filename].csv	load an external dataset

Karoo GP

User Guide

First time through

The Karoo GP user interface will prompt you for a series of inputs. Your first time through, enter Play mode in order to understand the functions of Karoo GP and to learn to visualise a GP tree, as follows:

1. Select Play mode.
2. Select a Full tree.
3. Select a depth of 1.

    Depth 1: 1 operator, 2 variables, eg: a / c

    Depth 2: 3 operators, 4 variables, eg: a + b – 2 * c

    Depth 3: 7 operators, 8 variables, eg: c + b * c^3 / a + c / a

Take a look the GP tree represented on-screen. There are 3 nodes composed of 2 terminals (variables) and 1 function (operator). Select Play mode a few more times, playing with larger trees and both Full and Grow methods. You will find that the branches of Full trees always reach the bottom, where the branches of Grow trees may terminate early, with a terminal always the final node. With some practice, you can read the tree as printed on screen. Future versions of Karoo GP will include an option to generate an image (similar to the cover page).

Second time through

Run Karoo GP again, but this time simply press ENTER at each of the queries. Sit back and watch as Karoo GP works to resolve a simple relationship between 3 variables which represent 3 columns of data: a + b + c = s.

We know the answer, but Karoo GP must discover a solution through an evolutionary process. As GP is based on random number generation, there is a chance that in the default 10 generations it will not work, or it might find a relationship between the columns of data other than that which was expected.

By configuring a run for deeper trees, or by increasing the minimum number of nodes required, you can encourage the evolutionary process to discover more complex solutions.

When done, you will be presented with *pause*. Type ? and ENTER to review the menu:

1. Type *l* and ENTER to view all trees which provide the correct solution.
2. Type *p*, ENTER, and then the unique number of any tree in the list, to print that tree to screen.
3. The solution will be listed in both its *raw* and Sympyfied format. Remember that multiple trees may carry the same or more than one solution to the same problem.
4. You may adjust the (b)alance of the operators, (cont)inue the evolutionary process, (test) the trees, or (q)uit.

Karoo GP
User Guide

Subsequent runs

When you are ready to dive in a bit further, take a few minutes to learn about each of the following:

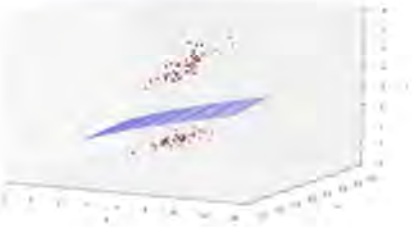
1 – Fitness Function (Kernel) Select

When prompted with a choice of (c)lassify, (r)egression, (m)atching, or (p)lay:

Regression is a minimisation function, meaning it seeks a fitness score with the *lowest* number. By default, this kernel will attempt to solve Kepler's 3rd Law of Planetary Motion. As the correct solution is t^2 / r^3, the challenge is for GP to not fall to a local minimum of *t/t or r/r*. You can experiment with both deeper trees and increasing the minimum number of nodes. Or, let Karoo GP run for 20, 30, ... 50 or more generations and random mutation might just discover the correct solution.

To learn more, visit: www.physicsclassroom.com/class/circles/Lesson-4/Kepler-s-Three-Laws

Classification is a maximisation function, meaning it seeks a fitness score with the *highest* number. By default, this kernel will attempt to solve the Iris flower problem. This is a machine learning classic built upon data generated before the dawn of DNA classification, when botanists used microscopes and calipers to measure the features of plants and animals. While Kepler's Law has one solution, the Iris problem has many. If you plot one of the GP generated functions over a scatter plot of the data (see tools/) you should find a clean separation of 2 or 3 species.



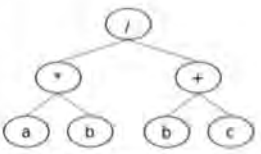
To learn more, visit: archive.ics.uci.edu/ml/datasets/Iris

Karoo GP supports multiclass classification from 2 to *n* classes, with automated scaling. See *Additional Scripts* (below) to learn more, and to play with the included karoo_multiclassifier.py found in the tools/ directory.

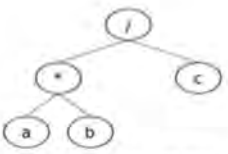
Match will attempt to discover a relationship between variables which produces an exact match, as with $a + b + c = s$, as noted in *Second Time Through* (above). This is the simplest of the fitness functions to experiment with on your own, but has limited realworld application. Try increasing tree depth and the minimum number of nodes. See *Using Your Own Data* at the bottom of this guide to learn how to replace the default data with your own.

2 – Type of Tree

All Full tree branches reach the maximum depth. These trees are less likely to solve problems as they are not as flexible in their solution space. For example, if the desired solution is a + b + c (5 elements) but a Full tree is confined to 15 elements, some will be forced to cancel each other in order to arrive to the desired solution. However, Full trees do contribute higher order expressions.



Full method GP tree



Grow method GP tree

Grow trees have unbalanced branches, meaning some branches reach the maximum depth while others do not. Grow trees are more likely to find simpler solutions, but may not discover improved solutions in a higher order space.

Originated by John Koza, *Ramped Half/Half* initialises the first population with a 50/50 split of Full/Grow methods, and a spread of depths from min to max. The Grow method is then applied for each subsequent population. This is the most popular method as it injects a higher degree of diversity into the initial population.

Karoo GP

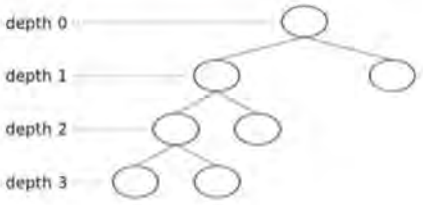
User Guide

3 – Base Tree Depth

This is the tree depth for the *initial* population. By default, this is also the Maximum Tree Depth (below), unless otherwise user specified. If the base and maximum tree depths are equal, this will inhibit bloat, but also restrict potential, more complex solutions.

The depth of a tree is illustrated to the right, and relates to the maximum possible number of nodes, as follows:

nodes = 2^(depth + 1) – 1



4 – Maximum Tree Depth

Karoo is unique from traditional tree-based GP in that it incorporates a user defined maximum tree depth, thereby restricting program bloat. However, deeper trees present opportunity for more complex solutions, and they enable the inclusion of a greater number of features (columns in your .csv). As depth 3 enables up to 15 nodes, and depth 5 enables up to 63 nodes, quite a bit of growth is possible with trees just 2-3 depths greater. Or, you can set the maximum to 10 (2047 possible nodes) and learn if a larger solution is in fact more able to resolve your data.

5 – Minimum Number of Nodes

If the Maximum Tree Depth is the *ceiling*, then this is the *floor*. The minimum number of nodes (both operators and terminals) defines the simplest expression allowed. For example, the correct solution to Kepler's 3rd Law of Planetary Motion is t * t / r * r * r which has 9 elements (nodes). In Karoo GP, the gene pool from which the tournament selection operates is built only from those trees which meet the minimum number of nodes criterion. This is very useful in solving the default problem for the Regression kernel, as GP tends to the simpler t / t.

NOTE: *If you set the minimum number of nodes too high, you may invoke elitism in the population, killing off simpler solutions or even the entire population.*

6 – Number of Trees

100 trees per population is a good place to start. However, larger populations offer a greater diversity of possible solutions. Feel free to experiment with larger populations.

7 – Number of Generations

For simpler problems with less than a dozen features, 10 generations will often give you a sense of whether or not Karoo GP is on the right track. With more complex problems, 20, 30, … 50 or more generations may be required for the random process of evolution to generate an improved tree which provides the desired solution.

You may (cont)inue with subsequent generations when all generations have run their course (see bottom).

8 – Display Mode

The display modes (i)nteractive, (m)inimal, (g)eneration, and (s)ilent are the means by which you interact with and monitor the evolutionary process of Karoo GP. They do not affect the outcome of the process itself.

(i)nteractive - *pause* with each step of the Karoo GP process.

(m)inimal - runs through the entire evolution, start to finish without *pause*. However, it does display each tree as it is evaluated, which is quite entertaining (if you are into that sort of thing).

(g)eneration – *pause* with the end of each generation, allowing you to make adjustments once per generation.

(s)ilent - designed to run remotely on a server with limited overhead. This mode presents the results of each generation just to let you know it is still alive 'n cranking through data while you are engaged in data reductions, Facebook, or a Star Wars role playing game.

There are 2 additional modes available to you at any time, as listed with the *pause* menu:

(d)e(b)ug - for those of you who want to watch the evolutionary process at the intimate level, this is a fully transparent window to the inner workings of tree-by-tree Mutation and Crossover. In addition, you will watch the criteria of the engaged fitness function, the tournament selection, and the population of the gene pool from which the next generation will be selected.

(t)imer - allows you to monitor the performance of your current GP run against the number of CPU cores currently engaged. At each *pause*, you may select (c), enter the number of CPU cores, and then ENTER again to continue. Karoo GP is *multi-core* engaged, meaning it spawns processes across physical CPU cores (not multiple threads on a single core). At the time of this edit, Karoo GP is *not* GPU enabled.

You can switch to any mode at any *pause*. However, once you are in *minimal* or *silent* modes, there are no longer opportunities to modify the evolutionary parameters until all generations are complete. However, you may continue the evolutionary process and Karoo GP will pick up where you left off. Simply enter (cont) followed by the number of subsequent generations, and off you go. If new parameters are desired, remember to set all configurations before you continue.

What You See On-Screen

The *pause* menu

Be certain to play around with all the options available to you in *pause*. "?" or "help" followed by ENTER will display all available options. You can change the level of interaction, minimum number of nodes, number of CPU cores; print and test trees, and more.

The most fit trees

At the end of each generation of population evolution, a list of bold numbers is presented on-screen. These are the trees GP has found to offer the best overall fitness scores in the latest generation. At each *pause*, you may (l)ist, (p)rint, and/or *test* (various tests may be applied) any given tree, and again at the end of any given run. If you enter (pop)ulation the full, current population will be displayed on screen.

The leading Trees and their associated expressions are:

1 : 2*a - b + 2*c

2 : a + 2*b + 1

5 : a + 2*b + 1

6 : a + b + c

10 : a + 2*b + 1

12 : a + 2*b + 1

21 : 2*a - b + 2*c

94 : a + 2*b + 1

Karoo GP

User Guide

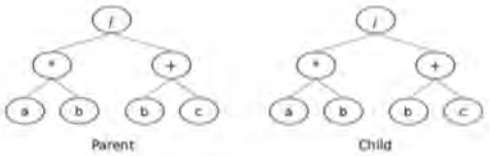
The Evolutionary Operators

There are 4 evolutionary operators applied in Karoo GP. The foundation for these evolutionary operators was derived from the "Field Guide to Genetic Programming" * by Riccardo Poli, William Langdon, and Nicholas McPhee, with contributions by John Koza. Using the (b)alance option in the *pause* menu, you can set the ratio of the operators, such that combined they equal 100%.

In the following, *tournament selection* refers to the random selection of a number of trees (default 10) whose fitness scores are compared. The tree with the highest score is moved to the next generation through an evolutionary operation. We use tournament selection, not a top-to-bottom evaluation of the entire population, else we risk *elitism*, premature convergence on what may not be the best overall solution.

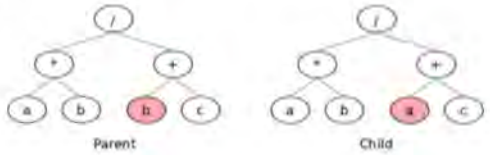
Reproduction

Through tournament selection, a single tree from the prior population is copied without mutation to the next generation. In the biological world, this is analogous to a member of a population entering the gene pool of the subsequent (younger) generation.



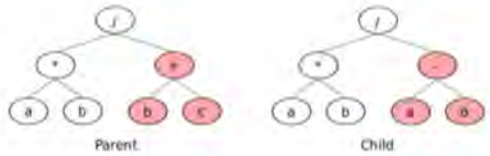
Point Mutation

Through tournament selection, a copy of a tree from the prior population mutates a single node before being added to the next generation. In the biological world, this may be analogous to asexual reproduction, that is, a copy of an individual with a minor mutation. In this method, a single point is selected for mutation while maintaining function nodes as operators and terminal nodes as terminals. The size and shape of the tree will remain identical.



Branch Mutation

Through tournament selection, a copy of a tree from the prior population mutates before being added to the next generation. In the biological world, this may be analogous to asexual reproduction, that is, a copy of an individual but with a potentially *substantial* mutation. Unlike Point Mutation, in this method an entire branch is selected. If the evolutionary run is designated as Full, the size and shape of the tree will remain identical, each node mutated sequentially, where functions remain functions and terminals remain terminals. If the evolutionary run is designated as Grow or Ramped Half/Half, the size and shape of the tree may grow smaller or larger, but it may not exceed the maximum depth defined by the user.



* The Field Guide and many other books about GP are listed at www.geneticprogramming.com

Karoo GP

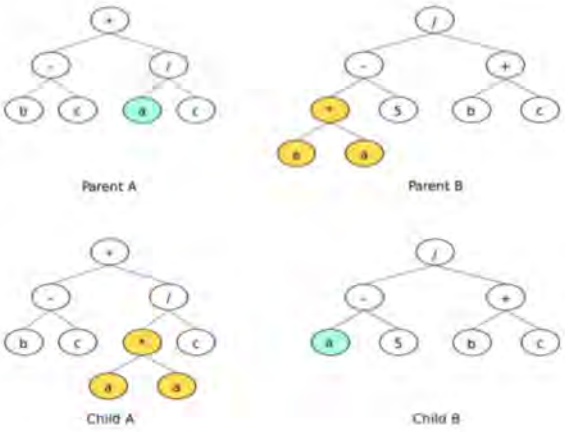
User Guide

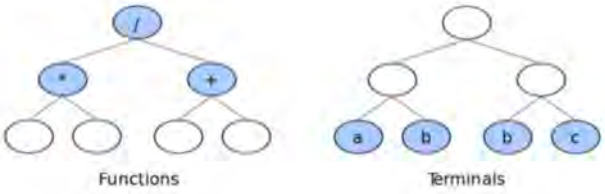
Crossover

Through tournament selection, 2 trees are selected as *parents* to produce 2 *offspring*. Within each parent tree a branch is selected. For child A, parent A is copied, with its selected branch deleted. Parent B's branch is then copied to the former location of parent A's branch, and inserted (grafted). This is reversed for child B. The size and shape of the offspring may be smaller or larger than either of the parents, but may not exceed the maximum depth defined by the user.

This process combines genetic code from trees which were chosen by the tournament process as having a higher fitness than the average population. Therefore, there is a higher probability their offspring will provide an improved fitness.

According to the literature, crossover is the most commonly applied evolutionary operator at 70-90%.

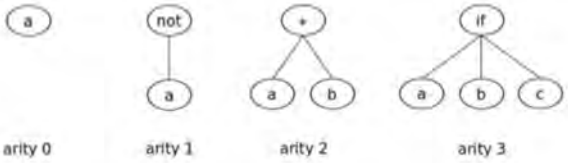




Functions (Operators)

The operators applied to Karoo GP are designated by you. The name of the .csv file which contains your desired operators must match the name of the fitness function (kernel) chosen. For example, the Classification kernel will auto-load /files/functions_CLASSIFY.csv.

The most commonly used are arithmetic [+, -, *, /], trigonometric [cos, sin], and Boolean [and, or]. In addition, [**, sqrt, exp] are supported. Many more have not been tested, but are open to exploration. As all Karoo GP expressions are processed by the library Sympy, you can learn about all possible operators at www.sympy.org



All operators presented to Karoo GP via the associated must be followed by an *arity*, that is, the number of variables or constants each operator expects to work with. For example, * has an arity of 2, meaning it expects something before and after itself, as in the expression *a * b*

The power operator (**) is not typically necessary as this is generated by any variable multiplied by itself. In the real world, most non-linear laws of nature are based on powers of 2 or 3, not ^5 or ^12. As such, GP finds these without the explicit operator. Square root too can be discovered through * and /, but this may be introduced if anticipated, to increase the efficiency of GP's search of the solution space.

As presented in karoo_gp/files/operators_list.txt, [cos, sin, exp, sqrt, log] must be preceded by another operator, as shown below. So, you will likely want to include all 4 of the arithmetic operators to provide a balanced set.

```
+ cos,2
- cos,2
* cos,2
/ cos,2
```

NOTE: *There is no space between the operator, the comma, and the arity.*

An example of the contents of a potential functions_[kernel].csv file follows:

```
operator, arity
+,2
-,2
*,2
/,2
+,2
-,2
*,2
/,2
+,2
-,2
*,2
/,2
+,2
-,2
*,2
/,2
+ cos,2
- cos,2
* cos,2
/ cos,2
```

NOTE: *The selection of any operator from this list is entirely random. The quantity of times any given operator appears relates to its chance of being selected, and therefore biases the outcome of the evolutionary process. If you have a single trigonometric operator which is represented 4 times, then you may desire to provide each of the arithmetic operators 4 times as well.*

Karoo GP

User Guide

Terminals (Operands)

Karoo GP automatically extracts the terminals (variables and constants) from the top of the data .csv file which you have provided. This enables you to name each column of your data to represent that particular feature. Later, when you read a GP derived expression, you will see how it relates to the features and original data.

Variables

If you are attempting to predict the weather using historical data[1] you might have something like this:

jd	sat	saz	hum	tmp	wm	wd	s
0	0.9292	1	0.2613	0.549	0.3956	0.0837	0
0.0092	0.9027	0.9865	0.2368	0.5862	0.4377	0.1328	0
0.0184	0.8761	0.9697	0.195	0.5908	0.4048	0.0913	0
0.0276	0.8496	0.9562	0.2021	0.6077	0.1905	0.1159	0
0.0368	0.8142	0.9461	0.2076	0.6156	0.2106	0.1221	0
0.0461	0.7788	0.9327	0.1821	0.611	0.185	0.0999	0
0.0553	0.7522	0.9226	0.1993	0.6099	0.152	0.3873	0
0.0645	0.7168	0.9125	0.2109	0.6043	0.2271	0.1416	0

Where “jd” is the variable for “Julian Day” (time), “sat” for the Sun's altitude, “saz” for the Sun's azimuth, “hum” for relative humidity, “tmp” for temperature, “wm” for the magnitude of the wind, and “wd” for the direction of the wind. The final column, “s” is the solution, that is, what you are are solving for. In this case, we are applying a regression algorithm, with a minimisation function approaching zero in order to discover a correlation between the given variables.

Constants

You can inject constants into the mix. In the current implementation of Karoo GP, constants are added at the top of unique columns in the data .csv. The entire column associated with each constant is then filled with zeros. These zeros are space fillers which have no impact on the processing of the data or operator.

For example, you might add a set of constant values [-5,-4,-3,-2,-1,1,2,3,4,5] which are balanced across the origin. Or, if you have a sense of how the variables need to be weighted, you may choose a particular set of constants. The same weather dataset above could include 5 constants expressed as decimal fractions, as the predicted variations in weights applied to this normalised data is relatively small.

jd	sat	saz	hum	tmp	wm	wd	0.1	0.2	0.3	0.4	0.5	s
0	0.9292	1	0.2613	0.549	0.3956	0.0837	0	0	0	0	0	0
0.0092	0.9027	0.9865	0.2368	0.5862	0.4377	0.1328	0	0	0	0	0	0
0.0184	0.8761	0.9697	0.195	0.5908	0.4048	0.0913	0	0	0	0	0	0
0.0276	0.8496	0.9562	0.2021	0.6077	0.1905	0.1159	0	0	0	0	0	0
0.0368	0.8142	0.9461	0.2076	0.6156	0.2106	0.1221	0	0	0	0	0	0
0.0461	0.7788	0.9327	0.1821	0.611	0.185	0.0999	0	0	0	0	0	0
0.0553	0.7522	0.9226	0.1993	0.6099	0.152	0.3873	0	0	0	0	0	0
0.0645	0.7168	0.9125	0.2109	0.6043	0.2271	0.1416	0	0	0	0	0	0

Future versions of Karoo GP will include constants as defined by the user via the interface.

[1] data from the SALT weather monitoring station and weather prediction software developed by Dr. Stephen Potter, head astronomer, South African Astronomical Observatory; Karoo GP is used to determine the weights

Data Management

Default Data Format

While Karoo GP can be recoded to work with any data format, by default, it anticipates .csv (comma separated values) files as readily exported from spreadsheet applications. Each row in a data .csv file is a data point (eg: a stamp in a time series), with 2 or more columns which hold raw data or a values derived through *feature engineering* (more below).

In supervised machine learning, it is standard for the right-most column to be the solution, either a value for which a regression algorithm is working to approach, or a label for each of 2 or more classes. Karoo GP requires the header for this column to be labeled lower-case "s".

With the Iris dataset (included with Karoo GP as a demonstration of multi-class classification), each row represents a single flower in the real world. The columns hold the measurements of parts of each flower. The right-most column holds the solution, in this case the classification [0,1,2] for each of the 3 species. No matter if you are working in biology, sociology, or astrophysics, the function of the rows and columns remain the same.

Loading your Dataset

Karoo GP is designed for easily repeatable experimentation. When you launch Karoo GP, data and function files associated with the user selected kernel are auto-loaded. For example, if you select "c" for classification, then "files/data_CLASSIFY.csv" (see *Functions*) "files/functions_CLASSIFY.csv" is loaded accordingly.

When you are ready to apply your own dataset, you can replace the data in one of the .csv files in the files/ directory (keeping the file name the same) or point Karoo GP to a properly prepared .csv which contains your data, as described in the Karoo GP Applications sections for Desktop and Server, above.

NOTE: *No matter which dataset you load the operators associated with the selected kernel will be loaded. As such, you must first edit files/functions_[kernel].csv before your Karoo GP run.*

A few things to keep in mind:

1. Learn about the Sort and Normalise scripts (see *Tools* below) included with this package which will help you properly prepare your own dataset.
2. If you generate your own dataset using a spreadsheet application, be certain to save the original spreadsheet as .ods or .xls in addition to the .csv required by Karoo GP in order to preserve any formula, notes, or formatting for reference or later modification.
3. If you replace the contents of an existing .csv in the files directory with your own, pay attention to the following:
   - If editing the .csv using a text editor, do NOT add spaces between the variables and the comma.
   - Label each column with a variable. For example: *a,b,c* or *a1,a2,a3* or give each column a variable which represents the real-world measurement, such as *t* for time or *p* for pressure.
   - The right-most column must be labeled "s" (lower case, no spaces before or after).
4. Be *very* wary of your editor applying hidden characters in the .csv as these will cause Karoo to crash.
5. If sending a .csv file through email, wrap it in an archive (.zip, .tar.gz) else data corruption might occur.

NOTE: *Your .csv must conform to the format described in the above or bad things will happen.*

Data Size

In theory, the size of the .csv file is limited only by the amount of RAM in your computer … and your patience. A practical run of GP is often against 10,000 rows of data. Runs of hours, even more a day or two are not uncommon. Use of 8, 12, or 24 cores on a single computer motherboard drastically improves performance.

Train vs Test

Using a SciKit Learn module, Karoo GP auto-splits all datasets greater than 10 lines into TRAINING and TEST in order to offer data on which to test that is unique from the training. However, if you present Karoo with 10 or fewer rows in your .csv, it automatically copies the same data into both TRAIN and TEST as it is assumed you are using Karoo GP in an instructional or experimental mode. This will not provide a valid run.

Tree Population Management

All trees generated for all populations of any given Karoo GP run are stored in files/ as follows:

population_a.csv - includes the foundation population and all subsequent evolved populations. If you run 10 generations with 100 trees each, you will have a .csv file with 1000 trees.

population_b.csv - written only if you export a snapshot of the evolutionary process (at the *pause* menu) using the (w)rite command.

population_final.csv - the final generation of GP trees should, if all goes well, offer the most fit trees of all the generations. While this final generation is also included in population_a, this file is isolated in order to make it easier for you to locate and work with these solutions.

TREE_ID	100								
tree_type	g								
tree_depth_base	4								
NODE_ID	1	2	3	4	5	6	7	8	9
node_depth	0	1	1	2	2	3	3	3	3
node_type	root	func	term	func	func	term	term	term	term
node_label	*	*	b	+	-	b	b	c	a
node_parent		1	1	2	2	4	4	5	5
node_arity	2	2	0	2	2	0	0	0	0
node_c1	2	4		6	8				
node_c2	3	5		7	9				
node_c3									
fitness	5			19					

Archiving and Loading Seed Populations

To archive a population of trees at any time, copy files/population_f.csv to a safe place.

When you desire to restore a population, rename the prior as population_s.csv and move it back into the files/ directory. From the *pause* menu, select "load" and immediately, the former population_s will be copied to *and overwrite* population_a which is the foundation population for the next generation.

With Karoo GP you can load a former population or a hand-crafted, *seed population* at any *pause*. This enables you to work with archived populations to carry forward with an evolutionary run with a new set of parameters to test a new hypothesis, work against a new dataset, or conduct a new series of validations. What's more, you can start with the same population and test any number of evolutionary parameter configurations over varying generations in order to determine which enables the most rapid convergence on a desired solution.

If you desire to load an archived population select the (g)eneration or (i)interactive display mode. Just after launch, at the first *pause*, conduct the load, configure your run, and then press ENTER to continue.

NOTE: *The Karoo GP population files are very specific in their format. Do not alter the file structure, in any way.*

Feature Engineering

If a dataset is a collection of raw data points, one or more values or measurements related to observation of the real world, then features are *transformed data*, values extracted from the observational data. According to Dr. Arun Aniyan, post-doctorate fellow at the Square Kilometer Array, South Africa, features are:

- Informative and non-redundant.
- Support subsequent learning and generalization.
- Reduce dimensionality (in most cases).
- Lead to better human interpretation of the overall data.

Feature engineering is the meat of machine learning, the most arduous and challenging aspect in which you become intimately familiar with your data in order to feed the machine learning algorithm *only* those features which give it the greatest chance of successful learning.

There are 3 basic processes within feature engineering: selection, extraction, and construction, as follows.

Feature Selection

Feature selection is the process of selecting a subset of relevant features which best describe the dataset as a whole, as some features may not be rich in discriminatory power. Feature selection helps reduce the dimensionality of the data in order that the machine learning algorithm is more readily able to learn. Common methods include comparing features via histograms or using genetic programming as a preprocessor to select which features to retain, and which to throw out.

SciKit Learn offers freely available methods for conducting Feature Selection in Python. Feature selection may also be conducted with Genetic Programming, as discussed by Bing Xue, et al, in this paper.

Feature Extraction

Feature extraction is an automated method of working with observational data to build new, derived, informative and non-redundant features which lead to better interpretation of the overall data. Because many observations produce far too much data to be used directly in machine learning, feature extraction *reduces* dimensionality of the data while at the same time improves the success of the predictive models.

Common examples include audio, image, text, and tabular data (eg: the Iris dataset or time series data) with millions of data points. For tabular data, Principal Component Analysis (PCA) and unsupervised clustering methods might be employed. For image data, filters familiar to graphic designers such as edge detection or quantitative analysis of a particular colour might be employed.

Key to feature extraction is that the methods are automatic (even if designed and constructed from simpler methods) which work to solve the problem of unmanageably high dimensional data.

SciKit Learn offers freely available methods for conducting Feature Extraction.

Feature Construction

Feature construction is similar to extraction in that you are working to build new, derived, informative and non-redundant features, but instead of an automated process, it is done manually. There are no standard means to this process, for it may differ for each dataset.

One method is to use Genetic Programming to build higher-order features, polynomial expressions which are in and of themselves demonstrations of relationships between two or more lower-level feature or raw data. This process is described by Soha Ahmed, et al, in "Multiple feature construction for effective biomarker identification and classification using genetic programming"

Develop your own Fitness Functions

Fitness functions lie at the heart of all machine learning. They may be very simple, classifying according to a value greater than or equal to zero as class 1, or less than zero as class 0. Or they may be very complex, applying external scripts which track trends in the classification process itself, even applying statistical analysis to determine how to adjust the fitness functions parameters over time.

Advanced users may modify existing fitness functions (kernels) or create their own. This guide does not provide a detailed guide, rather a pointer to the places in Karoo GP which require modification, as follows:

- In karoo_gp_main.py and/or karoo_gp_server.py, add the new kernel to the query for gp.kernel
- In karoo_gp_base_class.py:
  1. In the method fx_karoo_data_load, modify the dictionaries: data_dict, func_dict, and fitt_dict
  2. In the method fx_fitness_gym, search for "[other]" and modify.
  3. In the method fx_fitness_eval, search for "[other]" and modify.
  4. Now build your new fitness function as a Python method. Search for "def fx_fitness_function_[other]" to find the place-holder. Keep in mind that due to the multi-core functionality of this particular section, it is imperative that your sum be passed back through the pprocess portal, not saved as a global variable. Refer to the other fitness functions for examples.

Tools

The following scripts are found inside the included directory karoo_gp/tools/

Karoo Data Sort

In machine learning, it is often the case that your engaged dataset is derived from a larger one. In constructing the subset, if we grab a series of data points (rows in a .csv) from the larger dataset in sequential order, only from the top, middle, or bottom, we will likely bias the new dataset and incorrectly train the machine learning algorithm. Therefore, it is imperative that we engage a random array, guided only by the number of data points for each class.

*This script can be used *before* karoo_normalise.py, and assumes no header has yet been applied to the .csv.*

Karoo Data Normalise

This script works with a raw dataset to prepare a new, normalised dataset. It does so by comparing all values in each given column, finding the maximum and minimum values, and then modifying each value to fall between a high of 1 and low of 0. The modified values are written to a new file, the original remaining untouched.

*This script can be used *after* karoo_features_sort.py, and assumes no header has yet been applied to the .csv.*

Karoo Multiclass Classifier

This is a toy script, designed to allow you to play with multiclass classification using the same underlying function as employed by Karoo GP. Keep in mind that binary classification and multi-class classification are very different functions. You can enter finite or infinite *wing* bins, and as many classes as you desire:

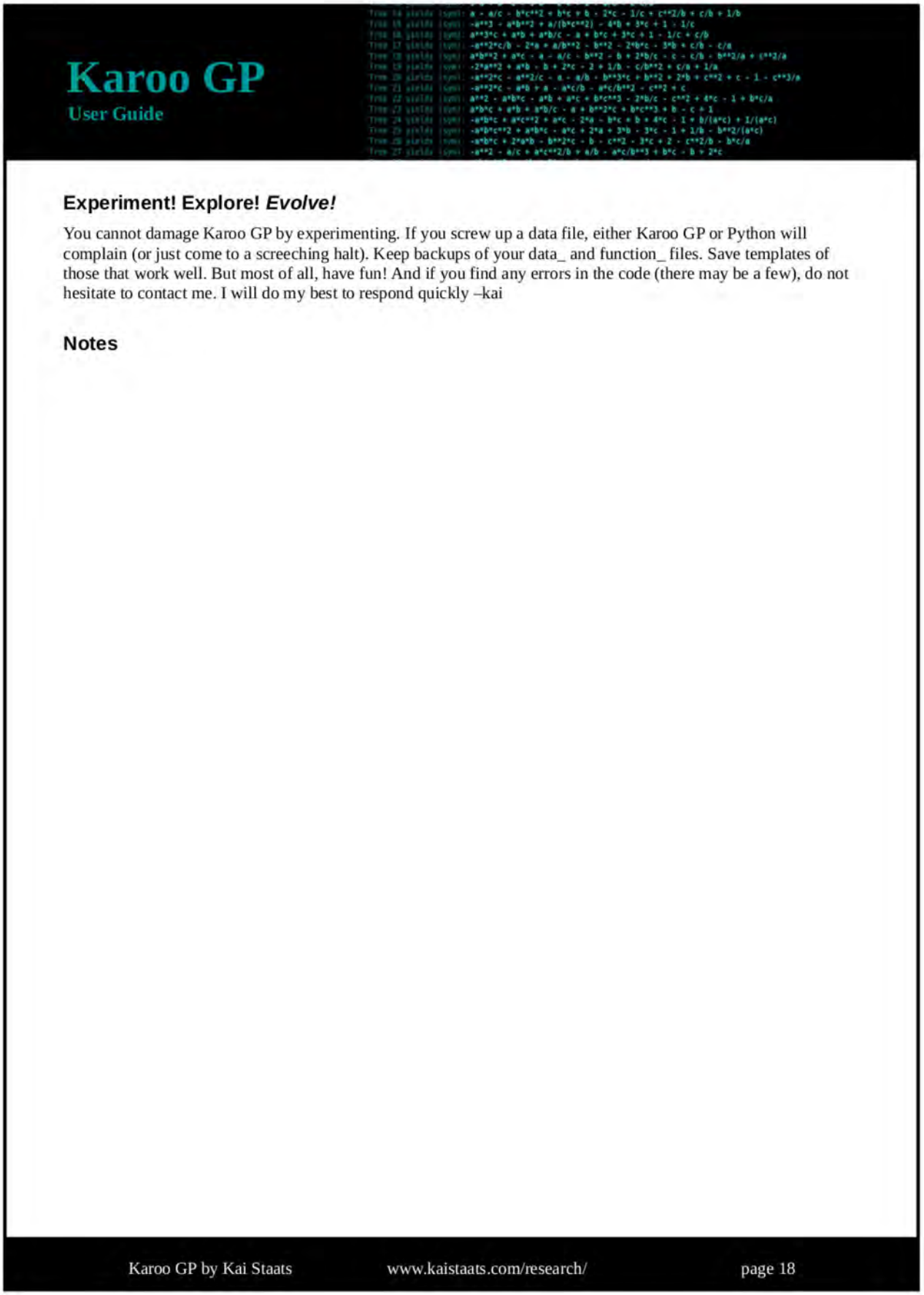
class 0 as -1.0 <= 0
class 0 as -0.5 <= 0
class 0 as 0.0 <= 0
class 1 as 0 < 0.5 <= 1
class 1 as 0 < 1.0 <= 1
class 2 as 1.5 > 1
class 2 as 2.0 > 1
class 2 as 2.5 > 1

While this classifier will scale to n classes, the success of GP will diminish dramatically once you surpass 2 classes unless your data exhibits sequential structure which fits neatly into sequential bins. It is typical that a higher dimension, multiclass classifier algorithm is employed (not currently included with Karoo GP), or multiple, binary classification runs are executed in a class_n:[other] fashion in order to extract each class from the dataset.

Karoo Iris Plot

This is a terribly rudimentary script which helps you visualise your 2D or 3D data against an expression generated by Karoo GP. The challenge comes with solving complex equations for a single variable such that you have a plot-able expression. If the algebra required is beyond your skills (or you forgot what you learned in high school), tools such as Matlab may be of some assistance. If you desire to normalise your data in advance of using this script, the Karoo GP normalisation script included in the karoo_gp/tools/ directory is very easy to use.

By default, this script plots a Karoo GP derived expression against a scatter plot of one of the Iris datasets included with this package: karoo_gp/files/Iris_dataset/data_IRIS_virginica-vs-setosa_3-col_PLOT.csv


Karoo GP
User Guide
Experiment! Explore! Evolve!
You cannot damage Karoo GP by experimenting. If you screw up a data file, either Karoo GP or Python will complain (or just come to a screeching halt). Keep backups of your data_ and function_ files. Save templates of those that work well. But most of all, have fun! And if you find any errors in the code (there may be a few), do not hesitate to contact me. I will do my best to respond quickly –kai
Notes
Karoo GP by Kai Staats
www.kaistaats.com/research/
page 18

Appendix B

Karoo GP Workflow

1. Set Genetic Programming parameters:
   (a) Select Fitness Function: Regression, Classification, or Matching
   (b) Select Tree type: Full, Grow, or Ramped Half/Half
   (c) Set Maximum Tree Depth
   (d) Set Minimum Number of Nodes
   (e) Set Maximum Population
   (f) Set Number of Generations
   (g) Select user display mode: Interactive, Minimal, Generational, Server, Debug, or Timer
   (h) Select percentage of subsequent populations to evolve through Reproduction, Point Mutation, Branch Mutation, and Crossover
   (i) Set number of individuals to enter each Tournament
   (j) Set number of CPU cores for multicore processing
   (k) Set floating point precision
2. Construct First Generation of GP Trees
   (a) Load data from default or user-defined .csv files
   (b) Split data into Training (e.g. 80%) and Test (e.g. 20%) sub-datasets
   (c) Set initial generation identification (0)
   (d) Construct the first population of GP Trees:
      i. Assign 13 global variables to the tree array
      ii. Build the Root node

iii. Build the Function nodes
iv. Generate a single Function node
v. Define a single Function
vi. Build the Terminal nodes
vii. Generate a single Terminal node
viii. Define a single Terminal
ix. Link each parent node to its children
x. Commit the values of a new node to the array "tree"

3. Evaluate First Generation of GP Trees

(a) Extract multivariate expression from each GP Tree
(b) Evaluate fitness (e.g. Regression, Classification, or Matching)
(c) Store fitness score with each GP Tree
(d) Save the first generation of GP Trees to disk

4. Evolve Multiple Generations

(a) Generate the viable gene pool
(b) Apply Genetic Operator: *Reproduction*
(c) Apply Genetic Operator: *Point Mutation*
(d) Apply Genetic Operator: *Branch Mutation*
(e) Apply Genetic Operator: *Crossover*
(f) Evaluate fitness (e.g. Regression, Classification, or Matching)

5. Terminate or Continue

(a) Check current Generation ID against user-defined maximum
    i. Continue with next generation; or
    ii. Terminate run, placing user in *pause* mode
(b) At *pause*, check for user invoked change to run-time configuration:
    i. Change display mode
    ii. Adjust the Tournament Size
    iii. Adjust the Min Node Count
    iv. Adjust the balance of genetic operators
    v. Adjust the number of engaged CPU cores
    vi. Display the generation ID

vii. List all Trees with the best fitness score

viii. Print a Tree to screen

ix. Evaluate a Tree for Accuracy (Train) or Precision and Recall (Test)

x. Continue evolution, starting with the current population

xi. Load *population_s* to replace *population_a*

xii. Write the evolving *population_b* to disk

xiii. Quit Karoo GP without saving *population_b*

Appendix C

Karoo GP Code

C.1 Overview

Initially designed and written for this MSc research, Karoo GP [1] is an evolutionary algorithm, a Genetic Programming application suite. Written in the programming language Python, Karoo GP calls upon the open source libraries numpy, sympy, pprocess, and sklearn. It has been fully tested, as of the writing of this document, on Ubuntu Linux 14.04 running on an Intel i7 (4 cores), Ubuntu Linux 14.04.3 LTS on an Intel Xeon E5-2650 (40 cores), and Apple OS X version 10.x on a variety of laptops.

Karoo GP is composed of three principal bodies of code, as follows:

- Karoo GP Desktop
- Karoo GP Server
- Karoo GP base class

The desktop application provides an intuitive user interface. The server application provides a fully scriptable, command-line interface. The base class contains all methods used by the desktop and server applications. Two additional scripts, Karoo Sort and Karoo Normalise aid in data management, and are found in the tools/ directory.

C.2 Karoo GP Main

Karoo GP *Main* (karoo_gpkaroo_gp_main.py) provides the user an interactive configuration with a text-based user interface (TUI). Each user query is programmed to provide

[1] The Karoo GP suite, sample data sets, and User Guide are freely available from Github.

a default setting such that the user may repeatedly press ENTER to launch a GP run. As the data and operand configuration files are associated by default, this is a rapid means to gain quick access to GP and machine learning, with no prior experience nor even programming.

The TUI is carefully coded to catch exceptions such that only desired results are allowed. An example is provided, as follows:

C.2.1 Sample Code

```
while True:
  try:
    kernel = raw_input('\t Select (a)bs diff, (c)lassify, (m)atch, or \
    (p)lay (default m): ')
    if kernel not in ('a','b','c','m','p',''): raise ValueError()
    kernel = kernel or 'm'; break
  except ValueError: print 'Select from the options given. Try again ...'
```

C.3 Karoo GP Server

Karoo GP *Server* (karoo_gpkaroo_gp_server.py)provides the user with a fully configurable, argument-driven, command-line executable for repeat GP runs with scripted configurations.

The full code of the server version of Karoo GP is provided below.

C.3.1 Full Code

```
ap = argparse.ArgumentParser(description = 'Karoo GP Server')
ap.add_argument('-ker', action = 'store', dest = 'kernel', default = 'm'
ap.add_argument('-typ', action = 'store', dest = 'type', default = 'r'
ap.add_argument('-bas', action = 'store', dest = 'depth_base', default = 3
ap.add_argument('-max', action = 'store', dest = 'depth_max', default = 3
ap.add_argument('-min', action = 'store', dest = 'depth_min', default = 3
ap.add_argument('-pop', action = 'store', dest = 'pop_max', default = 100
ap.add_argument('-gen', action = 'store', dest = 'gen_max', default = 10
ap.add_argument('-fil', action = 'store', dest = 'filename'
```

```
args = ap.parse_args()

# set default parameters which may be over-written by user arguments
gp.kernel = str(args.kernel)
tree_type = str(args.type)
tree_depth_base = int(args.depth_base)
gp.tree_depth_max = int(args.depth_max)
gp.tree_depth_min = int(args.depth_min)
gp.tree_pop_max = int(args.pop_max)
gp.generation_max = int(args.gen_max)
filename = str(args.filename)

gp.display = 'n' # display mode is set to (s)ilent
gp.evolve_repro = int(0.1 * gp.tree_pop_max) # percentage by Reproduction
gp.evolve_point = int(0.1 * gp.tree_pop_max) # percentage by Point Mutation
gp.evolve_branch = int(0.2 * gp.tree_pop_max) # percentage by Branch Mutation
gp.evolve_cross = int(0.6 * gp.tree_pop_max) # percentage by Crossover

gp.tourn_size = 10 # qty of individuals entered into each tournament
gp.cores = 1 # replace '1' with 'int(gp.core_count)' to auto-set to max
gp.precision = 4 # floating points for round function in 'fx_fitness_eval'

gp.karoo_gp(tree_type, tree_depth_base, filename) # run Karoo GP
```

C.4 Karoo GP Base Class

The bulk of Karoo GP is contained in the Object Oriented *base class* (karoo_gpkaroo_gp_base_class.py), the library of Python methods which give Karoo GP its functionality. As noted in the opening comments of this body of code, there are six primary categories of Python methods:

- 'fx_karoo_' Methods to Run Karoo GP
- 'fx_gen_' Methods to Generate a Tree
- 'fx_eva_' Methods to Evaluate a Tree
- 'fx_fitness_' Methods to Evaluate Tree Fitness
- 'fx_evo_' Methods to Evolve a Population

- 'fx_test_' Methods to Test a Tree
- 'fx_tree_' Methods to Append & Archive

The body of code is fully commented. All methods include a description that is accessible from the command line using the standard *method.?* invocation. The higher level Karoo GP methods combine lower-level methods into simpler executables which ultimately enable the single-line Karoo GP Server execution with user applied arguments.

Code samples are provided below, each including a brief overview extracted from the method itself. The method and variable names have in some cases been simplified, for readability.

C.4.1 Sample Code: Branch Mutate

A copy of a tree from the prior generation is selected via tournament, then undergoes branch mutation before being added to the next generation. The pseudo-code for this operation is provided here.

```
tourn_winner = fx_fitness_tournament(tourn_size) # engage a tournament
branch = fx_evo_branch_select(tourn_winner) # select point of mutation

 if tourn_winner[type] == 'f': # perform Full mutation on 'tourn_winner'
  tourn_winner = fx_evo_full_mutate(tourn_winner, branch)

elif tourn_winner[type] == 'g': # perform Grow mutation on 'tourn_winner'
  tourn_winner = fx_evo_grow_mutate(tourn_winner, branch)

population_b.append(tourn_winner) # append array to next population
```

C.4.2 Sample Code: Evaluation of a Tree by Means of Recursion

This method is called in order to prepare a multivariate expression for any tree contained in the current population. It anticipates receipt of the starting node for recursion via the variable *node_id* and the Numpy array *tree* which contains the given tree.

```
def fx_eval_label(self, tree, node_id):

  if tree[8, node_id] == '0': # arity 0 for '[term]'
```

```
      return '(' + tree[6, node_id] + ')' # function or terminal

    else:
      if tree[8, node_id] == '1': # arity 1 for '[func] [term]'
        return fx_eval_label(tree, tree[9, node_id]) + tree[6, node_id]

      elif tree[8, node_id] == '2': # arity 2 for '[func] [term] [func]'
        return fx_eval_label(tree, tree[9, node_id]) + tree[6, node_id] \
        + fx_eval_label(tree, tree[10, node_id])

      elif tree[8, node_id] == '3': # 'if [term] then [term] else [term]'
        return tree[6, node_id] + fx_eval_label(tree, tree[9, node_id]) \
        + ' then ' + fx_eval_label(tree, tree[10, node_id]) + ' else ' \
        + fx_eval_label(tree, tree[11, node_id])
```

C.4.3 Sample Code: Tournament

Tournament contenders are randomly selected from the gene pool, and then compared for their respective fitness. The tournament is engaged for each of the four types of genetic evolution: reproduction, point mutation, branch mutation, and crossover.

```
  if fitness_type == 'max': # fitness function is Maximising

    # first time through, 'tourn_test' will be initialised below
    if fitness > tourn_test: # current 'fitness' is greater than prior
      if display == 'i':
        print '\t\033[36m Tree', tree_id, 'has fitness', fitness, '>', \
        tourn_test, 'and leads\033[0;0m'
      tourn_lead = tree_id # set 'TREE_ID' for the new leader
      tourn_test = fitness # set 'fitness' of the new leader

    elif fitness == tourn_test: # current 'fitness' is equal to prior
      if display == 'i': print '\t\033[36m Tree', tree_id, 'has \
      fitness', fitness, '=', tourn_test, 'and leads\033[0;0m'
      tourn_lead = tree_id # in case there is no variance in this tournament

    elif fitness < tourn_test: # current 'fitness' is less than prior
      if display == 'i': print '\t\033[36m Tree', tree_id, 'has \
      fitness', fitness, '<', tourn_test, 'and is ignored\033[0;0m'
```

C.4.4 Development of a Multiclass Classifier

As shared in Chapter 4, when employed as a classifier, GP works to assign a label to each data point by evolving a hypothesis whose output falls into one of n bins, where there are n total labels. Each bin is associated with one label. In the training process, the GP assigned label is compared to the user provided correct label, or solution (the rightmost column of the .csv file).

The classifier developed for Karoo GP is a simple, auto-scaling, multiclass classifier. Meaning, it auto-detects the number of class labels and assigns data to an equal number of bins, where each class will be placed in one of two or more bins [64].

Code for a simple three-class fitness function follows:

```
if 0 <= result < 1 and solution == 0:
  fitness = 1 # class 0

elif 1 <= result < 2 and solution == 1:
  fitness = 1 # class 1

elif 2 <= result < 3 and solution == 2:
  fitness = 1 # class 2

else: fitness = 0 # no class match
```

where class 0 includes the Euclidean values 0 to 1, class 1 includes 1 to 2, and class 2 includes 2 to 3. It is important to note that the minimum value is 0 and the maximum value approaches but does not equal 3. Therefore, all three bins are of equal Euclidean distance from minimum to maximum boundaries and therefore, one might assume, offer each fitness case produced by each tree an equal opportunity for classification.

However, in reading "Representing Classification Problems in GP" by Loveard and Ciesielski [65], it was noted in section 3.2, paragraph 3, that negative infinity ($-\infty$) and positive infinity ($+\infty$) were included with the two outer bins. In "Multiclass Classification using Genetic Programming", on pages 46-47 and in the section following page 57, William Smart describes a multiclass classification system which includes $-\infty$ and $+\infty$ in the outer bins [66] as well.

One might assume this would bias the lower and upper values, such that the hypothesis would more likely satisfy those relatively unconstrained, larger spaces than the smaller,

bounded bins. However, development of such a multiclass classifier proved to be fully functional, as the literature described.

Simplified, sample code for a fully scalable (unlimited number of bins) multiclass classification fitness function which employs both finite *and* infinite bins follows:

```
skew = (class_labels / 2)

if solution == 0 and result <= 0 - skew:
  fitness = 1 # class 0

elif solution == class_labels - 1 and result > (solution - 1) - skew:
  fitness = 1 # class 1

elif (solution - 1) - skew < result <= solution - skew: # class 2
  fitness = 1 # class 2

else:
  fitness = 0 # no class match
```

where the *skew* forces a binary classification to split over the origin.

This code, when invoked as a stand-alone test for three classes $[0, 1, 2]$ moves from a value of -1 to +2.0 in increments of 0.5:

class 0 as -1.0 <= boundary 0
no match for -1.0 in class 1
no match for -1.0 in class 2

class 0 as -0.5 <= boundary 0
no match for -0.5 in class 1
no match for -0.5 in class 2

class 0 as 0.0 <= boundary 0
no match for 0.0 in class 1
no match for 0.0 in class 2

no match for 0.5 in class 0
class 1 as boundary 0 < 0.5 <= boundary 1
no match for 0.5 in class 2

no match for 1.0 in class 0
class 1 as boundary 0 < 1.0 <= boundary 1
no match for 1.0 in class 2

no match for 1.5 in class 0
no match for 1.5 in class 1
class 2 as 1.5 > boundary 1

no match for 2.0 in class 0
no match for 2.0 in class 1
class 2 as 2.0 > boundary 1

This same code was tested for values ranging from 2 to 100, demonstrating the function of a fully scalable, multiclass classification fitness function which included -∞ and $+\infty$ in the outer most bins.

C.4.5 Karoo Sort

Karoo Sort (karoo_gptoolskaroo_sort.py)selects a user-defined number of data points (rows) from each class (in the case of a classification dataset) and prepares a new, randomly selected dataset such that no data point is selected more than once. For example, if the least of three classes contains 1000 data points, then the new sorted dataset will contain all of the least class, and 1000 of each of the other two, with no duplicates.

C.4.6 Karoo Normalise

Karoo Normalise (karoo_gptoolskaroo_normalise.py) may be applied to a full dataset, or to a subset generated by the application of Karoo Sort. This script finds the minimum and maximum values for each feature (column) across all data points (rows) and then prepares a normalised copy of the dataset, maintaining order, such that all modified values, per each column, include and are between 0 and 1.

Appendix D

Karoo GP in the Real World

D.1 SALT Weather Prediction

D.1.1 Introduction

For the past seven years the Southern African Large Telescope (SALT) weather station has archived data [67]: temperature, humidity, air pressure, wind speed and wind direction. Stephen Potter, Ph.D., head Astronomer at the South African Astronomical Observatory, was inclined to develop a system for predicting the weather just one to two hours in advance of the live readings such that astronomers working with the SALT instrument might be more effective in scheduling their observations, based upon predicted humidity and seeing conditions.

Dr. Potter's system takes the current, live weather readings and compares them against the full or binned archive for similar conditions. A "goodness of fit" score is built by comparing each historic snapshot to the current weather parameters. The 24 hours following the closest n matches are averaged for a near-future prediction of one to two hours. The accuracy of the predictor is routinely tested and tuned against the full, updated archive to account for seasonal trends and total climatic shifts.

This system employed data (Table D.1) whose parameters were of several, disparate number types: Julian Day (jd as a continuous, non-cyclical measure of time), sun's altitude (sat), sun's azimuth (saz), atmospheric pressure (ap), relative humidity (rh), temperature (t), wind magnitude (wm), and wind direction (wd).

This system provided positive results. However, Dr. Potter asked if the application of an Evolutionary Algorithm might improve the accuracy of the system's predictions,

perhaps increasing the functional range (number of hours into the future) that he could accurately "see".

jd	**sat**	**saz**	**ap**	**rh**	**tmp**	**wm**	**wd**
2454578.03484	35	318	83.09	25.34	11.53	3.8	43.58
2454578.04875	32	314	83.09	24.2	11.86	4.03	58.46
2454578.06264	29	309	83.09	22.25	11.9	3.85	45.9
2454578.07654	26	305	83.09	22.58	12.05	2.68	53.35
2454578.09044	22	302	83.09	22.84	12.12	2.79	55.24
2454578.10434	18	298	83.09	21.65	12.08	2.65	48.5
2454578.11823	15	295	83.09	22.45	12.07	2.47	135.63
2454578.13214	11	292	83.09	22.99	12.02	2.88	61.15
2454578.14603	7	289	83.09	23.58	11.79	3.11	55.69
2454578.15993	3	286	83.09	24.39	11.74	2.82	41.9

TABLE D.1: Sample original SAAO weather data where *jd* is the Julian Day, *sat* the solar altitude, *saz* the solar azimuth, *ap* the atmospheric pressure, *rh* the relative humidity, *tmp* the temperature, *wm* the wind magnitude, and *wd* the wind direction.

D.1.2 Opportunity for Improvement

When I began work with Steve, we considered completely replacing his statistical method with Genetic Programming where it would discover relationships between the weather features. However, through many discussions, design sessions, and fine-tuning of his existing system, we determined the best approach would be to a) use GP to determine which weather parameters were correlated; b) normalise the data (Table D.2); then c) weight those parameters such that the "goodness of fit" function was no longer a simple sum, but a GP modified sum of weighted variables.

jd	**sat**	**saz**	**ap**	**rh**	**tmp**	**wm**	**wd**	**s**
0	0.9292	1	1	0.2613	0.549	0.3956	0.0837	0
0.0092	0.9027	0.9865	1	0.2368	0.5862	0.4377	0.1328	0
0.0184	0.8761	0.9697	1	0.195	0.5908	0.4048	0.0913	0
0.0276	0.8496	0.9562	1	0.2021	0.6077	0.1905	0.1159	0
0.0368	0.8142	0.9461	1	0.2076	0.6156	0.2106	0.1221	0
0.0461	0.7788	0.9327	1	0.1821	0.611	0.185	0.0999	0
0.0553	0.7522	0.9226	1	0.1993	0.6099	0.152	0.3873	0
0.0645	0.7168	0.9125	1	0.2109	0.6043	0.2271	0.1416	0
0.0737	0.6814	0.9024	1	0.2235	0.5784	0.2692	0.1236	0
0.0829	0.646	0.8923	1	0.2409	0.5727	0.2161	0.0781	0

TABLE D.2: The same data presented in Table D.1, but normalised such that all values include and are between 0 and 1.

D.1.3 Data Runs

Karoo GP was initially used in feature selection, that is, data runs conducted in order to discover which features drop out, and which are retained. This is done by providing all features, and their associate columns of data, to Karoo GP. Through repeated runs of various configurations, primarily increasing and decreasing the maximum tree depth, one reviews the final expressions and identifies which features are always included, and which are not.

In this effort, the solution (right-most column) is set to a single number, such as 0. I then employed Karoo GP's symbolic regression kernel and its inherent minimisation fitness function. This fitness function drives the evolutionary process toward, but not exactly equal to the provided solution.

In this particular data run, with the Function set [+, -, *, /] and the following configuration parameters:

kernel = a
tree_type = r
tree_depth_max = 5
tree_depth_min = 8
tree_pop_max = 100
generation_max = 10

The leading trees and their associated expressions:

Tree 6 : $rh * sat * saz * wd^2 * wm^2$
Tree 9 : $rh * sat * saz * wd * wm^3$
Tree 14 : $rh^2 * sat * saz * wd * wm^2$
Tree 23 : $rh^3 * sat * tmp^2 * wm^2$
Tree 28 : $rh^2 * sat * saz * tmp * wd * wm$
Tree 29 : $rh^3 * sat * tmp^3 * wm$
Tree 54 : $rh * sat * saz * tmp * wd * wm^3$
Tree 92 : $rh^2 * sat * saz * tmp * wd * wm^2$

Keep in mind that with a minimisation fitness function in Karoo GP, the tree with the highest ID number will offer a fitness value equal to or higher then all prior trees. Therefore, we select the tree with the highest ID number to test again against the 20% test data.

Tree 92 was tested with the following results:

Tree 92 yields (raw): $(rh) * (sat) * (wm) * (saz) * (tmp) * (wd) * (wm) * (rh)$
Tree 92 yields (sym): $rh^2 * sat * saz * tmp * wd * wm^2$

data row 0 yields: 0.0004
data row 1 yields: 0.0
data row 2 yields: 0.0
data row 3 yields: 0.0
data row 4 yields: 0.0
data row 5 yields: 0.0013
data row 6 yields: 0.0001
data row 7 yields: 0.0
data row 8 yields: 0.0
data row 9 yields: 0.0
data row 10 yields: 0.0003
data row 11 yields: 0.0001
data row 12 yields: 0.0002
data row 13 yields: 0.0007
data row 14 yields: 0.0001
data row 15 yields: 0.0004
data row 16 yields: 0.0009
data row 17 yields: 0.0002
data row 18 yields: 0.0014
data row 19 yields: 0.0001

In Table D.3 is a comparison of the original features introduced to GP, and those that remain in the final hypotheses:

Features IN:	jd	sat	saz	ap	rh	tmp	wm	wd
Features OUT:	-	sat	saz	-	rh	tmp	wm	wd

TABLE D.3: A review of features *dropped* and *retained* by Tree 92

D.1.4 Conclusion

What is immediately evident is a consistent retention, across all the showcased trees, of three key features in all eight hypotheses: rh, sat, and wm. Meaning, the relative humidity, sun's altitude, and wind magnitude appear to be important factors in predicting the weather. In the final five trees (which in the execution of Karoo GP are equal to or improved upon for accuracy over the first three trees), temperature is also retained while the Julian Day and atmospheric pressure dropped completely from all leading trees.

Ultimately, many more data runs will be conducted, followed by the subsequent application of those features which are retained with a new series of GP runs to determine the weighting of each feature through the application of coefficients.

In the end, the "goodness of fit" would no longer be a simple sum of raw features. Rather, a weighted sum of the normalised features provides subtle variation in how the features are correlated. This supports a more accurate weather model. This fine tuning may reduce the error bars in the prediction cone allowing the SALT weather prediction system can see further into the future, and support a more effective queuing system.

This work is continuing, with publication anticipated in 2016.

D.2 Additional Research

D.2.1 SKA-SA

Since my return to the United States, I have remained in contact with co-supervisor and post-doctoral fellow Dr. Arun Aniyan and associated staff members at the Pinelands Office of the Square Kilometre Array, South Africa. I regularly, remotely attend the machine learning Journal Club, and once presented about my current work at LIGO.

It is my intent, and that of Dr. Aniyan, to continue our work in the application of Karoo GP to RFI mitigation, working with an improved featureset, and to apply Karoo GP to the task of feature construction, generating multi-dimensional correlations between features for application to an Artificial Neural Network.

D.2.2 LIGO

I am now conducting research as a member of the Laser Interferometer Gravitational-wave Detector (LIGO) Scientific Collaboration (LSC). Working with Dr. Marco Cavaglia, assistant spokesperson for LIGO and professor at the University of Mississippi, in the application of evolutionary computation to the classification of glitches (Figure D.1), machine generated noise.

We are using the Omicron Trigger Generator, a transient search algorithm derived from the transient search pipeline called Omega [68], where as many as 200,000 degrees of freedom are associated with the LIGO instruments in Hanford, Washington and Livingston, Louisiana.

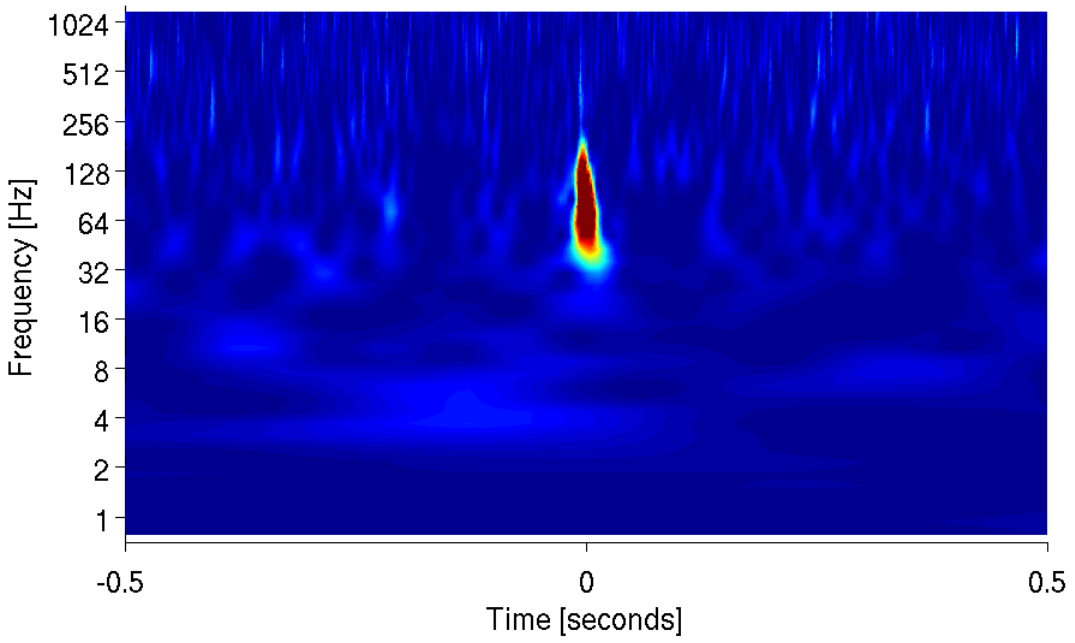


FIGURE D.1: A short-duration glitch in the Laser Interferometer Gravitational-wave Observatory, in the frequency range of 32 to 256 Hz [LIGO].

In our initial effort, we extract a dataset composed of features from Omicron: *duration*, $central frequency$, *bandwidth*, Q, SNR, *amplitude*, and *phase*. As with the application of Karoo GP at the SKA-SA, some features drop while others are retained. Future efforts will expand the featureset from the primary strain channel to auxiliary channels in order to benefit from cross-correlations which should improve the performance of the glitch classification effort.

The basic workflow is as follows:

1. From Omicron, extract a random subset of the full LIGO 01 run at a particular SNR threshold which does *not* include the glitch we desire to flag, and label as 0.
2. Manually extract the GPS times associated with the desired, human verified glitches, and label as 1.
3. Combine both labelled datasets into one.
4. Split the combined dataset into thirds, using the first two-thirds for training/testing of Karoo GP and the remaining third as the *real world* set-aside.
5. Run Karoo GP and extract the highest scoring hypothesis.
6. Run the pipeline with the extracted hypothesis against the set-side data.
7. Evaluate the data points labelled as glitch or non-glitch.

The final effort will be to inject synthetic wave forms to make certain the desired signal (e.g. coalescing binary black holes) is not inadvertently mis-classified as a glitch. As each glitch will be trained separately, an ensemble approach to classification will enable Karoo GP to provide a highly portable, flexible means of flagging one or more of the known glitches, either in post-production or real-time data analysis.

D.2.3 OSU CCAPP

In February of 2016 I was invited to give a brief introduction to Evolutionary Computation to the ANITA research group at The Ohio State University's Center for Cosmology and Astro-particle Physics (CCAPP). Shortly thereafter, a CCAPP graduate student applied Karoo GP to his dataset with immediate success. This reshaped the course of his research and led to the collaborative development of the first of its kind, hands-on workshop "Computing in High-Energy AstroParticle Research (CHEAPR2016)", July 24-26, Columbus, Ohio.

I was co-organizer and a lead presenter at the workshop which drew two dozen students and lecturers from a half-dozen states and associated universities, including two leading GP researchers at MIT. The interactive format engaged students in both presenting and learning about evolutionary computation applied to astro-particle physics. The central focus was on neutrino detection with instruments ANITA, ARRA, and ARIANNA, with further discussion around glitch classification at LIGO.

Karoo GP was featured as the primary tool, and was used by nearly all who attended in engagement of their own, real-world data. It is likely we will build upon the success of this forum and host a similar event in 2017.

Bibliography

[1] James J. Condon and Scott M. Ransom. *Essential Radio Astronomy*. Princeton University, 2016. ISBN 9781400881161.

[2] Arun Aniyan. *Feature Extraction and Analysis of Scientific Data*. PhD thesis, Mahatma Gandhi University, 2014.

[3] Steven W Smith et al. The scientist and engineer's guide to digital signal processing. 1997. California Technical Pub. San Diego.

[4] National Radio Astronomy Observatory (NRAO). Karl jansky and the discovery of cosmic radio waves, 2008. URL www.nrao.edu/whatisra/hist_jansky.shtml.

[5] Karl G. Jansky. Radio waves from outside the solar system. *Nature*, 132(3323), 1933.

[6] National Radio Astronomy Observatory (NRAO). Grote reber and his radio telescope, 2008. URL www.nrao.edu/whatisra/hist_reber.shtml.

[7] Ska radio astronomy workshop. SKA, December 2014. SKA-SA hosted workshop.

[8] W. T. Sullivan. *The beginnings of Australian radio astronomy*, volume 8. Journal of Astronomical History and Heritage, 2005.

[9] Peter Napier, Richard Thompson, and Ronald Ekers. The very large array: Design and performance of a modern synthesis radio telescope. *Proceedings of the IEEE*, 71(11):1295–1320, 1983. IEEE.

[10] Peter Dewdney, Peter Hall, Richard Schilizzi, and T Joseph LW Lazio. The square kilometre array. *Proceedings of the IEEE*, 97(8):1482–1496, 2009. IEEE.

[11] AR Foley, T Alberts, RP Armstrong, A Barta, EF Bauermeister, H Bester, S Blose, RS Booth, DH Botha, SJ Buchner, et al. Engineering and science highlights of the kat-7 radio telescope. *Monthly Notices of the Royal Astronomical Society*, 460(2): 1664–1679, 2016. Oxford University Press.

[12] Philippa Hillebrand. Literature review. May 2014. University of Cape Town.

[13] Gary Doran. *Characterizing Interference in Radio Astronomy Observations through Active and Unsupervised Learning*. NASA Jet Propulsion Laboratory, California Institute of Technology, August 2012.

[14] Goran Porko and Jukka-Pekka. Radio frequency interference in radio astronomy. Master's thesis, AALTO University, School of Electrical Engineering, Department of Radio Science and Engineering, 2011.

[15] Parliament of South Africa. Astronomy geographic advantage act 2007. 516(31157), 2007. Republic of South Africa.

[16] A. R. Offringa. The low-frequency environment of the murchison widefield array: radio-frequency interference analysis and mitigation. 2015. URL arxiv.org/abs/1501.03946.

[17] A. R. Offringa, J. J. van de Gronde, and J. B. T. M. Roerdink. A morphological algorithm for improved radio-frequency interference detection. *A&A*, 539, March 2012. Software application.

[18] RV Urvashi, A Pramesh Rao, and Pune NCRA. Automatic rfi identification and flagging. Technical report, Tech. Rep, 2003.

[19] P Dewdney, W Turner, R Millenaar, R McCool, J Lazio, and T Cornwell. Ska1 system baseline design. *Document number SKA-TEL-SKO-DD-001 Revision*, 1(1), 2013.

[20] Kiri L Wagstaff, Benyang Tang, David R Thompson, Shakeh Khudikyan, Jane Wyngaard, Adam T Deller, Divya Palaniswamy, Steven J Tingay, and Randall B Wayth. A machine learning classifier for fast radio burst detection at the vlba. *Publications of the Astronomical Society of the Pacific*, 128(966), 2016. IOP Publishing.

[21] Cornelis Johannes Wolfaardt. *Machine learning approach to radio frequency interference (RFI) classification in radio astronomy*. PhD thesis, Stellenbosch University, 2016.

[22] Joel Akereta, Chihway Changa, Aurelien Lucchib, and Alexandre Refregiera. Radio frequency interference mitigation using deep convolutional neural networks. *arXiv preprint arXiv:1609.09077*, 2016.

[23] Robert PW Duin and Elżbieta Pekalska. The science of pattern recognition. achievements and perspectives. In *Challenges for computational intelligence*, pages 221–259. Springer, 2007.

[24] Alan Turing. *Computing Machinery and Intelligence*. McGraw Hill, 1950. ISBN 0-07-042807-7.

[25] Nikita A Kangude and Sanil B Raut. Introduction to artificial intelligence. *International Journal of Computer Science Engineering and Technology*, 2, March 2012.

[26] Ron Kovahi. *Machine Learning*. Kluwer Academic Publishers, 1998.

[27] Phil Simon. *Too Big to Ignore: The Business Case for Big Data.* 2013. ISBN 978-1118638170.

[28] Venkat N. Gudivada, Ricardo Baeza-Yates, and Vijay V. Raghavann. *Big Data: Promises and Problems*. The IEEE Computer Society, 2015. ISBN 0018-9162/15/.

[29] Andrew Ng. Machine learning, 2014. URL www.coursera.org/learn/machine-learning/. Produced at Standford University for Coursera.

[30] Stephen Marsland. *Machine learning: an algorithmic perspective*. CRC press, 2015. ISBN 978-1466583283.

[31] Alice Zheng. *Mastering Feature Engineering*. O'Reilly Media, June], ISBN = 978-1-4919-5317-4 2016.

[32] Karl Pearson. Contributions to the mathematical theory of evolution. ii. skew variation in homogeneous material. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 186:343414, 1895.

[33] Yoav Freund, Robert Schapire, and N Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999. JAPANESE SOC ARTIFICIAL INTELL.

[34] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[35] Karl Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 11:559572, 1901.

[36] Bing Xue, Mengjie Zhang, Will Browne, and Xin Yao. A survey on evolutionary computation approaches to feature selection. 2015. IEEE.

[37] Soha Ahmed, Mengjie Zhang, Lifeng Peng, and Bing Xue. Multiple feature construction for effective biomarker identification and classification using genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 249–256. ACM, 2014.

[38] Derek Greene, Pádraig Cunningham, and Rudolf Mayer. Unsupervised learning and clustering. *Machine learning techniques for multimedia*, pages 51–90, 2008. Springer.

[39] Sotiris B Kotsiantis, I. Zaharakis , and P. Pintelas. *Supervised machine learning: A review of classification techniques*. 2007.

[40] H. William, Saul A. Teukolsky, William T. Vetterling, and B.P. Flannery. *Section 16.5. Support Vector Machines*. Cambridge University Press, 2007. ISBN 978-0-521-88068-8.

[41] Charles Darwin. *On The Origin of Species*. 1859. ISBN 0-8014-1319-2.

[42] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. ISBN 0-262-11170-5.

[43] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. ISBN 978-0262581110.

[44] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. creative commons, 2008. ISBN 978-1-4092-0073-4.

[45] SciKit Learn. Underfitting vs. overfitting, 2010-2016. URL scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html.

[46] *International vocabulary of metrology Basic and general concepts and associated terms*. Working Group 2 of the Joint Committee for Guides in Metrology (JCGM/WG 2, 200 edition, 2008.

[47] SciKit Learn. Precision-recall, 2010-2016. URL scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html.

[48] R.C. Lewontin. The units of selection. *Annual Review of Ecology and Systematics*, 1, 1970.

[49] A. M. Turing. Computing machinery and intelligence. *Mind*, 49:433–460, January 1950.

[50] Richard Forsyth. A darwinian approach to pattern recognition. *Kybernetes*, 10(3): 159–166, 1981.

[51] D.E. Goldberg. *Computer-aided gas pipeline operation using genetic algorithms and rule learning*. PhD thesis, University of Michigan at Ann Arbor, 1983.

[52] W.B.Langdon and Tao Chen. The genetic programming bibliography, 2016. URL www.cs.bham.ac.uk/~wbl/biblio/.

[53] John R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3/4):251–284, Sep 2010.

[54] Morgan Kaufmann. *Genetic Programming, An Introduction—On the Automatic Evolution of Computer Programs and its Applications*. Publishers Inc., 1998.

[55] Sara Silva, Pedro JN Silva, and Ernesto Costa. *Resource-limited genetic programming: Replacing tree depth limits*. Springer, 2005.

[56] Gabriel Kronberger, Michael Kommenda, Stefan Wagner, and Heinz Dobler. A framework-independent problem definition language for grammar-guided genetic programming. In *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1333–1340. ACM, Amsterdam, The Netherlands, Jul 2013. ISBN 978-1-4503-1964-5.

[57] Johannes Kepler, Max Caspar, Walther von Dyck, and Franz Hammer. *Astronomia nova*. CH Beck'sche Verlagsbuchhandlung, 1937.

[58] NASA Goddard Space Center. Kepler's planetary data set, 2005. URL www-spof.gsfc.nasa.gov/stargaze/Kep3laws.htm.

[59] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179188, 1936.

[60] Irvine University of California. Iris data set, 1936. URL archive.ics.uci.edu/ml/datasets/Iris. provided by the Center for Machine Learning and Intelligent Systems, University of California, Irvine.

[61] Cole Miller. Polarization and stokes parameters, early 1600s. URL www.astro.umd.edu/~miller/teaching/astr601/lecture10.pdf. Department of Astronomy, University of Maryland.

[62] SciKit Learn. Exhaustive grid search, 2010-2016. URL scikit-learn.org/stable/modules/grid_search.html.

[63] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014.

[64] SciKit Learn. Multiclass at scikit learn, 2010-2016. URL scikit-learn.org/stable/modules/multiclass.html.

[65] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. *IEEE Congress on Evolutionary Computation, Soul, Korea*, May 27-30:1072, 2001.

[66] William Richmond Smart. Genetic programming for multiclass object classification. Master's thesis, Victoria University of Wellington, 2005.

[67] S.B. Potter, Kai Staats, and Encarnacion Romero-Colmenero. Genetically optimizing weather predictions. In *Observatory Operations: Strategies, Processes, and Systems VI*, volume 9910. SPIE, Jul 2016.

[68] L K Nuttall, TJ Massinger, J Areeda, J Betzwieser, S Dwyer, A Effler, RP Fisher, P Fritschel, JS Kissel, AP Lundgren, et al. Improving the data quality of advanced ligo based on early engineering run results. *Classical and Quantum Gravity*, 32(24): 245005, 2015. IOP Publishing.