Utrecht University

Master Thesis

Automatically creating predictive features from datasets

Author:
Tim Waroux (5971241)

Daily supervisor:
Gerard Bos (Rabobank)

Project Supervisor:
dr. Cristian Spitoni (Utrecht University)

Second reader:
dr. Karma Dajani (Utrecht University)

October 29, 2021

Utrecht University

Rabobank

Abstract

Rabobank is one of the biggest banks in the Netherlands and a global leader in food and agriculture financing. Rabobank is developing a model for Early Warning Systems (EWS). These EWS are meant to predict if a client of Rabobank will go into default within the next 12 months. The EWS model is created using a Decision Tree classifier trained on time series data. To train a Decision Tree on time series data, feature engineering on the time series data is needed. This way predictive features are created that can be used in training the Decision Tree. Because clients of Rabobank are involved, these features need to be interpretable.

We will propose an algorithm based on the concept of Genetic Programming (GP) which will allow us to automate the feature engineering process for time series data. We will study the algorithm on various types of data, including data provided by Rabobank. We will answer the question if the algorithm can be used to create interpretable and predictive features. We will conclude that the proposed algorithm can guide as a helper to the data scientist performing feature engineering. The new features are predictive and can be interpreted in most cases.

Acknowledgement

This research was part of an internship at Rabobank. I would like to thank Rabobank for the opportunity to improve myself and learn more about the banking world. First I would like to thank my supervisor Gerard Bos and Heidene Bartie both working at Rabobank. I would like to thank them for their insight and support during the internship. Working with them was enjoyable because of their welcoming nature and knowledge. Also, I would like to thank my supervisor Cristian Spitoni from the Mathematics department at Utrecht University for his support and helpful comments during the process. In addition, I would like to thank the second reader Karma Dajani for the helpful comments.

Contents

1 Problem & Introduction

Rabobank is one of the biggest banks in the Netherlands, a global leader in food and agriculture financing. Therefore, Rabobank has a lot of rural clients, which will often be referred to as farmers. To help these farmers, Rabobank is developing a system that warns the bank that a client will possibly have difficulty paying back their loan in 12 months. These warnings, or triggers, are called *early warning signals* (EWS). The triggers are meant to give an early warning that the client might get into financial difficulty in the coming 12 months. If EWS returns a trigger for a client, a representative from the customer relations office gets notified. The representative then approaches the client who was triggered. It is explained to the client why he was triggered by the EWS model. This introduces a challenge. The trigger needs to be explainable. If the representative would tell the client, "You were triggered because our neural network-based model says so". It will most likely not have the desired effect. In addition, Neural Networks are considered 'black box' models and hence are very difficult to use with all the regulations the bank faces. If instead, the relations employee would explain to the client that he might be at risk of going into financial difficulty in the next 7-12 months because the rainfall in the area where the client operates has dropped by 5% over the past 5 months, it will be a lot clearer why the client is at risk. Hence, Rabobank opted for an easily interpretable and explainable model. This model is based on a *Decision Tree* (DT). Additional testing was done regarding model selection and Decision Tree proved to be the best candidate. It was chosen because Decision Tree is often interpretable. The Decision Tree is trained on different types of data. These consist of commodity prices, meteorological data and transaction data. For each client, various time series are created. An in-depth explanation of how the time series are created is shown in Section 5.4.1. For a specific client, we have a vector with values that are measured at certain time points $t_i \in \tau$, where τ is a set of time points that are 1 month apart.

$$T := \begin{bmatrix} x_{t_0} & \cdots & x_{t_n} \end{bmatrix}.$$

In addition to this t_i describes a certain month, say April 2012. We can think of x_{t_i} as the temperature at time t_i for the specific client. The time series T is not directly used to train the Decision Tree. Groups of new features are created which are based on T. This is important because the values x_{t_i} alone do not give much information. For example, the temperature of last month alone gives way less information compared to the increase of temperature compared to 2 months ago. In Section 5.3 we will see an example of the poor performance of Decision Tree on time series data. It shows that creating additional features for time series data does improve Decision Tree performance. The groups of features are created based on expert knowledge. An example of a group of features that is created from time series T is the 1 month difference. To describe 1 month difference we have for $t_i \in \tau$

$$d_{t_i} := x_{t_{i+1}} - x_{t_i},$$

which would exactly correspond to the difference between two consecutive months. A lot more features can be created, for example, lagged 1 month differences, % growth rates over 3, 6, 9, 12-months and lagged % growth rates over the same time periods. On top of that, the specific features created from a time series can differ per type of data. The above illustrates the procedure for using time series in DT. One has to create features from the time series which then

improve the performance of the DT. As already mentioned this creation procedure is expert-based. We might ask ourselves: 'Aren't we forgetting any important features?', we could for example ask ourselves whether or not it makes sense to also look at statistical measures of time series. For example, the mean or the standard deviation. Then, another question would be, for which time points do we calculate these statistical measures? We do not know which are the 'best' features to use in our Decision Tree. This thesis aims to explore a proposed method to extract optimal features concerning a certain fitness measure. Moreover, we want the method to be able to assist in the task of creating new features from a dataset. The relevance of the problem is illustrated by the fact that feature creation takes a lot of time, usually, it is the most time-consuming part of machine learning. In addition to this, there is an importance for correctly modelling these EWS. EWS serves two main purposes, they help the clients to prevent them from getting financial problems. But also for the bank, they help to minimize realized credit losses.

The remainder of this section will introduce related work from which we took inspiration in creating a solution for the feature construction problem. In Section 2 we will explain the basics of machine learning and introduce the Decision Tree which is at the center of multiple subjects in this thesis. The Decision Tree is the preferred model for Rabobank and the heuristics used in training a Decision Tree classifier heavily influenced our decisions made in later sections. In Section 3 we will introduce another core concept of this thesis. Genetic Programming (GP) will be at the base for the proposed algorithm which will be used for creating new features. A pivotal quantity for GP is the so-called fitness. We will introduce two main fitness functions, one of which will take heavy inspiration from the concept of Decision Tree introduced in the Section before. Furthermore, the introduction of these fitness functions will be a setup for a crucial discussion on which fitness function will be used in the algorithm. In Section 4 the algorithm that will be used to solve the Rabobank problem will be introduced and we will discuss which of the two proposed fitness functions will be best suited for our situation. In Section 5 we will go over the application of the developed algorithm. We will compare the algorithm to a random search algorithm, a standard algorithm. In addition to this, we will be comparing different selection methods with those that had been discussed in Section 3. We will also highlight the performance of the algorithm on artificial data with added noise. After this, we discuss the performance of the algorithm on meteorological data. To conclude the section we will discuss the performance and features constructed on the Rabobank dataset.

1.1 Related work in GP and inspiration

This section will give a brief introduction to the basic principles of Genetic Programming (GP), which is needed to give a good insight of the related work. The basic principles will, later on, be expanded in the thesis, see Section 3. After we have introduced the basic principles we will proceed with referencing papers that can be considered as an inspiration to our research in this thesis. Formally GP is the evolution of programs starting from a population that is generally unfit. It evolves the programs using genetic operators [1]. These programs can be anything from machine learning pipelines to functions [2, 3]. GP is a technique that offers great potential for classification problems. GP and other evolutionary techniques have been successfully applied to different supervised learning tasks like regression, but also in

unsupervised learning tasks like clustering and association discovery. GP can be seen as a probabilistic search algorithm: It searches a predefined space using evolutionary techniques based on the Darwinian theory of evolution. In fact, GP can be seen as a program searching for programs. The essential features shared by Evolutionary Algorithms (EA) as follows:

1. We have a population of individuals. These individuals are often called functions or programs.

2. A generational inheritance method. Genetic operators are applied to the individuals of a population to determine the next generation's population of individuals. The most often used Genetic operators are crossover (recombination) and mutation. Crossover swaps a part of the genetic material of two individuals, whereas mutation randomly changes a small portion of the genetic material of one individual.

3. A fitness-based selection method. The fitness function determines how fit a given individual is, a measure of quality. The better the fitness, the higher the probability that the given individual will be chosen to take part in breeding for the next generation.

On top of this, individuals are often chosen to be represented in a certain way. The most commonly used way is a scheme based on trees. This will be elaborated upon more clearly in Section 3.5.1. Since flexibility is one of the biggest advantages of GP it has been used quite extensively in the context of feature construction and feature selection. First, we will reference papers that mainly focused on feature construction. In the literature, there are often two main approaches to using GP in feature construction methods:

1. A filter approach [4]. Here, preprocessing is performed independently of the model's algorithm. In this case, the criterion on which the individuals are scored, i.e. the fitness, is some sort of data-dependent criterion. One example would be Information Gain or the Gini. These will be introduced in Section 2.2.

2. A wrapper approach [5]. In this situation preprocessing is done in combination with the model algorithm. That is, the individuals are scored according to some fitness related to the model's performance. More clearly, the individuals could be rated by their performance impact on the algorithm. Think about the impact of the individual on the correctly classified instances by the model.

To summarize, the filter approach only takes the data into account, while the wrapper approach often requires models to be trained or evaluated. The filter approaches followed roughly the same scheme. First, one defines a fitness function, then this fitness function is used in the application of GP. The results from their GP run would be fed into a machine learning algorithm ranging from Decision Trees and Support Vector Machines to Neural Networks.
In [6] four fitness functions are introduced, at the end of the evolutionary process there is one feature selected as the best. These fitness functions are all filter approaches and are closely related to the Decision Tree splitting criteria. They were Information Gain (IG), Gini impurity (GI) and a combination of IG and GI. The concept of IG and GI will be introduced in Section 2.2. When evolution ended one individual was chosen and then added to the original data. All the individuals in the population corresponded to exactly one created feature. The original data was expanded with the constructed feature. On the expanded data various machine

learning algorithms were trained. These algorithms included three Decision Tree algorithms (C5 [7], classification in a Decision Tree algorithm (CART)[7] and CHAID[7] and a neural network.
In [8] a modified version of the Fisher linear discriminant, often called Fisher criterion, was used as a fitness function. This fitness function measures the scattering of different classes. Therefore, an individual was scored by how well it can scatter the two (or multiple) classes in the dataset. Furthermore, the writers compared the modified Fisher criterion versus the standard Fisher criterion. Afterwards features created by GP were used in a multilayer perceptron (MLP) neural network and support vector machine (SVM) classifier. Again in this paper, the individuals corresponded to one feature.
It is often the case that the creation of new features is combined with dimensionality reduction. This is the case in [9], where a fitness function based on information entropy over class intervals was used. First, an orthogonal transformation function was used to construct a new variable terminal pool. The new variables were used in the GP algorithm. The constructed features were then ranked using the information entropy measure. The new features were then used to train a Decision Tree classifier. This approach was then compared with principle component analysis (PCA), where final classification performances were directly compared. This paper is also an example of a way to extract more than one feature using GP.
In [10] the authors used a completely different search algorithm. They used a gravitational search algorithm to search the space of agents, where each agent represents a constructed feature. But the goal of this paper was mostly to reduce the amount of non-informative features in the dataset.

The other approach from papers that we deem as related work are the papers that used the concept of wrappers. In [11] each individual in the population corresponded to a forest of n trees, where each tree in the forest corresponded to a different feature transformation. The fitness function then was the classification accuracy of the transformed original dataset by the n features in the forest. Where the classification was done by a k-nearest neighbors algorithm.
A combination of GP and GA was used in [12]. Genetic Programming was used to construct the features while GA performed the selection. The features were scored by various classifiers, namely C4.5 [7], k-nearest neighbours (kNN) and a bayesian classifier. In this paper, each individual corresponds to a forest containing n trees, and each tree corresponds to a function that is used to construct a feature. In this paper, n was chosen to be the number of features in the original dataset.

2 Machine Learning

The main problem of Rabobank is to classify, or label, if clients go into financial difficulty in a following certain time period. For each client we are trying to label, we have a set of characteristics called inputs. These inputs influence an output. In the case of the client at Rabobank, the output would be equal to 'financial difficulty' or 'no financial difficulty'. Our goal is to make predictions based on the inputs of clients. As introduced in Section 1, Rabobank uses a Decision Tree to make these predictions. Making a prediction based on the inputs is called supervised machine learning. In this section, we will give an example of supervised machine learning. Furthermore, this section will serve to introduce the concept of Decision Tree. In addition to this, we will state the heuristics used to defining a Decision Tree classifier. These heuristics formed the basis for developing the algorithm in Section 4. In machine learning, the inputs are often also called predictors or independent variables. We will often refer to these inputs as features, but all terminology can be used interchangeably. The output is often referred to as responses or dependent variables. This section is based on and inspired by [13] and [14].

2.1 A statistical framework for learning

In this section, we will consider the following simplified example that will guide us through explaining how learning and supervised learning work. Suppose that we have the task of organizing a korfball [15] tournament. To play korfball you need a korfball, and one of our jobs is to select balls that will be used for the matches played in our tournament. So given a ball we want to predict, or label, whether or not the ball is fit to be used as a korfball. Unfortunately, we have no experience playing korfball and hence we have no clue how to select balls that are fit for the matches. We do have experience with selection balls for football matches. So we use that experience to define criteria we will judge the balls on. These criteria are exactly the features that we will use to make our prediction. In this case, we think it is suitable to take the measurements of the ball, and the pressure. From our football experience, we know that footballs have specific measurements and pressure requirements. In the end, we are trying to figure out a prediction rule, which we then use to predict whether or not a ball could be used as a korfball. This prediction rule will be based on a dataset that we create. This dataset consists of a range of balls. We have gathered all the relevant information on the balls in the dataset. For each ball we know the measurements and pressure, after that we gave each ball to a professional korfball player and they gave us feedback on whether or not the ball was suited for playing.

Definition 1 (Instance space)**.** *An arbitrary set $\mathcal{X}$ is called the instance space if it consists of all the objects we are trying to label. Points in this space are called instances and are usually represented as a vector of features.*

Definition 2 (Label space)**.** *A set $\mathcal{Y}$ is called a label space if the points in this space consist of labels for a corresponding instance space $\mathcal{X}$.*

The labels are defined as y_i which are integers. Most of the time we will consider $\mathcal{Y}$ to be the set $\mathcal{Y} = \{0, 1\}$.

Definition 3 (Training dataset)**.** *A training dataset is a dataset* $\mathcal{D} := \{(x_1, y_1), \ldots, (x_n, y_n)\}$ *where the points* $(x, y) \in \mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$*. A dataset is often defined in combination with an instance space* $\mathcal{X}$ *and a label space* $\mathcal{Y}$*.*

With these definitions, we return to the korfball example to illustrate what they represent

Example 1 (The korfball dataset)**.** *We decided to model a ball by its measurements, which we now assume to be the diameter. In addition to this, we also consider the pressure of the ball. If we were to assume that all the balls we can consider have a maximum diameter of 30 cm, and a maximum pressure of 1 PSI. Our instance space* $\mathcal{X}$ *would be given by* $\mathcal{X} := [0, 30] \times [0, 1] \subset \mathbb{R}^2$*. The label space, in this example where we predict good or bad, is often defined as the set* $\mathcal{Y} := \{0, 1\}$ *or* $\{-1, 1\}$*. Where* 0 *or* -1 *corresponds to negative and* 1 *corresponds to a positive outcome. Our dataset* $\mathcal{D}$ *would consist of the balls* $x_i \in \mathcal{X}$ *which we have labeled by our professional with an outcome* $y_i \in \mathcal{Y}$*.*

Definition 4 (Prediction Rule)**.** *A prediction rule is a function* $h : \mathcal{X} \to \mathcal{Y}$*. This function can assign a label to every instance in the instance space* $\mathcal{X}$*. Often* h *is used to label unseen/new instances from* $\mathcal{X}$*. The function* h *is often referred to as the hypothesis or classifier and is obtained by applying a learning algorithm* A *to a training set* $\mathcal{D}$ *and is denoted by* $A(\mathcal{D})$*. So* $h = A(\mathcal{D})$*.*

Let $\mathbb{P}_{\mathcal{X}}$ be a probability distribution on $\mathcal{X}$. In the case of our korfball example, this distribution would be the environment from which the balls are chosen. Thus $\mathbb{P}_{\mathcal{X}}$ describes how likely it is that a certain korfball is drawn from the environment. We will also assume that there exists a function f such that $f : \mathcal{X} \to \mathcal{Y}$ and $f(x_i) = y_i$ for all $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$. The assumption that such a function f exists will later on be relaxed. Assuming that there is such a function f means that every instance can only have one label. Thus, f is the function giving the true label. We will assume that the data in $\mathcal{D}$ is generated in the following way, first we assume that the x_i are sampled from $\mathbb{P}_{\mathcal{X}}$, and then labeled by the function f. This labeling will give us the corresponding y_i.

The error of a classifier or hypothesis can be defined in various ways. We proceed in the following way and give it the following definition. We will define the error as the probability that a hypothesis h mislabels, or misclassifies, a certain instance. This results in the following definition.

Definition 5 (Error of a classifier on all instances)**.** *Let* h *be a classifier, see Definition 4, for the instance space* $\mathcal{X}$ *and label space* $\mathcal{Y}$*. Let* $f : \mathcal{X} \to \mathcal{Y}$ *be the function that always returns a correct label. Then for* $h : \mathcal{X} \to \mathcal{Y}$ *we define the error*

$$L_{\mathbb{P}_{\mathcal{X}}, f}(h) := \mathbb{P}_{\mathcal{X}}\left(h(x) \neq f(x)\right),$$

where $x \sim \mathbb{P}_{\mathcal{X}}$*.*

From Definition 5, it follows exactly that the error is given by the probability of randomly sampling an x from $\mathbb{P}_{\mathcal{X}}$ and then misclassifying it using h.

Remark 1. *Our algorithm used to obtain* h *does not know what* $\mathbb{P}_{\mathcal{X}}$ *is, it can only get a glimpse of the distribution by looking at the training set* $\mathcal{D}$*. This is important to note as often*

the quality of our classifier, which would be often seen as the error, heavily depends on the quality of $\mathcal{D}$. Often the quality of a training dataset $\mathcal{D}$ is considered to be how well it captures the distribution $\mathbb{P}_{\mathcal{X}}$. If for example, a training set consists of only 5 out of 100 different points in $\mathcal{X}$, it is very likely that the quality of the classifier h is low, which would result in a high error.

Just like the error of a classifier, which is based on the distribution $\mathbb{P}_{\mathcal{X}}$, we can define the learning error. This learning error is based on the training dataset $\mathcal{D}$. As we already motioned the algorithm A returns a rule h based on some training set $\mathcal{D}$. This set $\mathcal{D}$ is sampled from the unknown distribution $\mathbb{P}_{\mathcal{X}}$ and labeled by the function f. We want to construct a hypothesis $h_{\mathcal{D}} : \mathcal{X} \to \mathcal{Y}$. This hypothesis is obtained from $\mathcal{A}(\mathcal{D})$ and we want it to have a low error on all training instances. Therefore, we will consider the following definition of training error.

Definition 6 (Training error)**.** *Let $\mathcal{D}$ be a training dataset as defined in Definition 3. With $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and $h : \mathcal{X} \to \mathcal{Y}$ a hypothesis, the training error is given by*

$$L_{\mathcal{D}}(h) := \frac{|\{(x, y) \in \mathcal{D} | h(x) \neq y\}|}{n}.$$

The training error in Definition 6 is often also-called the empirical error. In the case where we assume the training data to be representative of the real distribution $\mathbb{P}_{\mathcal{X}}$ it makes sense that we could search for a solution that works well on the training data, as we expect the training data to generalize well to the real distribution $\mathbb{P}_{\mathcal{X}}$. This way of learning, where we come up with a hypothesis that minimizes $L_{\mathcal{D}}$ is called Empirical Risk Minimization (ERM).

Below Definition 4 we had assumed that there exists a function f that will always return the true label. As mentioned in [13], this assumption is often not feasible in practical problems. In the context of the korfball example, it is also not feasible to assume that the label is solely based on the characteristics we measure. We only have access to the pressure and diameter of the ball. What about the material the outside of the ball is made from? The material will also influence the label, yet we are not using it for prediction. Sometimes instances in our limited space $\mathcal{X}$ can then not be correctly labeled by the function f. Instead of considering the function f and $\mathbb{P}_{\mathcal{X}}$ we will consider $\mathbb{Q}$. We will assume that $\mathbb{Q}$ is a probability distribution on $\mathcal{X} \times \mathcal{Y}$. Where again $\mathcal{X}$ is the instance space and $\mathcal{Y}$ is the label space. Hence $\mathbb{Q}$ is a joint distribution. We assume that $\mathbb{Q}$ describes the probability that we sample an instance x with a label y from $\mathcal{X} \times \mathcal{Y}$. The joint distribution $\mathbb{Q}$ can be thought of as the composition of marginal distribution $\mathbb{Q}_x$ and a conditional probability $\mathbb{Q}((x, y)|x)$. In the korfball example, $\mathbb{Q}_x$ describes the probability that we encounter a ball with a specific diameter and pressure. Whereas $\mathbb{Q}((x, y)|x)$ describes the probability that a ball given its diameter and pressure has label y. Such way of modelling allows us to have two balls that share the same diameter and pressure but have different labels. We will also redefine the error of a classifier on the instance space by

$$L_{\mathbb{Q}}(h) := \mathbb{Q}(\{(x, y) : h(x) \neq y\}).$$

We will use this definition of error of a classifier for the next subsection about the Decision Tree.

2.2 Decision Tree

In this subsection, we will introduce the concept of Decision Tree (DT). The concept of DT is used by Rabobank to create a classifier based on the meteorological data. Furthermore, the heuristics used to determine the learning rule for Decision Trees will play a big role in Section 4. Again, this subsection is mainly based on [13]. A Decision Tree is a predictor $h : \mathcal{X} \to \mathcal{Y}$, that predicts (in the case of classification) the label associated with an input vector $x \in \mathcal{X}$. The input vector travels from the root of the tree towards a leaf. Let us assume that we are in a binary classification setting, i.e. $\mathcal{Y} = \{0, 1\}$, but it must be noted that the binary classification setting is not the only applicable area of DT. In fact, it is a widely used learning algorithm because of its simplicity and is also used in state of the art learning algorithms like xgboost [16] which apply gradient boosting to DT.

At each node on the path of the input vector a decision is made. The decision to go left, or right, is based on the decision rule in said node. Usually, this rule is based on one of the features of the input vector x. A leaf contains a specific label and that label is used as the prediction for the input vector x. In the context of the korfball example given in Example 1 an example of a Decision Tree is given by

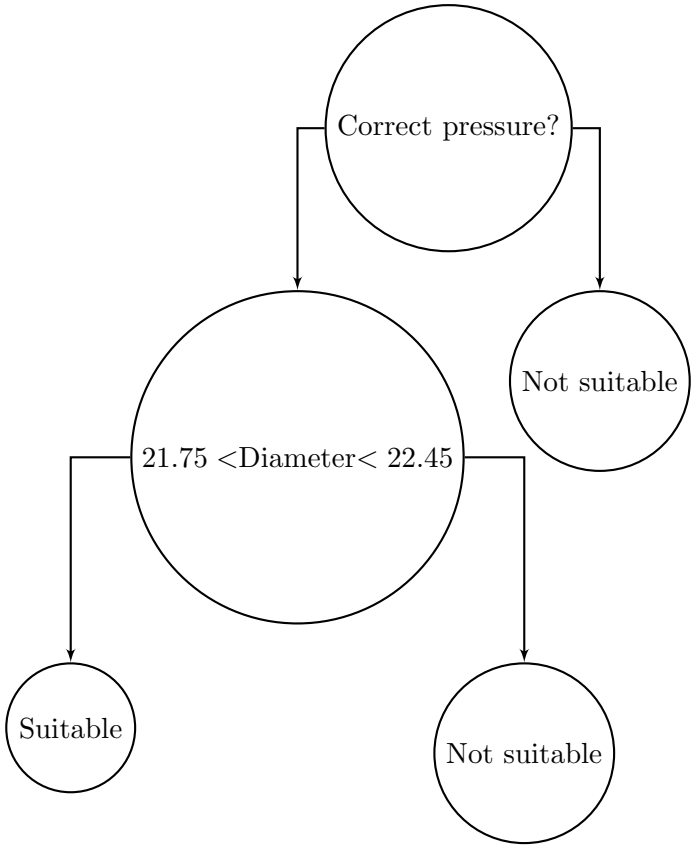


Figure 1: Example of a Decision Tree in the context of the korfball example.

We will assume that every internal node (that is every node that does not contain a label)

has two children. In other words, we will consider a binary tree. In addition to this, we will assume that $\mathcal{X} = \mathbb{R}^k$. Then it can be shown that, see Equation 18.1 in [13], that with probability $1-\delta$, a training set $\mathcal{D}$ with n samples, for every Decision Tree $h \in \mathcal{H}$ with l nodes we have that

$$L_{\mathbb{Q}}(h) \leq L_{\mathcal{D}}(h) + \sqrt{\frac{(l+1)\log_2(k+3) + \log(2/\delta)}{2n}}. \tag{1}$$

Our goal is to find a tree h that minimizes the right-hand side of (1). Unfortunately, this is an NP-hard problem [17]. Therefore, in practice, the search for a tree is based on heuristics. This heuristic is a greedy approach, where the tree is constructed from the top down. At every node a locally optimal decision is made. As described in [13] this approach cannot guarantee globally optimal trees, but it tends to work reasonably well in practise. The heuristic of making locally optimal decisions is what we based our algorithm, introduced in Section 4, on. We make locally optimal decisions in the following way. We will assume that $x_i \in \mathcal{X} \subset \mathbb{R}^k$, and that $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. We have that $x_i = \left(x_i^1, \ldots, x_i^k\right)$. Where x_i^k corresponds the value of instance i at feature k. For every feature $1 \leq i \leq k$ sort the instances such that $x_1^i \leq \cdots \leq x_n^i$. Then a set of thresholds is defined $\Theta := \{\Theta_1, \ldots, \Theta_k\}$, where $\Theta_i := \{\theta_{0,i}, \ldots, \theta_{n+1,i}\}$ such that $\theta_{j,i} \in (x_j^i, x_{j+1}^i)$. For x_0^i and x_{n+1}^i we use the convention that they are equal to $-\infty$ and ∞ respectively. Then we define a function for each $1 \leq i \leq k$ and $1 \leq j \leq n$. For an instance x_m we define $f_{i,j}(x_m) = \mathbf{1}\{x_m^i \leq \theta_{j,i}\}$. This returns 1 if the value of instance x_m at feature i is greater than $\theta_{j,i}$ and 0 otherwise. We then let $\{x^i > \theta_{j,i}\}$ describe the event that a feature i is greater than the threshold $\theta_{j,i}$. Then the locally optimal decision rule is the decision rule which leads to the highest gain. Let $\hat{p}_{\mathcal{D}}(E)$ be the empirical probability that an event E holds on $\mathcal{D}$. For example,

$$\hat{p}_{\mathcal{D}}(\{y = 1\}) = \frac{\sum_{j=1}^n \mathbf{1}\{y_j = 1\}}{n},$$
$$\hat{p}_{\mathcal{D}}(\{x^i > \theta_{j,i}\}) = \frac{\sum_{m=1}^n f_{i,j}(x_m)}{n}.$$

For a cost function $C : [0,1] \to \mathbb{R}$ the gain is defined as

$$\begin{aligned}\text{Gain}(\mathcal{D}, i, j) = C\left(\hat{p}_{\mathcal{D}}\{y = 1\}\right) \\ - \left(\hat{p}_{\mathcal{D}}\{x^i \leq \theta_{j,i}\} C\left(\hat{p}_{\mathcal{D}}\{y = 1 | x^i \leq \theta_{j,i}\}\right) + \hat{p}_{\mathcal{D}}\{x^i > \theta_{j,i}\} C\left(\hat{p}_{\mathcal{D}}\{y = 1 | x^i > \theta_{j,i}\}\right)\right).\end{aligned}$$

Hence the Gain function describes a way to measure the quality of splitting 1 labeled instances in terms of a cost function C. Where

$$\hat{p}_{\mathcal{D}}\{x^i \leq \theta_{j,i}\} C\left(\hat{p}_{\mathcal{D}}\{y = 1 | x^i \leq \theta_{j,i}\}\right) + \hat{p}_{\mathcal{D}}\{x^i > \theta_{j,i}\} C\left(\hat{p}_{\mathcal{D}}\{y = 1 | x^i > \theta_{j,i}\}\right),$$

describes the cost of when splitting on feature i. The decision rule is given by $\mathbf{1}\{x^i > \theta_{j,i}\}$, where i and j are chosen to maximize $\text{Gain}(\mathcal{D}, i, j)$. There are various cost functions that can be used

1. Information Gain [18] (IG) is a measure for which

$$C(x) = -x\log(x) - (1-x)\log(1-x).$$

2. Gini index [18] (GI) is a measure for which

$$C(x) = 2x(1-x).$$

In the Decision Tree created by Rabobank, the corresponding cost function is given by the Gini index. This is also one of the reasons why we will be using the Gini-based measure in the remaineder of the thesis. See Section 3.5.4 and Section 4.1.

Remark 2. *For DT we can talk about feature importance. Feature importance is calculated as the decrease in node impurity weighted by the probability of reaching that specific node. Since every node corresponds to a feature, we can calculate the feature importance by considering in which nodes the features occur. Feature importance of a feature f_1 indicates how many instances get split by f_1 and how good that split is.*

3 Genetic Programming

The following section will mainly be based on [19] and [20]. The goal of this section is to give an overview of the concept of Genetic Programming (GP). This is done by giving a historical overview and explaining how GP is different compared to other search methods. In Section 3.5 the fundamentals of GP are introduced, and we will explain in more detail how the brute force search method introduced in Section 3.4 will not work. The remainder of the Section will serve to achieve the basic understanding which will be needed for the algorithm explained in Section 4. For more information about the concept of Decision Tree see [13].

3.1 Introduction

In nature living creatures are often divided into groups called species. These species change over time (which could be comparatively short or long), i.e. evolve. There are a few components that are essential to natural evolution. The individuals (e.g. plants or animals) belonging to the population are different from each another. This fact has as a consequence that some individuals live longer than others and are more likely to produce offspring that lives to adulthood. These individuals are deemed to be fitter. They are fitter because of variations. These variations must have a genetic component. That means that these variations can be carried onto the offspring. Consequently, after some time generations have passed, the proportion of individuals with the beneficial inheritable variations tends to increase. In other words, the whole group of species tends to change or evolve.

In nature, many species produce offspring whose genes only come from one adult, e.g. think about how some plants have offspring grow from the parent's root or how viruses replicate [21, 22]. This produces offspring that are genetically similar to the adult where exceptions in the genetics occur as random copy errors called mutations. In other species, we have that two adults are needed to produce offspring. In this case, the offspring consists of a genetic combination of the two adults (e.g., humans). The process where a combination of the adults is used to produce offspring is called crossover. Various researchers [23, 24] have been using the idea of natural evolution in their computer programs, called Artificial Evolution, for many years. One of such techniques, to use Artificial Evolution, is called Genetic Programming (GP). Genetic Programming uses Artificial Evolution to create new computer programs, hence the name Genetic Programming. It must be noted that GP is a generalisation of a different artificial evolutionary technique, Genetic Algorithms (GA). Furthermore, GP is the main technique that will be used in this thesis.

It has been shown that GP can be applied to solve numerous problems in various areas, like automatic design [25], data mining [26], and fault prediction in software [27]. Genetic Programming is often depicted as a search algorithm, it searches in the space of all possible programs. This search is with respect to a fitness that scores how well a program performs on a specific problem. We will view evolution as a process of trying possibilities of programs and determining how fit these programs are. Then this fitness, which is just a measure of performance, will guide us through the search space to find more even fitter individuals. We can compare it to the Gradient Descent Algorithm (GD) [28]. In GP we move towards a locally optimal solution using the fitness (compared to the gradient in GD). We must note that in general, we have no continuity assumptions of the search space. As opposed to GD on a continuous differentiable surface, our fitness might make jumps. This search space is

in practice huge or even infinitely big. Hence we cannot try all possibilities, see Section 3.5.3. As already mentioned, we cannot guarantee, that the search space is differentiable or even continuous. We are dealing with programs instead of a subset of $\mathbb{R}^n$. So traditional numerical optimisation techniques often do not apply. Hence why we choose a stochastic search technique [29] like Genetic Programming which does not rely on differentiability.

3.2 Why use Genetic Programming?

According to [20] GP has done well in areas with the following properties:

First, the relationship between variables or features is unknown or not well understood. In light of the situation where Rabobank finds itself this corresponds to the features in the time series dataset. For these time series and their variables, which are just values at specific timepoints from the time series. We lack the information as to which timepoints have some underlying connection. It could be that if the temperature from last month and 2 years ago differ by a lot a client is more at risk. These connections are not clear for us the modeller. Second, We have some basic understanding of how they interact with each other, but not in-depth knowledge over longer time periods. For example, we have by subtracting consecutive time points values x_n and x_{n+1} a way to interpret the notion of gain (or loss) from timepoint n to $n+1$. Or when we divide the values x_n and x_{n+1} we get a certain growth factor (or decrease factor). How they precisely interact with each other, and what the best features are to create these transformations is not known. This problem is often solved by the use of expert knowledge where the data scientist creates the variables based on what they think is the best option. Third, if we are searching for a function of a predetermined size standard evolutionary algorithms work better. The strength of GP is that during the evolution the size of the function, which is just a tree, can vary. In addition to this GP can be used as a feature selection method. GP can figure out which features in the original dataset are of interest for modelling.

3.3 History of Genetic Programming and evolutionary computing

This section will contain information about the history of Genetic Algorithms, Evolutionary Computing, and Genetic Programming and will mainly be based on Chapters 2 and 3 in [30] and Chapter 6 in [31].

Evolutionary Algorithms or Evolutionary Computing is an area in computer science where certain search heuristics are applied which are inspired by natural evolution. It is applied in various domains. One great example is parameter optimization, but also in a broader setting, there is another good example. Namely, it is applied in machine learning pipeline optimization. Most notably a great recent example is the papers concerning the Python package TPOT [32, 33, 34]. TPOT is a Python library that optimizes machine learning pipelines using Genetic Programming. Before we had these recent packages and papers there was of course a beginning.

This beginning dates back to the time when computers were completely different. They could not fit on your desk and were as big as a room. In the paper [35] from 1950, Alan

Turing wondered whether or not computers were able to think. At that time computers could only execute deterministic code. Given a specific input, deterministic code always returns the same output. There is no randomness involved in the executing the code. Turing was slightly worried about the fact that computers might only be able to execute these deterministic pieces of code. If this would be the case he felt like this, i.e. the execution of deterministic code, was not sufficient to produce intelligent decisions or behaviour by the machine. He would later describe that he expects intelligence to be achieved in a way of computing that would deviate from the deterministic pieces of code. This deviation would be in the form of introducing a certain random behaviour. Adding this randomness could give rise to an interesting machine which could especially be useful when searching for solutions to problems. In the present, we have already seen that this randomness occurs in Gradient Descent and later on we will also see that are multiple elements of randomness in Genetic Programming. Turing later on in [35] described that this non-deterministic way of computing could be seen as the process of learning for computers and this process of learning he connected to biology. In [35] Turing also took inspiration from how children learn and described the concept of a child program which learns by an educational process which in itself would be inspired by evolution. These notions of randomness described by Turing would, in the context of evolutionary computing and Genetic Programming, become known as mutation and crossover.

In 1962 Holland wrote a paper about adaptive systems [36], these adaptive systems later became Genetic Algorithms. He wrote that the 'adaption' involved both the adaptive system and its environment. In general, it considers how system can generate procedures, the adaptation, which allows them to adjust to their environments. This environment foreshadows the need for a fitness function. The fitness function will be a substitute for the environment. In [36], Holland also stresses that we should look at these adaptive systems as a population of programs. This would have certain advantages as this would be more general. If the production of sets of populations of programs happens in a parallel fashion, opposed to only looking at individual programs.
Hence, in 1962 there was a concrete sketch for Evolutionary Algorithms. The randomness from Turing's paper would later on still be considered in Friedberg's paper [37], which introduced a variation selection loop. This is a method to select variations occurring in programs. Around the same time, developments were also made in Evolutionary Programming and Evolutionary Strategies (ES). These ES will turn out to be a very important building block in creating our implementation of GP. These evolutionary strategies will be explored later, see Section 3.6.

3.4 Fitness landscape & Search

In this subsection, we will introduce the concept of a fitness landscape. That can help us to understand the type of space we will be searching in. This space will be searched for the optimal functions described in the introduction, Section 1. In addition to this, we will go over different types of search methods that we could consider to search in a variety of spaces. This section should be treated as an overview of other options for searching.

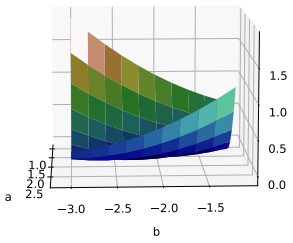
3.4.1 Fitness landscape

Fitness landscapes are landscapes (or surfaces) of fitness. For every point in the search space, the height at that specific point corresponds to the fitness. In general, the fitness landscape is nothing more than the image of a function $h : \mathcal{S} \to \mathbb{R}$. Where $\mathcal{S}$ is considered to be the search space. The search space is dependent on the problem itself and, in the sense of genetic algorithms, consists of combinations of genes. Consider the following example where we illustrate an example of a search space and the fitness landscape. A definition of a fitness function h will be given in Section 3.5.

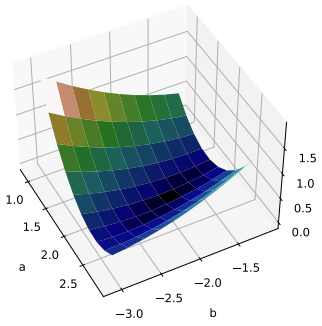
Example 2. *To give an example of a fitness landscape we consider the problem of fitting a function of the form $g(x, a, b) = ax^2 + b$ on the true function $f(x) = 2x^2 - 2$. Suppose that in this specific case we know that genes are $a \in \mathbb{R}$ and $b \in \mathbb{R}$, suppose we also know that we can limit our search to the Cartesian product $[1, 3] \times [-3, -1]$. In this situation, our search space $\mathcal{S} := [1, 3] \times [-3, -1]$ and thus the genes are points $(a, b) \in [1, 3] \times [-3, -1]$. We let our fitness be the Mean Squared Error (MSE) on the interval $[-1, 1]$. So the fitness landscape will be the image of the function $h : S \to \mathbb{R}$ where*

$$h(a, b) := \frac{1}{n} \sum_{i=1}^{n} \left(g\left(x_i, a, b\right) - f\left(x_i\right) \right)^2,$$

where the $x_i \in [-1, 1]$. Indeed, h depends on the chosen points $x_1, \ldots, x_n$. Hence we are considering a approximation of the landscape. In this case, the fitness landscape will look, by approximation, like this



(a) Fitness landscape with on z-axis the MSE.



(b) Fitness landscape with on z-axis the MSE, slightly rotated.

We say by approximation because we can only calculate the MSE for a finite amount of points in the interval $[-1, 1]$. For the plots in the figure $n = 10000$ where all x_i are evenly spaced. The fitness is calculated for a finite subset of the Cartesian product $[1, 3] \times [-3, -1]$, hence the grid-like structure on the surface. As we can see the MSE for the point $(2, -2)$ is equal to zero as that point corresponds to the exact function we are trying to determine the values of a, b for.

Now that we have specified what a fitness landscape is, we need a way to search through it. The fitness landscape namely corresponds to the quality of individuals. In the context of

Example 2, we are trying to find an individual (a^*, b^*) which minimizes the fitness function h. Where again we note that h depends on the $x_1, \ldots, x_n$. There are various methods to search through a search space.

3.4.2 Brute-force search

One way to search for the optimal solution is to simply brute-force the problem. In brute-forcing we determine the optimal solution by calculating the fitness of all the possible gene combinations. To refer back to Example 2, if we were to assume, or know, that the optimal gene combination (a^*, b^*) lies in the set $\tilde{\mathcal{S}} := \{[1, 3] \times [-3, -1]\} \cap \{\mathbb{Z} \times \mathbb{Z}\}$ we would simply calculate the fitness for every $(a, b) \in \tilde{\mathcal{S}}$. Then $(a^*, b^*) := \arg\min_{(a,b)\in\tilde{\mathcal{S}}} h(a, b)$ would be our solution for the problem. In the case where the search space $\mathcal{S}$ is finite we are guaranteed to find a global optimum as the $\arg\min_{(a,b)\in\mathcal{S}} h(a, b)$ always exists for finite $\mathcal{S}$. In general an exhaustive search is the only certain way to always find the best solution in the search space.

3.4.3 Gradient descent

Gradient descent is an algorithm that uses the gradient of the fitness function to navigate the landscape. The algorithm is used to find local optima of a differentiable function. So that $\theta^* := \arg\min_{\theta\in\mathcal{S}} F(\theta)$, where $\mathcal{S}$ is the search space. It is an iterative search method and, relies on the fitness function (or cost function) to be differentiable. The algorithm works in the following way. First, a starting point θ_0 is chosen, random uniformly in the search space. So that $\theta_i \in \mathcal{S}$ for all i, where i is some index referring to the current number of iterations. Then we apply the following iteration until we reach some stopping criterion

$$\theta_{n+1} = \theta_n + \gamma_n \nabla F(\theta_n),$$

where $\nabla F(\theta)$ is the gradient of the function F, and γ_n is the learning rate. In the setting of Example 2 it means that $F(\theta) := h(a, b)$, where $\theta := (\theta_0, \theta_1) \in [1, 3] \times [-3, 1]$. To apply gradient descent we would first have to calculate the gradient of F. It follows that

$$\frac{\partial}{\partial\theta_0} F(\theta) := \frac{\partial}{\partial a} h(a, b) = \frac{2}{N} \sum_{i=1}^{N} \left(ax_i^2 + b - f(x_i)\right) x_i^2,$$
$$\frac{\partial}{\partial\theta_1} F(\theta) := \frac{\partial}{\partial b} h(a, b) = \frac{2}{N} \sum_{i=1}^{N} \left(ax_i^2 + b - f(x_i)\right).$$

Now let $\theta_0 = (0, 0)$, $\gamma_n = 0.5$ for all $1 \leq n \leq N$ and $N = 70$. Plotting the θ_i will then result in the following figure

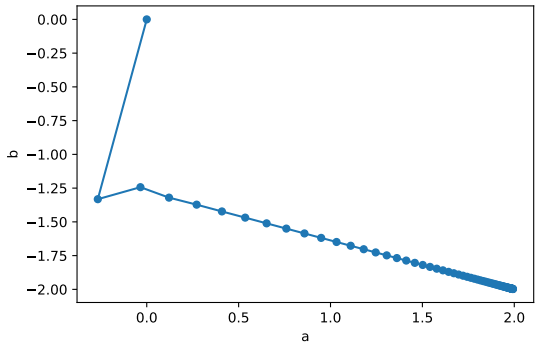


Figure 3: Plot of all θ_i obtained from the Gradient Descent Algorithm.

As we can see in Figure 3 the algorithm converges towards the optimum $\theta^* = (2, -2)$. Now at the beginning of this subsection we motioned that gradient descent converges to a local optimum. This is not a problem when there is only one optimum, because then the local optimum is a global optimum. In the case when there are multiple points, say $\hat{\theta}$ and $\tilde{\theta}$, for which we have that $\nabla F(\hat{\theta}) = \nabla F(\tilde{\theta}) = 0$ there is no guarantee that we will always end up in the point θ^*, which we defined as $\arg\min_{\theta \in \mathcal{S}} F(\theta)$. In general gradient descent guarantees to converge to a global optimum when the function F is convex (minimization) or concave (maximization). To overcome the assumption of convexity or concavity of the function F one could use stochastic gradient descent [28].

3.4.4 No free lunch

The no free lunch (NFL) theory was originally introduced by Wolpert and Macready in [38]. They developed a framework to explore the connection between effective optimization algorithms and the problems that they try to solve. Informally NFL means that when an algorithm is evaluated over all possible problems, all algorithms will perform equally good or bad irrespective of the evaluation criteria. Formally for any algorithms a_1 and a_2, at iteration step m one has that

$$\sum_f P\left(d_m^y | f, m, a_1\right) = \sum_f P\left(d_m^y | f, m, a_2\right),$$

where d_m^y corresponds to the ordered set of size m which contains the cost values y associated with the input values $x \in X$ where X is the search space, f is the function being optimized where $f : \mathcal{X} \to Y$. In this situation Y is the space of possible cost values. The results from [38] caused uproar in the research community. One of the assumptions of Wolpert and Macready was to consider the set of all discrete functions. One argument of researchers dismissing the NFL theory argued that concerning the set of all discrete functions is not applicable in the real world, the reasoning behind this was that the set is not representative of real-world problems. A second response to the paper was that researchers who were trying to develop the best possible algorithms for a specific use case often had to use extensive problem-specific knowledge. The NFL theory then confirmed the intuitive idea that there is no 'best' algorithm for all problems.

Considering GP and NFL there is, to the best of our knowledge, no NFL theorem that applies to GP specifically. Although it has been argued that for search spaces where there is a non-uniform many to one genotype-phenotype mapping there is a free lunch. It can be argued that GP search spaces have this characteristic. Namely, many different trees represent the same function. That means, there are many genotypes (the trees) that map to the same function (phenotype). We illustrate this with the following example in Figure 4. How to read these functions will be shown later in Section 3.5.1.

(a) A tree representing the syntax $(+x_1x_2)$. Where x_1 and x_2 are children and + corresponds to the parent.

(b) A tree representing the syntax $(+x_2x_1)$. Where x_1 and x_2 are children and + corresponds to the parent.

Figure 4: Two trees that are the same function.

As can be seen, by the figure, both trees represent a different syntax. Whilst both function represent the same function $f(x) = x_1 + x_2$.

3.5 Genetic Programming fundamentals

In this subsection, we will further specify some of the terminologies which were introduced in the introduction of this chapter and give the fundamentals of Genetic Programming in general which is needed to understand the remaining parts about Genetic Programming of this thesis.

Definition 7 (Individual)**.** *Let $\mathcal{X}$ be a sample space, i.e. a set of instances. Then an individual I is a function $I : \mathcal{X} \to \mathbb{R}$.*

Definition 8 (Population)**.** *A population $\mathcal{P}$ is a set of individuals. So $\mathcal{P}$ can also be considered as a set of functions. The population size is given by $|\mathcal{P}|$.*

Remark 3. *As mentioned in Definition 8 we denote with $\mathcal{P}$ the set of individuals. Which is not to be confused with the population from which $\mathcal{D}$ is sampled. Here $\mathcal{D}$ is referred to as the training dataset introduced in Section 2. In particular, $\mathcal{D}$ is sampled from the space $\mathcal{X} \times \mathcal{Y}$. $\mathcal{D}$ is sampled according to a probability distribution on $\mathcal{X} \times \mathcal{Y}$, which is $\mathbb{Q}$.*

Now that we have the definition for individuals and the population we can define a fitness function

Definition 9 (Fitness function)**.** *Let $\mathcal{I}$ be the set of all possible individuals and $\mathcal{P} \subseteq \mathcal{I}$ a population. A function f is called a fitness function if $f : \mathcal{I} \to S$, with $S \subseteq \mathbb{R}$ and where the outcome of the function quantifies the optimality of an individual $I \in \mathcal{P} \subseteq \mathcal{I}$. These values can be used as a ranking to compare individuals in the population. Fitness functions can either be minimized or maximized. In both cases, we use to convention that rank 1 is the best regardless of minimization or maximization.*

We will now give an example of these three definitions in the context of Example 2.

Example 3 (Example 2 in the context of Genetic Programming)**.** *A specific example of an individual would be a pair* (a, b)*, where* $a \in [1, 3]$ *and* $b \in [-3, -1]$*. An example of an individual would thus be* $(2, -2)$*. Each pair* (a, b) *would uniquely refer to a function* $g(x, a, b)$*, in the case of* $(2, -2)$ *it corresponds to the function* $(2, -2) = g(x, 2, -2) = 2x^2 - 2$*. In the referenced example we have not defined a concept which compares to a population. An example of a population in this specific setting would be a set* $\mathcal{P} = \{I_i\}_i^n$*, where each* I_i *is sampled uniformly from the Cartesian product* $[1, 3] \times [-3, -1]$*. The set of all individuals is given by* $\mathcal{I} = \{(a, b) | a \in [1, 3], b \in [-3, -1]\}$ *and the corresponding fitness function is given by the function h which corresponds to a Mean Squared Error. For this particular fitness follows that when it is minimized, i.e. equal to 0, the individual is more optimal compared to one with higher fitness. So in particular the individual (2,-2) has lower fitness and thus lower rank than the individual (2,-1).*

3.5.1 Representations

In Genetic Programming, the individuals/programs are often represented as trees. This is not only limited to a visual graph of a tree but also to the so-called syntax trees. Take for example the individual describing the function $f(x, y) = x^2y + y^2 + y$. We could represent this in a textual form by writing it in the following syntax $+(+(*(*xx)y)(*yy))y$ or we can just use $x^2y + y^2 + y$. Often also a graph of the tree is used which in this case would result in the following figure

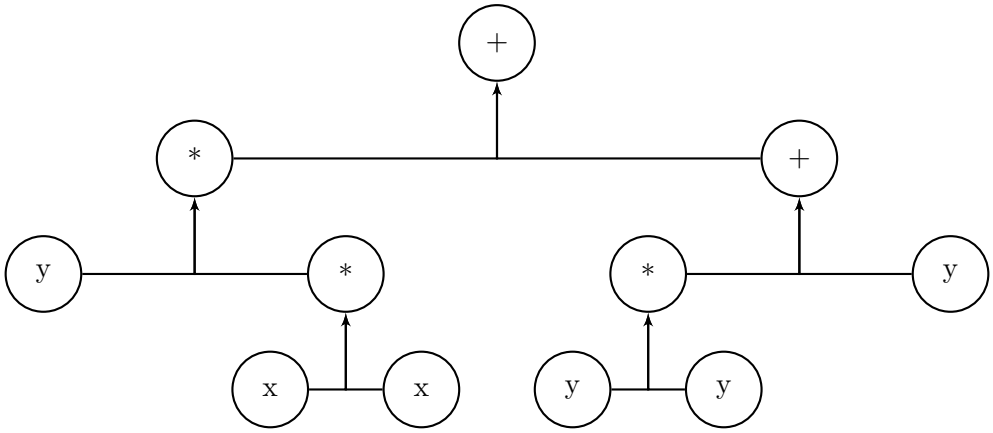


Figure 5: $f(x, y) = x^2y + y^2 + y$ in tree form.

Although the tree representation is mostly used, there are also other representations. Depending on the problem at hand researchers sometimes choose Linear Genetic Programming (LGP)[39]. Linear Genetic Programming is an example of a situation that would introduce different representations, see [39]. Going back to the function $f(x, y) = x^2y + y^2 + y$. The function is made out of building blocks. These building blocks come from two sets. Namely $\mathcal{T} = \{x, y\}$ and $\mathcal{F} = \{+, *\}$. The set $\mathcal{T}$ is called the terminal set and the elements terminals. The set $\mathcal{F}$ is the function set that contains functions as elements. These sets are called building blocks because they are all we need to create and describe individuals. The choice

for representation is often based on the programming language. As we will be using Python for this thesis we will be using the tree representation. In addition to this, we make use of the Python library DEAP[40]. This library will take care of the tree representation and will allow us to create our own algorithm which will be introduced in Section 4.

3.5.2 The initial population

Before we start the Genetic Evolutionary process we need an initial population. In most Evolutionary Algorithms this initial population is generated randomly. For Genetic Programming this is not different, although in some specific cases one might choose to use a specifically generated population but that is not the case in this thesis. In general, there is a lot of different approaches to generate the initial population. We will introduce, and make use of, three easy methods to generate the initial population. These three methods used are a full method, a grow method, and a combination of full and grow called Ramped half-and-half.

Definition 10 (Arity of a function)**.** *The arity of a function corresponds to the number of input variables of a function.*

Consider addition, it needs two values. Hence addition has arity 2. The logarithm operator needs only 1 value. Hence the logarithm operator has arity 1.

Definition 11 (Root)**.** *A root is the node in a graph that has no outgoing edges. Furthermore, the incoming degree needs to be equal to the arity of the function $f \in \mathcal{F}$ that is the label of the root. In the special case that the root is the only node in a graph, it will contain an element from $\mathcal{T}$.*

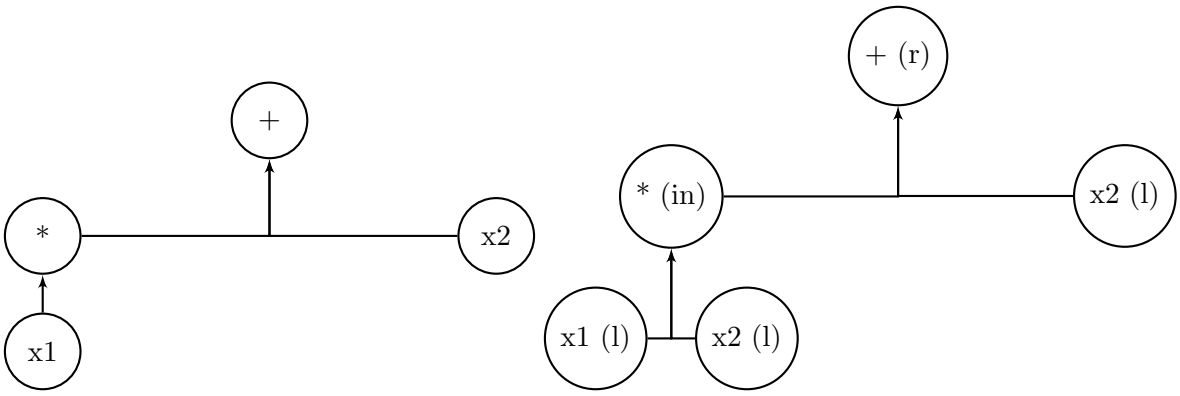
Definition 12 (Leaf)**.** *A leaf is a node in a graph that has no incoming edges. Leaf nodes will be labeled with elements from $\mathcal{T}$ and will always have degree 1. A leaf node will only have outgoing edges. A leaf node is a root if it is the only node in the graph.*

Definition 13 (Inner node)**.** *An inner node is neither a root nor a leaf. Inner nodes will be labeled with elements from $\mathcal{F}$. The degree of the node will be determined by the arity of the function $f \in \mathcal{F}$. The degree of an inner node will be $1 + n$, where n is the arity of f.*

Now that we have definitions for the building blocks of a tree we can define what a tree is

Definition 14 (Tree)**.** *A tree is a directed graph $G = (V, E)$ where the orientation is towards the root. The tree G consists of at least a root as defined in Definition 11. Furthermore, we say a tree is admissible if the inner nodes have correct degrees if it has any.*

The depth of a tree is often also referred to as length. These two terms will be used interchangeably. The following figure illustrates examples of trees that are admissible and inadmissible. For the remainder of this thesis, we will only consider the case where the trees are admissible. Namely in the case when they are admissible they represent functions that can be used as a mapping $f : \mathcal{X} \to \mathbb{R}$. Where $\mathcal{X}$ is the original feature space.



(a) Example of an inadmissible tree. The tree is inadmissible because the $*$ operator takes two inputs.

(b) Example of an admissible tree. Where (r) corresponds to the root node, (in) to the inner node and (l) to a leaf node.

Figure 6: Overview of two trees. One of which is inadmissible and one is admissible

Definition 15 (Node depth and Tree depth)**.** *The depth of a node is defined as the length of the path to the root. The depth of the tree is defined as the maximum over all the node depths. Hence a tree with only a root, which then automatically is also a leaf, has depth 0.*

There are two ways to create trees. These are called full and grow methods. The full and grow methods are similar in the sense that they both create admissible trees that fall in a certain depth range. For example, if we let the minimum depth be 1 and the maximum depth be 7, both methods will create trees that have a depth in the set $\{1, 2, \ldots, 7\}$. The full method is called 'full' because it creates only full trees. Examples of full trees is given in Figure 4. An admissible tree is full when all the leaves are at the same height (or depth). It creates the trees in the following way. It selects, uniformly at random, elements from the function set $\mathcal{F}$, and proceeds to repeat this until the maximum defined depth is reached. Then for the leaves it selects, uniformly at random, elements from the terminal set $\mathcal{T}$. Below follows the pseudo code for the full method, where for simplicity we assumed all functions to have arity 2.

Algorithm 1: Full method

```
Input: A depth d, function set F, terminal set T
initialize array A;
initialize tree T;
max_depth = d ;
if max_depth ≠ 1 then
    select node uniformly from F;
    add node to A;
    add node to T as root;
    while max_depth−1 is not reached do
        parent = A[0];
        remove A[0];
        nr_child = 0;
        while nr_child ≠ of parent do
            select child from F;
            add child to back of A;
            nr_child = nr_child + 1 ;
            add child to T connect child and parent with edge in T;
        end
    end
    while A ≠ ∅ do
        parent = A[0];
        remove A[0];
        nr_child = 0;
        while nr_child ≠ of parent do
            select child from T;
            nr_child = nr_child + 1 ;
            add child to T connect child and parent with edge in T;
        end
    end
else
    select node from T;
    add node to T as root
end
return T
```

From this method, we can see that it creates very static trees. This depends on the arity of the nodes in the function set, but as we will see, compared to the grow method the shapes of the full trees are rather limited. Now the grow method allows for the creation of fewer static trees. Grow trees do not have all leaves necessarily on the same depth. The grow method works by selecting nodes from the primitive set, which is $P := \mathcal{F} \cup \mathcal{T}$. Below follows the psuedo-code for the grow method.

Algorithm 2: Grow method

```
Input: A depth d, function set F, terminal set T
initialize array A;
initialize tree T;
max_depth = d ;
select node uniformly from P = F ∪ T;
add node to A;
add node to T as root;
while A ≠ ∅ and max_depth is not reached do
    parent = A[0];
    remove A[0];
    nr_child = 0;
    while nr_child ≠ of parent do
        select child from P;
        add child to back of A;
        nr_child = nr_child + 1 ;
        add child to T connect child and parent with edge in T;
    end
end
return T
```

In other words, it keeps selecting nodes until the maximum depth is reached at which only terminals are selected. A terminal may be selected before we reach the maximum depth. In this case, the creation of the tree is stopped, as terminals have arity 0. Hence a shorter tree is returned.

On their own the full and grow methods do not create a very different set of trees. A population created with only full trees will have similar-looking trees. The same holds for a population created with grow trees. We want to use both of them. This way we can ensure a variety of different looking trees. Koza [23] proposed a method that uses a combination of the two. This method is called Ramped half-and-half (RHH). Half of the population is created using the full method and the other half using the grow method. Ramped indicates that this is done for a wide range of depths. We vary the depths such that we obtain trees of all different kind of sizes. The depth is varied uniformly on a predefined interval. This interval is usually given by $[1, m]$ where m corresponds to the max depth. This max depth corresponds to the maximum size the created trees will have. The depth d which is used as input for the algorithm is thus varied and sampled from the interval $[1, m]$. Note that d is an integer. So we sample an integer uniformly from $[1, m]$. There is a small remark about using RHH, although it is very easy to implement, it is clear that the grow method is very much influenced by the proportion of terminals in the primitive set $\mathcal{P}$. If, for example, the fraction of terminals is 0.9 we would have a very high probability that the tree created with the grow method would be very short, regardless of the predetermined maximum depth. This could be a problem if one wants to have an equally distributed population with a high maximum depth. Below follows the pseudo-code for RHH.

Algorithm 3: Ramped-half-and-half

```
initialize array A;
set pop_size = p;
set max depth d;
set i = 0
while i < p do
    sample u_i from U ~ U(0,1);
    sample integer d_i from U ~ U(1,d);
    if u < 0.5 then
        create I_i with full method with depth d_i;
    else
        create I_i with grow method with depth d_i;
    end
    add I_i to A;
    i = i+1;
end
return initial population A
```

3.5.3 Possible trees

For this subsection, we will consider binary trees. This section is to illustrate the complexity of the Genetic Programming problem. We also show the sheer size from which the initial population introduced in Section 3.5.2 is sampled from. The assumption of only binary trees translates to the situation where all the functions in $\mathcal{F}$ have an arity of 2. That means that we consider only functions in the function set $\mathcal{F}$ that take two arguments. For example, the logarithm operator is not considered for this analysis. We let $\mathcal{T}$ again be the terminal set. Then how many binary trees are possible?

Proposition 1. *A binary tree with n inner vertices has n+1 leaves.*

Proof. This proof is done by induction. Let $n = 0$, then the statement in Proposition 1 is true. Namely, a tree without inner vertices has exactly 1 vertex which is automatically a leaf. So a tree T with 0 inner vertices has 1 leaf. Now we assume the statement to be true for all $n \leq k-1$ where $k \in \mathbb{N}$. This is the induction hypothesis. Now let T be a tree with $k > 0$ inner vertices. We will now show that T has $k+1$ leaves.
Since T is a tree with k inner vertices and since $k \geq 1$ we know that the root r has two sub-trees T_1 and T_2. Let k_1 and k_2 be the inner vertices of T_1 and T_2 respectively. Since every inner vertex in T is either a root or in T_1 or T_2 we conclude that T has $1 + k_1 + k_2$ inner vertices. Since $k_1 \leq k-1$ and $k_2 \leq k-1$, namely T has k inner vertices and the root is not in T_1 or T_2. It follows that we can use the induction hypothesis and conclude that T_1 and T_2 have $k_1 + 1$ and $k_2 + 1$ leaves respectively. Now since every leaf in T must be included in T_1 and T_2 we conclude that, with the fact that $k = 1 + k_1 + k_2$, that T has $k_1 + 1 + k_2 + 1 = k + 1$ leaves. □

Now using the Catalan numbers we can determine the amount of unlabeled binary trees. An unlabeled tree is a tree for which the inner nodes have not been filled with elements from $\mathcal{F}$. In addition to this, the leaves have not been filled with elements in $\mathcal{T}$. Hence unlabeled

trees are different purely by the structure of the nodes. A labeled tree is a tree for which the inner nodes and leaves have been labeled with their respective elements from $\mathcal{F}$ and $\mathcal{T}$.

Proposition 2. *Let T be a binary tree with n inner vertices and $n \in \mathbb{N}$, then the amount of different unlabeled binary trees that can be constructed is equal to*

$$C_n = \frac{1}{n+1}\binom{2n}{n}. \tag{2}$$

Let $L \geq 2$ be the length of tree T, we can write

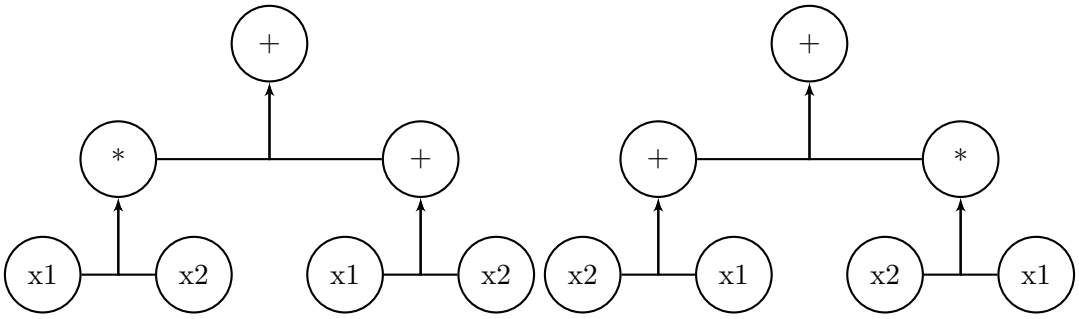
$$\bar{C}_L = \frac{2}{L-1}\binom{L-1}{\frac{L+1}{2}}, \tag{3}$$

where $\bar{C}_L$ gives the number of trees for a corresponding depth L (or length).

Proof. For 2 see Proposition 11 in [41]. For 3 it follows from an application of Proposition 1. Namely, the inner vertices n and total length of a binary tree L are described by the equation $L = 2n + 1$. □

Using Proposition 3 we now can determine how many different unlabeled trees there exists with a certain length. Take for example the tree length 7. We now know by Proposition 2 that there exist precisely 5 different binary trees of that length. Now we were only talking about unlabeled binary trees. To now calculate how many different trees there exist, note that this is not equal to the different functions the trees represent. Let $|\mathcal{T}|$ the set cardinality of the terminal set, idem for $|\mathcal{F}|$. Now for every inner vertex, we have a choice from $\mathcal{F}$ and for every leaf, we can take an element from $\mathcal{T}$. So for a tree with n inner vertices, we know that we have $n + 1$ leaves. So to label one possible binary tree with n inner vertices we have $|\mathcal{F}|^n \cdot |\mathcal{T}|^{n+1}$ options. In other words, the total possible labeled binary trees with n inner nodes are equal to $|\mathcal{F}|^n \cdot |\mathcal{T}|^{n+1} \cdot C_n$. In terms of length of the binary tree, we would obtain $|\mathcal{F}|^{(L-1)/2} \cdot |\mathcal{T}|^{(L+1)/2} \cdot \bar{C}_L$. So suppose that we would have that $\mathcal{F} = \{+, -, *, /\}$, so that $|\mathcal{F}| = 4$. Let $|\mathcal{T}| = 50$, suppose that we would like to check all functions up to $L = 11$, this would work out to be at least $6.72 \cdot 10^{14}$ trees. So assuming we could calculate the fitness of all these functions really fast, say $1\mu s$, this would still take years assuming we will not omit duplicates. As this is something to keep in mind, functions can be represented in different ways in tree form.

Remark 4. *What we mean by duplicate functions are the trees that reduce to the same function. Let $\mathcal{T} = \{x_1, x_2\}$, and $\mathcal{F} = \{*, +\}$. The following trees are equal based on the commutative properties of operators.*

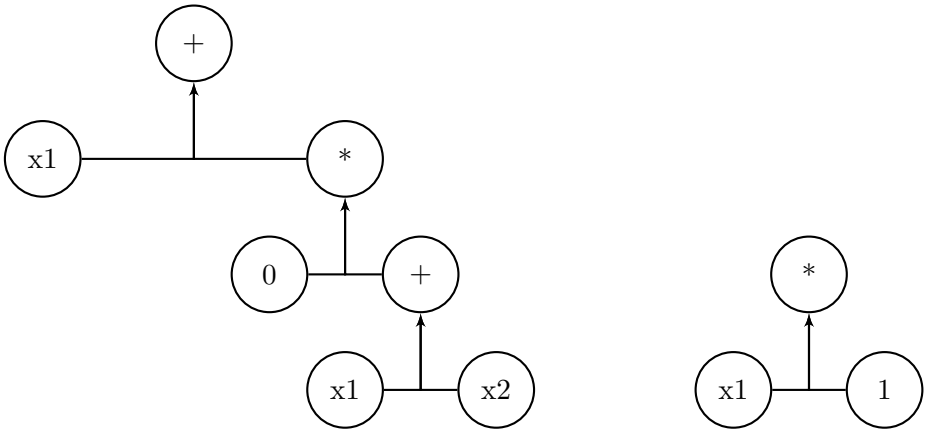


(a) A tree that represents the function $f(x) = x_1x_2 + (x_1 + x_2)$.

(b) A tree that represents the function $f(x) = x_1x_2 + (x_1 + x_2)$.

Figure 7: Two trees that represent the same function.

Indeed by making use of the commutativity of the addition and multiplication operator we have that both trees represent the same function. In addition to this, depending on the Terminal set $\mathcal{T}$ we can also have another way to represent different functions. For example assume that also $0 \in \mathcal{T}$ and $1 \in \mathcal{T}$, then we could have the following



(a) A tree that represents the function $f(x) = x_1$.

(b) A tree that represents the function $f(x) = x_1$.

Figure 8: Making use of $\mathcal{T}$ to create similar functions.

As we can see the addition of certain terminals also influences the different representations of trees, which in fact will lead to the same function. In Figure 8 we see that in sub figure (a) we have a tree that represents the function $f_a(x_1, x_2) = x_1 + (0 * (x_1 + x_2))$ which reduces to $f_a(x_1, x_2) = x_1$. Something similar happens with the multiplication in sub figure (b), which also reduces to $f_b(x_1, x_2) = x_1$.

3.5.4 Fitness functions

A crucial part of Genetic Programming is the choice of fitness function. The fitness function will be our indication for which individuals perform well in the evolution, and will thus help guide us to a good solution. An important observation, which will influence the fitness

functions considered, is that our functions need to be beneficial for the Classification And Regression Trees algorithm (CART). The Decision Tree algorithm tries, at every split, to split the data into two buckets. These buckets need to have a low average Gini impurity, or high Information Gain, see Definitions below. We will mostly consider the situation where the algorithm finds low average Gini impurity. Assume for now that the dataset is balanced in the sense that we have an equal proportion of class 0 and class 1 instances. If our individual, which is scored by the fitness function, achieves to separate the class 0 and class 1 instances this will lead to a low average Gini impurity. Hence we are searching for individuals which separate the class 0 and class 1 instances well. Therefore we will consider two different fitness functions. We note that for now, we assume to deal with binary classification problems, that is $\mathcal{Y} := \{0, 1\}$. Furthermore, we again let $\mathcal{X}$ be the sample space and $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ a training dataset, where the $x_i \in \mathcal{X}$, on which we perform GP.

Definition 16 (Fisher fitness)**.** *Assume we are in a Genetic Programming setting. That is we have a function set $\mathcal{F}$ and a terminal set $\mathcal{T}$. Now let I be an individual. Then given a training dataset $\mathcal{D}$ as defined in the paragraph above. We let $i \in \{0, 1\}$ and $n := |\mathcal{D}|$, first we define*

$$\mu_i(I) = \frac{1}{c_i} \sum_{j=1}^{n} I(x_j) \mathbf{1}\{y_j = i\},$$

where $c_i := \sum_{j=1}^{n} \mathbf{1}\{y_j = i\}$. This is exactly the average of I at points with label i. Furthermore, we define

$$\sigma_i(I) := \frac{1}{c_i - 1} \sum_{j=1}^{n} \left(I(x_j) - \mu_i(I)\right)^2 \mathbf{1}\{y_j = i\}.$$

Which is exactly the population variance of the values of I at points with label i. Given these quantities the Fisher fitness for an individual I is given by

$$f(I) := \frac{|\mu_0(I) - \mu_1(I)|}{\sqrt{\sigma_0(I)^2 + \sigma_1(I)^2}}.$$

The Fisher fitness is often also referred to as the Fisher criterion.

The Fisher fitness is based on Fisher's linear discriminant [42], which describes the separation between two distributions, in this case, the class 0 and class 1 instances. Besides the Fisher criterion, we will also consider a fitness based on the Gini impurity. This Gini impurity is often directly used in Decision Tree algorithms and hence we deem it a good candidate for possibly finding good individuals. First, we define the Gini impurity for a dataset $\mathcal{D}$

Definition 17 (Gini-impurity)**.** *Let $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a dataset as defined in the section above. Let $n := |\mathcal{D}|$ then we define*

$$p_i(\mathcal{D}) := \frac{1}{n} \sum_{j=1}^{n} \mathbf{1}\{y_j = i\}. \tag{4}$$

Which is the fraction of elements from $\mathcal{D}$ that have label i. Then the Gini impurity of dataset $\mathcal{D}$ is given by

$$G(\mathcal{D}) := 1 - \sum_i p_i \left(\mathcal{D}\right)^2,$$

which in the case of binary classification reduces to the convenient form $G(\mathcal{D}) = 2p_1(1-p_1)$. This is exactly twice the variance of a Bernoulli random variable with parameter p_1.

Definition 18 (Gini fitness)**.** *Let $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. We define*

$$I(\mathcal{D}) := \left[\ I(x_1), I(x_2), \cdots, I(x_n)\ \right],$$

which is the mapping generated by I on the instance values in the training dataset $\mathcal{D}$. We assume that $I(x_i) \leq I(x_{i+1})$ for all i, and thus that the values are sorted. Now we define a set of thresholds $\Theta_I := \{\theta_0, \ldots \theta_{n+1}\}$ such that $\theta_i \in (I(x_i), I(x_{i+1}))$, which is exactly the one-dimensional case of what we discussed in Section 2.2. Thus we notice $\theta_i \in \mathbb{R}$ for $i \in \{1, \ldots, n\}$. We use the convention that $\theta_0 = -\infty$ and $\theta_{n+1} = \infty$. Furthermore, let $n := |\mathcal{D}|$, and for $\theta \in \Theta_I$ we define the following

$$\mathcal{D}^\theta_<(I) := \{(I(x_i), y_i) \,|I(x_i) < \theta\},$$
$$\mathcal{D}^\theta_\geq(I) := \{(I(x_i), y_i) \,|I(x_i) \geq \theta\}.$$

Let $n_1 := \left|\mathcal{D}^\theta_<\right|$ and $n_2 := \left|\mathcal{D}^\theta_\geq\right|$. Then the Gini fitness of an individual I is given by

$$f(I) := \min_{\theta \in \Theta_I} \left(\frac{n_1}{n} G\left(\mathcal{D}^\theta_<(I)\right) + \frac{n_2}{n} G\left(\mathcal{D}^\theta_\geq(I)\right)\right).$$

We must note that we have defined both fitnesses in the case of binary classification but they generalize well to n-ary classification. Both fitness functions have their benefits. Which will result in an interesting discussion regarding which fitness we will be using in the remainder of the thesis.

Remark 5. *The fitness of a function will not be the only measure which we will use to determine if a function performs well on the data. Remember how we also want the functions (or extracted features) to be interpretable. For us, the interpretability of a function f is related to the number of nodes in the corresponding tree T_f. If a function f_1 and f_2 have the same fitness, but T_{f_1} has fewer nodes than T_{f_2}, we will prefer function f_1. This is exactly why we introduced the NSGA-II selection operator in Section 3.5.5, as this selection operator will let us minimize both the fitness and the number of nodes in the corresponding trees.*

3.5.5 Selection methods

The first step in Genetic Programming would be to initialize the population as we discussed in Section 3.5.2. The second step is scoring every individual with the fitness function, which we explained in Section 3.5.4. The next step in Genetic Programming (or Genetic Evolution in general) would be to select the individuals that we deem fit for reproduction methods like crossover and mutation. One can choose the best performing individual, and use this individual for reproduction. This would induce a strong selection pressure. Selection pressure

is a concept which describes the favouritism the selection method has for individuals. Strong selection pressure indicates that the selection method favours more fit individuals while a weak selection pressure does not discriminate much between lower and higher fitness individuals. The situation we described where we only take the fittest individual is an extreme case of strong selection pressure. A natural question to ask is what is the consequence of strong selection pressure? If we only choose the fittest individual from our initial population, it will be the only possible parent for reproduction. This will then lead to a rapid decrease in the diversity of children. Less diversity makes it harder to explore the search space and hence can lead to a local optimum. So ideally we want selection methods that do favour more fit individuals but not in a way that causes the diversity of the population to decrease extremely fast.

Definition 19 (Tournament selection)**.** *Let $k \in \mathcal{N}$ and $\mathcal{P}_k := \{I_1, \ldots, I_k\}$ where I_i are sampled with replacement from $\mathcal{P}$. Where $\mathcal{P}$ is a population. Thus I_i is a random individual selected from $\mathcal{P}$. Then the individual chosen by tournament selection, I_c, is given by*

$$I_c := \arg\max_{I \in \mathcal{P}_k} f(I),$$

where $f : \mathcal{P} \to \mathbb{R}$ is the corresponding fitness function.

As we can read from Definition 19 tournament selection first samples candidates, uniformly at random from the current population $\mathcal{P}$, then it proceeds to select 1 individual which is the best according to the fitness value of all sampled individuals. This random sampling, if k is small enough, ensures that mediocre individuals also have a chance at survival. This, compared to the selection of the best individual(s), has the effect that there is more diversity in the population after selecting candidates for reproduction. We expect that for $k > 1$ the selected subset population has a 'better' average fitness than the whole population, but not necessarily by much. Effectively this selection method rescales the fitness such that the selection pressure remains constant. It will not occur that a single best individual becomes the main resource of genetic material for the next generation, hence the reduction in loss of genetic diversity. The caveat of this method lies in the randomness of the selection method. This is inherent to noise, which causes the best individual to not necessarily survive or even reproduce. So on average, we expect the fitness to improve. But there might be a few generations that experience a setback in fitness gain. Tournament selection has been used often in the literature [43]. Another selection method that is often used in the literature is the following.

Definition 20 (Roulette selection/Proportionate fitness selection)**.** *Consider a population $\mathcal{P}$, where $n := |\mathcal{P}|$. Let $I_i \in \mathcal{P}$ be individuals where $i \in [n]$. Let F be the set of corresponding fitnesses where $f_i \in F$ corresponds to the fitness of individual I_i. We note F is not to be confused with $\mathcal{F}$ which is a set of function operators. Hence for fitness function $f(I_i) = f_i$. Consider the following random variable X with probability distribution given by*

$$\mathbb{P}(X = I_i) = \frac{f_i}{\sum_{j=1}^{n} f_j},$$

then in roulette selection the individuals are sampled from X.

Definition 19 and Definition 20 are great examples of how to select individuals for reproduction. They are stated to inform the reader about possible options one could use to select individuals for reproduction. In Section 5.2.1 we will discuss tournament selection in a bit more detail and compare it to only selecting the best individual. As we will see tournament selection performs well for obtaining better fitness. Unfortunately, it also has its drawbacks in the form of extracting very complex features, for this see Section 5.2.1. To overcome this we introduce the following selection method based on Non-Dominated Sorting Genetic Algorithm (NSGA-II) [44]. NSGA-II can group individuals based on multiple objectives, these objectives could be the fitness and length of the corresponding tree. Since we are trying to limit the size of the function and optimize the fitness, the number of objectives would be 2. Hence using a NSGA-II selection method we can select individuals based on both objectives which would help us to optimize for both. The NSGA-II algorithm assigns individuals to groups called fronts, where individuals in front 1 are regarded as the best. Inside these fronts, individuals are ranked according to a crowding distance. For more information, we refer to [44] which has an excellent explanation of the algorithm. All we need to explain the selection using NSGA-II is that we have several fronts $F_1, \cdots, F_n$, and inside each front F_i we have k_j individuals ranked in order where 1 is the best. These fronts are all that are returned by the NSGA-II algorithm, again how the algorithm works exactly can be read in [44]. We create the following ordering of individuals

$$O = \{I_1^{F_1}, \cdots, I_{k_1}^{F_1}, I_1^{F_2}, \cdots, I_{k_n}^{F_n}\},$$

where $I_j^{F_i}$ is the $j-th$ ranked individual in front F_i. Now to select ξ individuals using the NSGA-II algorithm we will select the first ξ individuals from O. This selection method will be our preferred one because it allows us to optimize for two (or more) objectives. Hence it will be used in the algorithm introduced in Section 4.1.

3.5.6 Genetic operators

We now know how to select individuals for reproduction. But how do we perform these genetic reproductions? For this, we use genetic operators. In Genetic Programming and Genetic Algorithms there are various ways to define genetic operators. The most common operators are crossover and mutation but stating that you use crossover or mutation is not precise enough. There are namely a lot of different ways to define crossover and mutations themselves. In this section, we will give few examples of how to define crossover and mutation operators. We start with crossover and its most used variant, subtree crossover. But before that, we give a small remark,

Remark 6. *Note that all the operators shown in this section are merely only applicable to the Genetic Programming situation where a tree representation is chosen. It might be however that they are indeed applicable to a more general situation with small changes in the definition or pseudo code.*

In subtree crossover, two parents are selected as candidates to produce offspring. Parents are represented as individuals in a population $\mathcal{P}$. Then for both parents, a node is chosen. This process is done independently from both parents so not necessarily the same node is chosen for the parents. When the node is chosen for each parent subtree crossover creates

a new individual by swapping the subtrees of both nodes. Then randomly, one of the new individuals, as there are two, is chosen to be selected as offspring. However it is possible to define a version of subtree crossover where two individuals are returned, but this is not done commonly. As we have seen in Proposition 1, there are more leaves than inner vertices. This would lead to a problem when we use uniform selection of the nodes where the subtree will be swapped. Namely roughly 50% of the time we would select a leaf. So a lot of our crossovers will just be swapping leaves, to overcome this obstacle Koza [23] defined an improved version of subtree crossover which has a 0.9 probability of selecting a function node or inner vertex, and 0.1 probability of selecting a leaf. The is improved variant of subtree crossover is still widely popular in Genetic Programming [45] and will also be used in this thesis' results. From now on the improved subtree crossover by Koza with the 0.9/0.1 probabilities will be referred to as standard crossover. An example of subtree crossover is given in Figure 9.

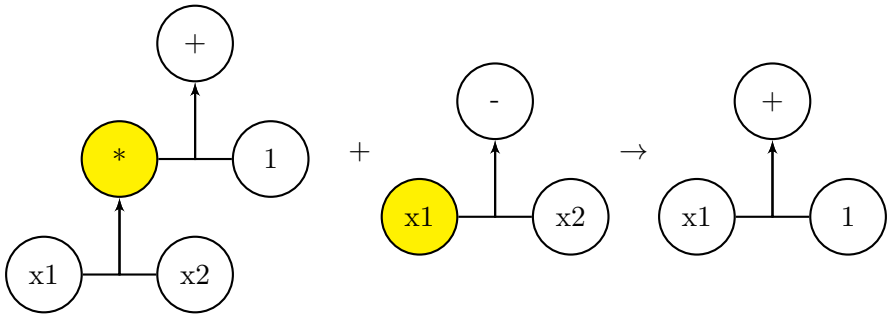


Figure 9: Example of sub-tree crossover happening on the yellow nodes.

Remark 7. *We want to elaborate on the fact that in Figure 9 we have made use of convention regarding subtree crossover. As we mentioned subtree crossover is often defined to return only one new individual. The convention we used is that the modification of parent 1 is returned, in the case of the Figure. This is exactly the individual that is right of the arrow. The other individual which is created shown in the figure below is discarded.*

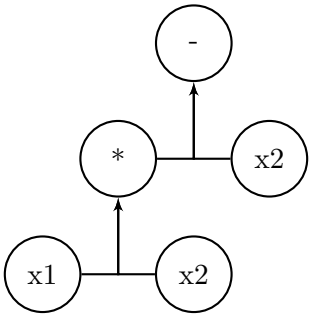


Figure 10: The individual that got discarded.

As we have already mentioned there are many different ways to perform mutation in Genetic Programming. We will shed a light on the most often used ways to define the concept of

mutation. The most commonly used form of mutation is what is often referred to as 'subtree mutation'. In subtree mutation a random node is selected, from an individual I, as the point of mutation, we call this node N. Then the subtree starting at node N is replaced by a completely new and randomly generated subtree. This is shown in Figure 11. Sometimes subtree mutation is viewed in the context of crossover, where the crossover acts on node N in individual I and the newly generated subtree. An example is given in Figure 12.
Another form of mutation is the so-called one-point mutation, which is equivalently the Genetic Programming variant of the bit-flip mutation used in Genetic Algorithms. In one-point-mutation a random node N is selected in the tree, and replaced by a node, i.e. function or terminal, with the same arity. It is replaced by a node with the same arity to preserve the admissibility of the tree. If no candidate exists to replace the selected node with, nothing will happen and the individual will stay the same.

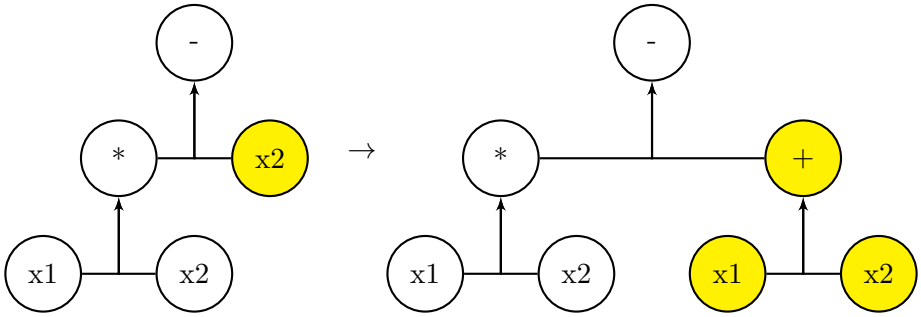


Figure 11: An example of subtree mutation.

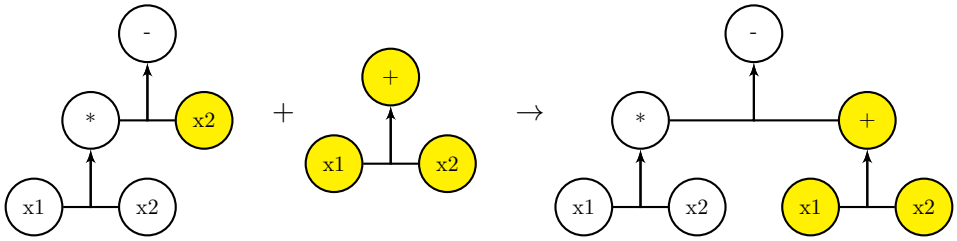


Figure 12: Subtree mutation viewed as a crossover, where the second parent is the randomly generated individual.

In general, genetic operators are considered to be mutually exclusive. That is, we consider only one of the operators to happen to an individual. So it is for example not possible for an individual to be first affected by mutation and later on be used in a crossover. The only way this would be possible is if the mutated individual survives to the next generation and is then used in a crossover. The corresponding probabilities crossover rate and mutation rate indicate how often each operator will be chosen. In the case where the crossover rate, c_r, and mutation rate m_r do not add up to 1. There is a probability of $1 - c_r - m_r$ that none of the operators will be chosen and the operator that will be applied will be reproduction. Reproduction is nothing else than just applying some identity operator on an individual. What we mean by this is that the individual gets mapped on itself, hence nothing happens to its genetics (or

tree nodes).

3.6 Algorithms & Evolution strategies

Now that we have all the basics down, we will talk about the strategies that can be used to perform evolution. In this section, we will formally introduce the basic pseudo-code of a genetic algorithm. Using this basic pseudo-code we will state Evolutionary Strategies that are often used in GP.

To use Genetic Programming we often also have to speak about Genetic Algorithms. The two are very similar, whereas their biggest differences occur in the representation choices. Genetic Algorithms often have a fixed length and are represented using a fixed length of bits. Genetic Programming can take on various representations as we introduced earlier. Often the genetic operators are influenced by these representation choices. For example, the subtree crossover only makes sense in the domain of tree representations and is therefore not applicable to the fixed-length bits representation that is often used in Genetic Algorithms. The same principle holds for the mutation operators and other genetic operators. That is not to say that we cannot use similar algorithms for both. If in these algorithms we, for example, only refer to crossover and mutation in the sense that they have to be applied it does not matter how these operators are applied. If we keep these algorithms vague enough, to the point it works for various representations. These Genetic Algorithms can then be applied in the context of Genetic Programming. The basic algorithm in pseudo-code for Genetic Algorithms can be found below. First of all, let $\mathcal{P}(t)$ denote the population at time point t. So that $\mathcal{P}(0)$ is the initial population.

Algorithm 4: Basis of Gentic Algortihm

set $t = 0$ choose fitness function f;
set $NGEN = n \in \mathbb{N}$;
randomly create $\mathcal{P}(t)$ using initialization operator;
assign fitness to every individual in $\mathcal{P}(t)$ by using f;
while $t < n$ **do**
    t = t+1;
    select individuals from $\mathcal{P}(t-1)$ and place them in $\mathcal{P}(t)$;
    apply genetic operators on the individuals in $\mathcal{P}(t)$;
    assign fitness to every individual in $\mathcal{P}(t)$ by using f;
end
return $\mathcal{P}(t)$ *and best individual found during the evolution;*

In the context of DEAP, Algorithm 4 corresponds closely to the implementation of the algorithm 'eaSimple'. As stated in the pseudo-code an initial population is created and scored. Then a 1:1 selection is made by the selection operator, and the selected individuals are placed in the next Population $\mathcal{P}(t)$. Then genetic operators like crossover, mutation, and reproduction are applied to all individuals in population $\mathcal{P}(t)$. Various variations from this basic genetic algorithm have occurred. The two main variations we will consider are the so-called $(\mu + \lambda)$ variant and the (μ, λ) variant. These variants have been extensively used in the literature [46].

Algorithm 5: $(\mu + \lambda)$-algorithm

set $t = 0$ choose fitness function f;
set $NGEN = n \in \mathbb{N}$;
randomly create $\mathcal{P}(t)$ using initialization operator;
assign fitness to every individual in $\mathcal{P}(t)$ by using f;
while $t < n$ **do**
    t = t+1;
    use genetic operators to create λ individuals from $\mathcal{P}(t-1)$ and place them in $\mathcal{P}'(t-1)$;
    assign fitness to every individual in $\mathcal{P}'(t-1)$ by using f;
    $\mathcal{T}(t) = \mathcal{P}(t-1) + \mathcal{P}'(t-1)$;
    select μ individuals using the selection operator from $\mathcal{T}(t)$ and place them in $\mathcal{P}(t)$;
end
return $\mathcal{P}(t)$ *and best individual found during the evolution;*

Algorithm 6: (μ, λ)-algorithm

set $t = 0$ choose fitness function f;
set $NGEN = n \in \mathbb{N}$;
randomly create $\mathcal{P}(t)$ using initialization operator;
assign fitness to every individual in $\mathcal{P}(t)$ by using f;
while $t < n$ **do**
    t = t+1;
    use genetic operators to create λ individuals from $\mathcal{P}(t-1)$ and place them in $\mathcal{P}'(t-1)$;
    assign fitness to every individual in $\mathcal{P}'(t-1)$ by using f;
    select μ individuals using the selection operator from $\mathcal{P}'(t-1)$ and place them in $\mathcal{P}(t)$;
end
return $\mathcal{P}(t)$ *and best individual found during the evolution;*

As we can read from the pseudo-code, the $(\mu + \lambda)$ and (μ, λ) algorithms are extensions of the basis genetic algorithm. It is an extension in the sense that we allow for different amounts of created children. Also, we can influence the selected amount of individuals that move on to the next generation. The biggest difference between $(\mu + \lambda)$ and (μ, λ) is found in the selection part. In $(\mu + \lambda)$ selection is applied on $(\mu + \lambda)$ individuals, where we select exactly μ individuals and there is a connection with the last generation in the sense that μ individuals are from the previous generation. In (μ, λ) selection is applied on λ individuals where these λ individuals are created from μ individuals in the previous generation. So with respect to the last generation, it is not necessarily a subset of the temporary population on which selection is applied, which is the case in the $(\mu + \lambda)$ algorithm. The consequence of this subtle difference is that the $(\mu + \lambda)$ leans more to the convergent side, whereas (μ, λ) is more exploratory. This means that for $(\mu + \lambda)$, the average fitness of the population will gradually go down until a local optimum is reached, whereas (μ, λ)'s average fitness of the population is more fluctuating. Since (μ, λ) is more fluctuating in fitness we will be opting to use the $(\mu + \lambda)$ algorithm in the remainder of the thesis. One other consequence of the $(\mu + \lambda)$ is that it

can be dominated by a single individual. If one individual that performs exceptionally well appears in the population, it will likely stay there. This also depends on the chosen selection operator.

4 The algorithm and its pitfalls

In this section, we will introduce the GP algorithm. This algorithm will make use of the fundamentals discussed in Section 3.5 and the $(\mu + \lambda)$ algorithm introduced in Section 3.6. Furthermore, we will elaborate on why we choose Gini fitness as our preferred fitness function.

4.1 The algorithm

In this subsection, we will introduce the algorithm in further detail. We must note that the algorithm, i.e. the algorithm to create new features, depends on multiple concepts. We will first describe the algorithm in words. First, we start with a dataset $\mathcal{D}$ for which we want to create data-driven features. We then proceed with applying GP on the dataset $\mathcal{D}$, this will result in a function f_0. The function f_0, which is a mapping from $\mathcal{X} \to \mathbb{R}$ can be applied on the dataset $\mathcal{D}$. For which we assume that $\mathcal{D} = \{(x_0, y_0), \ldots, (x_n, y_n)\}$ We will denote

$$f_0(\mathcal{D}) := \begin{bmatrix} f_0(x_0) \\ \vdots \\ f_0(x_n) \end{bmatrix}.$$

Then accompanied with the function f_0 we also know its fitness using Gini, as it was obtained by the GP algorithm. Hence we know a $\hat{\theta}$ such that

$$\frac{n_1}{n} G\left(\mathcal{D}_<^{\hat{\theta}}(f_0)\right) + \frac{n_2}{n} G\left(\mathcal{D}_\geq^{\hat{\theta}}(f_0)\right),$$

is minimized. Where n_1 and n_2 are the same constants as discussed in Section 3.5.4. Now we also have obtained two new datasets. Namely $\mathcal{D}_<^{\hat{\theta}}$ and $\mathcal{D}_\geq^{\hat{\theta}}$, rename these datasets to be respectively $\mathcal{D}_1$ and $\mathcal{D}_2$. Then we continue applying GP on these datasets for $\mathcal{D}_i$ where $G(\mathcal{D}_i) \neq 0$. In pseudo-code:

Algorithm 7: Feature Creation

Input: $\mathcal{D}$, GP algorithm GP
initialize an array A;
initialize an array B;
set $\mathcal{D}_0 = \mathcal{D}$;
set $i = 0$;
for *dataset* $= \mathcal{D}$ *in* A **do**
  if $G(\mathcal{D}) \neq 0$ **then**
    remove $\mathcal{D}$ from A;
    apply GP on $\mathcal{D}$ and get function f_i;
    add f_i to B;
    get $\hat{\theta}$ for which fitness of f_i is minimal;
    get $\mathcal{D}^{\hat{\theta}}_{<}(f_i)$ and $\mathcal{D}^{\hat{\theta}}_{\leq}(f_i)$;
    rename $\mathcal{D}^{\hat{\theta}}_{<}(f_i)$ and $\mathcal{D}^{\hat{\theta}}_{\leq}(f_i)$ into $\mathcal{D}_{2i+1}$ and $\mathcal{D}_{2i+2}$ respectively;
    add $\mathcal{D}_{2i+1}$ and $\mathcal{D}_{2i+2}$ to A;
  else
    remove $\mathcal{D}$ from A;
  end
  i = i+1;
end
return *B;*

There are some subtleties in the pseudo-code of Algorithm 7. First of all, we take as input a GP algorithm. Which could also have been called an Evolutionary Strategy. We require this strategy to return a function f, just like Algorithm 5 and Algorithm 6. Furthermore, we implicitly assume that the fitness used is the Gini fitness, which was discussed in Section 3.5.4. The reason why we assume this will be discussed in Section 4.2. Since the GP algorithm is of such big importance in Algorithm 7 the rest of this subsection will be devoted to explaining in detail what the operators are that occur in Algorithm 5 and Algorithm 6. As already mentioned we assume the fitness to be the Gini fitness. Secondly, the operators that need to be defined are

- Initialization operator.
- Genetic operators.
- Selection operator.

For the initialization operator, we will use the Ramped half-and-half method discussed in Section 3.5.2 Algorithm 3. For the genetic operators, we will use the subtree crossover and subtree mutation described in Section 3.5.6. Our selection operator of choice will be the NSGA-II selection method as it will allow us to use multi-objective optimization. Hence we can optimize for the Gini fitness and the length of the individual. Now that we have clarified exactly which operators we will be using in our GP algorithm we must specify what our GP algorithm will look like. It turns out that our GP algorithm (or Evolutionary Strategy) is very similar to the $(\mu + \lambda)$ and (μ, λ) strategies discussed in Section 3.6. Below we give the

pseudo-code with a little more detail compared to Algorithm 5, that is since we specifically know the operators we can also specifically denote what we require as input to the algorithm.

Algorithm 8: GP algorithm

Input: $\mathcal{D}$, max_gen, initial_pop_size, μ, λ, cxpb, mutpb
set $t = 0$
randomly create $\mathcal{P}(t)$ using RHH of size initial_pop_size;
assign fitness to every individual in $\mathcal{P}(t)$ by using the Gini fitness on $\mathcal{D}$;
while $t <$ *max_gen* **do**
    t = t+1;
    for $1 \rightarrow \lambda$ **do**
        sample u from $U \sim \mathcal{U}(0, 1)$;
        if $u \leq$ *cxpb* **then**
            create individual I using subtree crossover;
            add I to $\mathcal{P}'(t-1)$;
        end
        else if $cxpb < u \leq cxpb + mutpb$ **then**
            create individual I using subtree mutation;
            add I to $\mathcal{P}'(t-1)$;
        end
    end
    assign fitness to every individual in $\mathcal{P}'(t-1)$ by using f;
    $\mathcal{T}(t) = \mathcal{P}(t-1) + \mathcal{P}'(t-1)$;
    select μ individuals using NSGA-II from $\mathcal{T}(t)$ and place them in $\mathcal{P}(t)$;
end
return $\mathcal{P}(t)$ *and best individual found during the evolution;*

The pseudo-code above also describes the procedures used to how we exactly create the functions based on a dataset $\mathcal{D}$. It must be noted that Algorithm 8 can be easily changed into a version that makes use of the (μ, λ) Algorithm. In the following remark, we dive a little deeper into the small changes that can be made to Algorithms 7 and Algorithm 8.

Remark 8. *Note how in Algorithm 7 the variable i is used to labelling the datasets. In addition to labelling the datasets obtained by applying the split of the function f_i it can also be used to track how many splits we have made. Hence this variable can also be used as an early stopping threshold, say we only want to extract 5 functions, then we can limit the process by moving the whole for loop into a while loop, conditioned on the fact that $i < 5$. But similarly, we can also influence the datasets $\mathcal{D}$ on which GP is applied. One example would be to only look at datasets $\mathcal{D}$ for which we have that $|\mathcal{D}| > k$, where $k \in \mathbb{N}$. But these types of early stopping are not only applicable to Algorithm 7. In fact, for Algorithm 8 we can also define an early stopping threshold. In this thesis, we choose to stop GP when the best individual that has occurred in the evolution has not been changed for $k \in \mathbb{N}$ generations.*

In the remainder of the thesis, we will often refer to a standard GP algorithm, this standard version of the algorithm will be defined based on the following standard values for the parameters.

parameter	value
max_gen	1000
initial_pop_size	1000
μ	100
λ	200
cxpb	0.9
mutpb	0.1
early_stopping	100
#_functions	1

Table 1: Overview of what we refer to as the standard parameters for our GP algorithm.

Unless stated otherwise the set $\mathcal{F} = \{+, -, /, *\}$ and $\mathcal{T}$ will be the set with all the input variables as well extended with the set $C = \{-1, 0, 1\}$ so that $\mathcal{T} = \{x_1, \cdots, x_n, -1, 0, 1\}$. We stress that the parameter 'early_stopping' refers to the number of generations after which the evolution stops if we have not found a better individual. Furthermore, the '#_functions' refers to how many splits we will do on a dataset $\mathcal{D}$. These base values for the parameters were chosen from a trial and error perspective. The max_gen parameter is based on a run time perspective. Often running for a maximum of 1000 generations will never be reached as the evolution has already converged to an optimum it cannot easily escape. The initial population size is subject to change, it does not impact the total run time of the algorithm, and the impact on the results is negligible, which will together be explained with the choices for μ and λ in Section 5.2.2. The crossover and mutation probabilities are chosen to be 0.9 and 0.1 respectively. For most of the research discussed in Section 1.1 the probabilities for crossover ranged from 0.7-0.9 and mutation from 0.1-0.3. Based on this we decided to go with 0.9 and 0.1, but they can be changed according to preferences.

Remark 9 (Dealing with imbalanced data)**.** *It might be the case that dataset $\mathcal{D}$ contains heavily imbalanced data. A dataset is imbalanced if one of the two classes occurs far more often than the other. A great example of an imbalanced data problem is the question of whether or not mail is spam. This is because only a fraction of the mail is often spam. To deal with imbalanced data we will use weights for certain labels. The weights given to each label are inversely proportional to the class frequencies in the data. That is, let $|\mathcal{D}| = N$ and let the number of classes be k. Then the weight w_i of label y_i will be defined as*

$$w_i = \frac{N}{k \cdot \sum_{j=1}^{N} \mathbf{1}\{y_j = i\}}.$$

These weights will then be used in evaluating the Gini-fitness of the individuals, which means that in 17 Equation 4 will be changed in

$$p_i(\mathcal{D}) := \frac{1}{n} \sum_{j=1}^{n} w_i \mathbf{1}\{y_j = i\}.$$

Note that these weights are calculated at the beginning of the GP algorithm and will then not be re-evaluated for the rest of the algorithm, it is purely a way to weigh the importance of certain classes.

For the remainder of the thesis, the feature creation algorithm in Algorithm 7 will be referred to as the GP algorithm. Where the standard GP algorithm will make use of the parameters defined in Table 1.

4.2 Fisher versus Gini

One very important aspect of the algorithm is the fitness function we choose to use during the evolution. We are focusing on data-driven options, the so-called filter approaches, opposed to the wrapper approaches we have seen in related works. The decision to choose a fitness will be based on the following observation. We will show that the Fisher fitness is more prone to overfitting, which is why we will prefer to use the Gini fitness. This is the last step before we can use Algorithm 7 and apply it in Section 5.

Suppose that we have an individual I that perfectly splits the data. With this, we mean that all the class 0 instances belong to the same split, and all the class 1 instances belong to the other split. This is equivalent with assuming that there exist $\theta_0, \theta_1, \varepsilon_0, \varepsilon_1 \in \mathbb{R}$ such that $I : \mathcal{D} \to \mathbb{R}$ maps all class 0 instances to an interval $I_0 = [\theta_0 - \varepsilon_0, \theta_0 + \varepsilon_0]$ and similarly for the class 1 instances to $I_1 = [\theta_1 - \varepsilon_1, \theta_1 + \varepsilon_1]$. Furthermore, we assume without loss of generality that $\theta_0 < \theta_1$ and state the obvious consequence that $[\theta_0 - \varepsilon_0, \theta_0 + \varepsilon_0] \cap [\theta_1 - \varepsilon_1, \theta_1 + \varepsilon_1] = \emptyset$. Then from Definition 16 we obtain that for a given dataset $\mathcal{D}$, where for simplicity we also assume that $\mathcal{D}$ contains no duplicates, that we have that the Fisher fitness of the function is given by

$$f(I) = \frac{|\mu_0(I) - \mu_1(I)|}{\sqrt{\sigma_0(I)^2 + \sigma_1(I)^2}}, \tag{5}$$

where the constants are exactly as in Definition 16. Since we have that all class instances are mapped in their respective interval we can choose ε_0 and ε_1 such that additionally

$$\begin{aligned} \exists j : I(x_j) &= \theta_0 + \varepsilon_0, \\ I(x_{j+1}) &= \theta_1 - \varepsilon_1, \end{aligned} \tag{6}$$

This shows that for $\hat{\theta} \in (\theta_0 + \varepsilon_0, \theta_1 - \varepsilon_1)$ we have that, using Definition 18,

$$G\left(\mathcal{D}^{\hat{\theta}}_{<}(I)\right) = G\left(\mathcal{D}^{\hat{\theta}}_{\geq}(I)\right) = 0.$$

This follows directly from the fact that in both $p_1\left(\mathcal{D}^{\hat{\theta}}_{<}(I)\right)$ is either 0 or 1 because of the perfect split. The same follows for $p_1\left(\mathcal{D}^{\hat{\theta}}_{\geq}(I)\right)$. Now since the Gini fitness is bounded from below by 0, we have that $\hat{\theta}$ minimizes the Gini fitness, and hence $f_g(I) = 0$. Let us define $\mu_i := \mu_i(I)$ for $i \in \{0, 1\}$. We will now define a new individual I' which will have the same μ_0 and μ_1 as I but a smaller σ_0 and σ_1. This will prove useful later on in our analysis. We proceed by defining the new individual I', which is given by

$$I'(x) = \left(\frac{1}{\sqrt{2}}\left(I(x) - \mu_0\right) + \mu_0\right)\mathbf{1}\{I(x) \in I_0\} + \left(\frac{1}{\sqrt{2}}\left(I(x) - \mu_1\right) + \mu_1\right)\mathbf{1}\{I(x) \in I_1\}.$$

We will now calculate $\mu_0(I')$ and show that is equal to μ_0 for the given dataset $\mathcal{D}$. It follows that

$$\begin{aligned}
\mu_0(I') &= \frac{1}{c_0}\sum_{j=1}^{n} I'(x_j)\mathbf{1}\{y_j = 0\} \\
&= \frac{1}{c_0}\sum_{j=1}^{n}\left(\frac{1}{\sqrt{2}}\left(I(x_j) - \mu_0\right) + \mu_0\right)\mathbf{1}\{y_j = 0\} \\
&= \frac{1}{c_0}\sum_{j=1}^{n}\frac{1}{\sqrt{2}}\left(I(x_j) - \mu_0\right)\mathbf{1}\{y_j = 0\} + \frac{1}{c_0}\sum_{j=1}^{n}\mu_0\mathbf{1}\{y_j = 0\} \\
&= \frac{1}{c_0\sqrt{2}}\sum_{j=1}^{n} I(x_j)\mathbf{1}\{y_j = 0\} - \frac{\mu_0}{c_0\sqrt{2}}\sum_{j=1}^{n}\mathbf{1}\{y_j = 0\} + \frac{\mu_0}{c_0}\sum_{j=1}^{n}\mathbf{1}\{y_j = 0\} \\
&= \frac{\mu_0}{\sqrt{2}} - \frac{\mu_0}{\sqrt{2}} + \mu_0 \\
&= \mu_0.
\end{aligned} \tag{7}$$

Where we made use of Definition 16 and the fact that

$$\begin{aligned}
&\mathbf{1}\{I(x) \in I_0\}\mathbf{1}\{y_j = 0\} = \mathbf{1}\{y_j = 0\}, \\
&\mathbf{1}\{I(x) \in I_1\}\mathbf{1}\{y_j = 0\} = 0.
\end{aligned}$$

This follows from the assumption that I perfectly separates the labeled instances. Similar computations wills how that $\mu_1(I') = \mu_1$. Now we proceed with calculating $\sigma_0(I')$, let again $\sigma_i := \sigma_i(I)$ then it follows that

$$\begin{aligned}
\sigma_0\left(I'\right) &= \frac{1}{c_0 - 1}\sum_{j=1}^{n}\left(I'(x_j) - \mu_0(I')\right)^2\mathbf{1}\left\{y_j = 0\right\} \\
&= \frac{1}{c_0 - 1}\sum_{j=1}^{n}\left(I'(x_j) - \mu_0\right)^2\mathbf{1}\left\{y_j = 0\right\} \\
&= \frac{1}{c_0 - 1}\sum_{j=1}^{n}\left(I'(x_j)^2 + \mu_0^2 - 2\mu_0 I'(x_j)\right)\mathbf{1}\left\{y_j = 0\right\} \\
&= \frac{1}{c_0 - 1}\left(\sum_{j=1}^{n} I'(x_j)^2\mathbf{1}\left\{y_j = 0\right\} + \frac{\mu_0^2}{c_0 - 1}\sum_{j=1}^{n}\mathbf{1}\left\{y_j = 0\right\} - \frac{2\mu_0}{c_0 - 1}\sum_{j=1}^{n} I'(x_j)\mathbf{1}\left\{y_j = 0\right\}\right).
\end{aligned} \tag{8}$$

Where again the parts containing $\mathbf{1}\{y_j = 1\}$ vanish similarly as before. Using the result from (7) and the quantities in Definition 16 we have that

$$\begin{aligned}
\frac{\mu_0^2}{c_0 - 1}\sum_{j=1}^{n}\mathbf{1}\left\{y_j = 0\right\} &= \frac{\mu_0^2 c_0}{c_0 - 1}, \\
\frac{2\mu_0}{c_0 - 1}\sum_{j=1}^{n} I'(x_j)\mathbf{1}\left\{y_j = 0\right\} &= \frac{2\mu_0^2 c_0}{c_0 - 1},
\end{aligned} \tag{9}$$

Furthermore, we have that

$$\frac{1}{c_0-1}\sum_{j=1}^{n} I'(x_j)^2 \mathbf{1}\{y_j=0\} = \frac{1}{c_0-1}\sum_{j=1}^{n}\left(\frac{1}{\sqrt{2}}\left(I(x_j)-\mu_0\right)+\mu_0\right)^2 \mathbf{1}\{y_j=0\}$$
$$= \frac{1}{c_0-1}\sum_{j=1}^{n}\left(\frac{1}{2}\left(I(x_j)-\mu_0\right)^2+\mu_0^2+\sqrt{2}\mu_0\left(I(x_j)-\mu_0\right)\right)\mathbf{1}\{y_j=0\}.$$

Splitting the terms up gives us that

$$\frac{1}{c_0-1}\sum_{j=1}^{n}\frac{1}{2}\left(I(x_j)-\mu_0\right)^2\mathbf{1}\{y_j=0\} = \frac{1}{2}\sigma_0,$$
$$\frac{1}{c_0-1}\sum_{j=1}^{n}\mu_0^2\mathbf{1}\{y_j=0\} = \frac{c_0\mu_0^2}{c_0-1}, \tag{10}$$
$$\frac{1}{c_0-1}\sum_{j=1}^{n}\sqrt{2}\mu_0\left(I(x_j)-\mu_0\right)\mathbf{1}\{y_j=0\} = 0.$$

Where the last equality follows from the fact that

$$\frac{1}{c_0-1}\sum_{j=1}^{n}\sqrt{2}\mu_0\left(I(x_j)-\mu_0\right)\mathbf{1}\{y_j=0\} = \frac{\sqrt{2}\mu_0}{c_0-1}\sum_{j=1}^{n}I(x_j)\mathbf{1}\{y_j=0\} - \frac{\sqrt{2}\mu_0^2}{c_0-1}\sum_{j=1}^{n}\mathbf{1}\{y_j=0\}$$
$$= \frac{\sqrt{2}\mu_0^2 c_0}{c_0-1} - \frac{\sqrt{2}\mu_0^2 c_0}{c_0-1}$$
$$= 0,$$

where we used Definition 16 and our definition of μ_0. Now using the results from (9) and (10) in (8) we conclude that

$$\sigma_0(I') = \frac{1}{2}\sigma_0.$$

Now following similar calculations, we can show that $\sigma_1(I') = \frac{1}{2}\sigma_1^2$. Hence we have shown that for the individual I' we have achieved a mapping that still has the same mean for the class 0 and class 1 instances, but a smaller variance or standard deviation for both class instances. Since both class instances still fall in their respective intervals. Namely for class 0 instances only we first note that we must have by construction

$$\theta_0-\varepsilon_0 \le \mu_0 \le \theta_0+\varepsilon_0. \tag{11}$$

Furthermore, this observation leads to the following equation which holds only if $I(x) \in I_0$, for readability we omit the indicator

$$\theta_0-\varepsilon_0 \le I(x) \le \theta_0+\varepsilon_0,$$

which is by elementary operations equal to

$$\frac{1}{\sqrt{2}}\left(\theta_0-\varepsilon_0\right)+\left(1-\frac{1}{\sqrt{2}}\right)\mu_0 \le I'(x) \le \frac{1}{\sqrt{2}}\left(\theta_0+\varepsilon_0\right)+\left(1-\frac{1}{\sqrt{2}}\right)\mu_0.$$

Now when we use (11) this results in

$$\theta_0 - \varepsilon_0 \leq I'(x) \leq \theta_0 + \varepsilon_0.$$

Now if we would restrict ourselves to $I(x) \in I_1$ we would find similar bounds in the sense that then $\theta_1 - \varepsilon_1 \leq I'(x) \leq \theta_1 + \varepsilon_1$. The fact that $I'(x)$ maps all the instances to the same interval means that the Gini fitness we have that

$$f_g(I) = f_g(I'),$$

by taking the same $\hat{\theta} \in (\theta_0 + \varepsilon_0, \theta_1 - \varepsilon_1)$. But when we calculate the Fisher fitness it follows that

$$f_f(I) = \frac{|\mu_0(I) - \mu_1(I)|}{\sqrt{\sigma_0(I)^2 + \sigma_1(I)^2}} < \sqrt{2}\frac{|\mu_0(I) - \mu_1(I)|}{\sqrt{\sigma_0(I)^2 + \sigma_1(I)^2}} = f_f(I').$$

This perfectly illustrates what we wanted to show. In the specific case where the individual I already perfectly separates the data perfectly the Gini fitness alone cannot deem an individual I more important, or better, than individual I'. In case of the Fisher fitness the algorithm would keep trying to find individuals with higher fitness. This, while the data is already perfectly separated. Therefore the Fisher fitness is more prone to overfitting. Now the specific transformation we proposed would not necessarily impact the accuracy and performance of the newly created individual, it does show that it is fairly easy to create individuals that seem better but are just only more complicated. Where overfitting could occur is the case when the algorithm would pick up noise from the data and use that noise to create new individuals. If these newly created individuals behave similarly to the situation we just described, i.e. have a higher Fisher fitness but not create a better split, this will likely impact the general performance of the created individual on unseen data.

Remark 10. *The issues of the Fisher fitness described in this section are not limited to individuals that already perfectly split the data. In theory, it is possible for individuals that do not perfectly split the data to behave similarly. In general for every new individual $\hat{I}$ for which we have that*

$$\frac{|\mu_0 - \mu_1|}{|\hat{\mu_0} - \hat{\mu_1}|} < \frac{|\sigma_0 - \sigma_1|}{|\hat{\sigma_0} - \hat{\sigma_1}|},$$

will result in $f_f(I) < f_f(\hat{I})$. So as long as the spread of the classes between the individuals decreases faster than the mean of the classes this will result in a higher fitness. If this new individual does not result in a better Gini impurity it might be an indication that the individuals are overfitting.

Since we deal with automatically generated functions we have to carefully consider our operators. The genetic programming implementation of DEAP only considers the arity of the functions when creating individuals, only when evaluating the mapping the program will find out if the function is admissible. What we mean with this is if a function selects a division operator in one of the inner nodes and the input for the denominator is equal to

0 for some instances $x \in \mathcal{D}$ our mapping becomes inadmissible. For example consider the one-dimensional problem where $\mathcal{X} = \mathbb{R}$, the individual corresponding to the function

$$f(x) = \frac{1}{\sqrt{x-2}}. \tag{12}$$

For it to be an admissible function, it must have a domain that is equal to $[2, \infty)$. We do not consider complex values as we defined in Definition 7 individuals as mappings from $\mathcal{X} \to \mathbb{R}$. But this illustrates a problem, we must have that all the individuals that are created are well defined on $\mathcal{X}$. In the example of the division, we can circumvent this problem of the well-defined property by defining a new division operator. Let the new division operator for genetic programming be

$$\text{newdiv}(x, y) = \begin{cases} \frac{x}{y} & y \neq 0 \\ 0 & y = 0 \end{cases}.$$

However, while we have solved the problem of the mapping from the individual in (12) not being well defined we have introduced a new problem. Therefore we refer back to the observation we made earlier in this section. Suppose again that we have an individual I that perfectly separates the class 0 and class 1 individuals, again into the intervals I_0 and I_1. We define a new individual I' as

$$I' := \frac{I(x)}{0}\mathbf{1}\left\{I(x) \in I_0\right\} + \left(1 + \frac{I(x)}{0}\right)\mathbf{1}\left\{I(x) \in I_1\right\}.$$

We have now created an individual for which all class 0 instances get mapped on the value 0, whereas the instances of class 1 all get mapped to the value 1. This leads to $\sigma_0^2 = \sigma_1^2 = 0$, and therefore the Fisher fitness is undefined. We could define the case where $\sigma_0^2 = \sigma_1^2 = 0$ as a Fisher fitness of ∞, but this would not solve the most apparent issue. This issue is, and the Gini fitness is also subject to this, that during the evolution the algorithm will make use of the fact that dividing by zero returns a constant. This makes it easy, compared to the ordinary situation, to create individuals that have instances mapped onto one line. We will consider two ways in which this issue can roughly materialize itself:

1. The algorithm picks up, early in the evolution, individuals that make use of this newly defined operator to deliberately use it to separate instances. Then these individuals and their subtrees are represented during the remainder of the evolution. This will most likely lead to individuals which are full of this new operator.

2. This specific way is maybe a little more of a concern to the Fisher fitness. When we have an individual I it was already shown that we can improve the Fisher fitness while keeping the Gini fitness constant, and hence it would not be of benefit to the classification problem. Self-defined new operators can work this particular behaviour in the hand as we will show in Example 4 and therefore is not only limited to division.

We do not want the first to happen because it can limit the interpretability of the individuals. Namely, what does it even mean in the real world to divide by zero? One could argue that it certainly would not be a mapping to zero, you could choose to let it map to infinity. If that would be the case we would need to redefine our concept of individual and let them map to the

extended reals $\bar{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$. But remember we are trying to create new features. What would it mean for a feature to be infinite? Division by zero from a mathematical standpoint is undefined and for good reason. All in all, we have to be careful with defining new operators. Another example of being careful with defining new operators is given in the example below

Example 4. *Let I be an individual who corresponds to the mapping*

$$f(x) = \log(\log(x)).$$

Assume we have redefined the logarithm such that it can take negative values. Where we defined it in the following way

$$\log(x) = \begin{cases} \log(x) & x > 0 \\ 0 & x \leq 0 \end{cases}.$$

This has as a consequence that all $x \leq e$ get mapped to the values in $\mathbb{R}_{\leq 0}$, whereas all values in the interval $[0, 1]$ get mapped to the value 0. Consider the following easy dataset $\mathcal{D} := \{(0, 0), (\frac{1}{2}, 0), (\frac{1}{3}, 0), (3, 1)\}$. It is not hard to show that individual I' which corresponds to the function $g(x) = x$ achieves the same Gini fitness as function $f(x)$. Whereas the Fisher fitness of both functions are different, even so much that $f(x)$ has an undefined Fisher fitness.

The above example gives another illustration of how we should be careful with defining new operators. Even so that the example clearly shows how that can impact the interpretability of the individuals created. In the example, individual I tells us a story. That is, we want to split the instances at either the point $x = 1$ or $x = e$. Whereas individual I' exactly shows the same but does it via an identity function. Now this example is based on a very easy dataset $\mathcal{D}$ so that individual I is somewhat interpretable, but the excessive use of compositions of logarithms could lead very fast to less intuitive transformations. It also shows that when we again consider both fitness candidates, that the Gini fitness in the specific example is less influenced by these self-defined operators.

5 Application of the developed algorithm

In this section, we will look into applications of the newly developed algorithm. First, we will introduce our testing procedures. Given a dataset $\mathcal{D}$ on which we want to apply GP or any other machine learning technique, we will divide $\mathcal{D}$ in a test and train set. This split will be done on a 70/30 basis. This is done using the 'train_test_split' function from the sklearn library [47]. Every time a train-test split is made this is done by specifying a 'random_state'. This random_state is chosen to be equal to 12 for all train-test splits, this is to ensure consistency by comparing different runs. When referring to the accuracy of a classifier h we will be talking about the fraction of data that is correctly classified from the test dataset. That is, for $x_i \in \mathcal{D}_{\text{test}}$ and let $|\mathcal{D}_{\text{test}}| = n$, then the accuracy of a classifier h is given by

$$A(h) := \frac{\sum_{i=1}^{n} \mathbf{1}\{h(x_i) = y_i\}}{n}.$$

The accuracy of the GP algorithm will be the performance of the Decision Tree trained on the created features by the algorithm. That is we train the Decision Tree on a dataset $\mathcal{D}_{\text{train}}$ as shown later on in Section 5.4.2. We note that this notation of accuracy is only useful when we have a balanced dataset. That is, each label occurs equally often. In the case of the Rabobank dataset this will not be the case and hence accuracy in that specific dataset will be the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC). For more information about the AUC-ROC concept of accuracy, we refer to [48]. In addition to this we note that the Decision Trees created with the Rabobank data are created in Dataiku, so for exact details as to how Dataiku trains its Decision Trees we refer to the documentation of Dataiku DSS 9.0 `https://doc.dataiku.com/dss/9.0/`. The parameters on which the Decision Trees were trained using Dataiku are a maximum_depth of 3,4,5 and a minimum samples per leaf in the interval [1,4000] where a grid search was done using 100 values. The optimization metric for the hyperparameters was the AUC score.

5.1 The algorithm vs random search

In section 3.4.2 we discussed the brute-force search method. This search method can also be used to go over all possible functions that GP can find, where we then would exhaustively search the space of all possible functions. For this to be possible we must have a limit on the maximum tree size and we must have that the sets $\mathcal{F}$ and $\mathcal{T}$ are finite. If this were not the case we would have an infinite amount of possible trees. In practice, this means that the set $\mathcal{T}$ cannot contain uniform random variables, which is often used to add real values from intervals to the building blocks of the functions. In practice, it will almost always happen that the tree size is finite. This happens because of the limitations in the programming language Python and the obvious memory/space requirements. Most of the time brute force is not a viable solution to searching for functions. This is due to the possible amount of trees we can construct using rather limited sized $\mathcal{T}$ and tree size. We refer back to Section 3.5.3 for a more in-depth look at this. One of the most approachable solutions to searching functions would be a random search method, which will be the main comparison we will be comparing our new algorithm with. First, we discuss how we will be applying random search, there are two different ways to randomly searching the space.

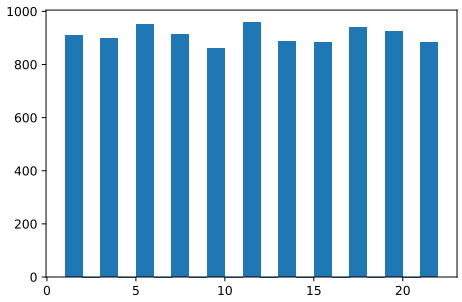
First, we could select a maximum and then uniformly select a tree size, and then create a tree of that size. For example, let the maximum be 5. In the case of binary trees only, we sample a length from $[1, 3, 5]$. Then create a tree of the given sampled length. This will create a uniform distribution over all the lengths. This uniform distribution heavily favours the lower-length trees. That is, they are way less possible trees of lower lengths. What we mean by this is that the amount of different trees that can be constructed for length 3, is way lower than the number of different binary trees that can be constructed with for example length 11. As an example, take $|\mathcal{T}| = 20$ and $|\mathcal{T}| = 4$. Then using Proposition 2. It follows that

$$\frac{20^6 \cdot 4^5 \cdot 42}{20^2 \cdot 4^1 \cdot 1} = 20^4 \cdot 4^4 \cdot 42 = 1.72032 \cdot 10^9,$$

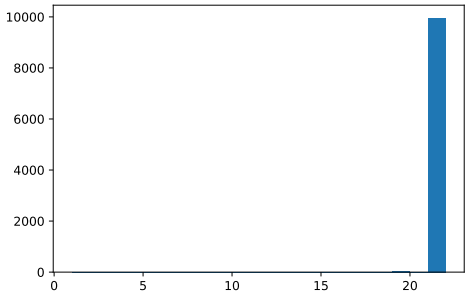
where we have divided the different amount of function of length $L = 11$ by the number of functions of length $L = 3$. Hence there are $1.72032 \cdot 10^9$ more functions of length 11. If then both lengths have equal probability to be created, it is clear that length $L = 3$ has the advantage. This advantage is that more of its functions have a probability to occur in the search process. Furthermore, this affects the quality of the results too. If the optimal function has a lower length, for example, $L = 3$. Then it has a way higher probability of being discovered compared to an optimal function of length $L = 11$. One possibility to circumvent this is to let the probability of selecting a certain length $L = l$ be proportionate to the possible functions of length l. Let the max length be $k \in \mathbb{N}$ where $k \geq 3$ is odd. Let the $\Xi = \{n \in \mathbb{N} : n \text{ odd and } n \in [3, k]\}$ which is the set of all possible lengths for the trees where the length is greater than 1. Let L then be the random variable from which we sample the length of a tree we will be creating, then we define

$$\mathbb{P}(L = l) := \begin{cases} \dfrac{|\mathcal{F}|^{(l-1)/2} \cdot |\mathcal{T}|^{(l+1)/2} \cdot \bar{C}_l}{|\mathcal{T}| + \sum_{\xi \in \Xi} |\mathcal{F}|^{(\xi-1)/2} \cdot |\mathcal{T}|^{(\xi+1)/2} \cdot \bar{C}_\xi} & l > 1 \\ \dfrac{|\mathcal{T}|}{|\mathcal{T}| + \sum_{\xi \in \Xi} |\mathcal{F}|^{(\xi-1)/2} \cdot |\mathcal{T}|^{(\xi+1)/2} \cdot \bar{C}_\xi} & l = 1 \end{cases},$$

where $l \in \{1\} \cup \Xi$. In the figure below we can view a sample of the sampled lengths l from L for both distributions. Both where sampled $n = 10000$ times



(a) Sample from the uniform distribution of L.



(b) Sample from the proportionate distribution of L.

Figure 13: Sample distributions of both sample methods.

As we can see in the figure, when we sample from the proportionate probability distribution the sampled value is almost always equal to 21. This is because the sheer size of different trees with size 21 is so much higher than the other possible sizes. If the perfect function (or tree) is of smaller size the proportionate creation method will most likely not find it. First, we will compare the three on an artificial dataset that was created by using the sklearn Python library. The function 'make_classification' is a Python adaptation of the code described in Appendix B in [49]. The parameters to create the dataset are

n_samples	1000
n_features	300
n_informative	5
n_redundant	0
shuffle	False
random_state	420

Table 2: Overview of the parameters of the 'make_classification' function from the sklearn library.

We note that the parameters in Table 2 are a baseline for the dataset. We will most of the time not use all 300 features in the dataset. Most of the time, like in this case, we will select the first 100 columns from the dataset such that we have 100 features in total. From these 100 features, there are 5 features that are deemed informative to classify the instances to a certain class 0 or 1. The shuffle parameter influences whether or not the features are shuffled, in the case of 'False' this means that the first 5 features of this dataset will be the informative ones. We will now proceed to apply the search algorithms on this created dataset with 100 features. The GP algorithm parameters are equal to those in Table 1 but the max_gen parameter was set to 300. For the random algorithms, we created 100000 individuals each run and selected the best performing one based on fitness. The algorithms were repeated 100 times where the result was the fitness of the extracted feature. This gave us the following results

algorithm	avg_fit$\pm std$
GP	0.262 $\pm$0.0480
Random_uni	0.298 $\pm$0.000627
Random_prop	0.335 $\pm$0.0377

Table 3: Results of the search algorithms on the artificial (sklearn) dataset.

Over 100 runs the functions got extracted by the GP algorithm were

$$f(x) = x_4 - x_2,$$
$$g(x) = x_4 + x_5,$$
$$h(x) = x_3 + 2x_4 + x_5,$$

where $f(x)$ was the most frequent. So it shows that the GP algorithm was able to explicitly extract the informative features and use them to create new functions. Since these functions have short trees, the random uniform search obtained similar functions but occasionally added with a noisy feature. This shows that the random uniform search is not able to distinguish between informative and noisy features. Although we must note that this artificial dataset does not show that the GP algorithm can extract intuitive features. We will explore this further in 5.4.2. From the table, we see that there is a marginal improvement in fitness when using the GP algorithm. Although it has a higher standard deviation. This is due to the fact that sometimes the GP algorithm gets stuck in a local minimum and is not able to escape it. The random search algorithms are not prone to this behaviour. But as we know these search algorithms have other caveats. First of all, we have noticed during our initial exploration of the data. There are many short functions of the first 5 features that perform reasonably well on splitting the data. These functions are of around size 3, which is heavily favoured for the uniform random search algorithm. The results from Table 3 are very interesting but they do not tell the whole story. Eventually, we want these functions to be used in a Decision Tree algorithm. Therefore the trees found by the algorithms must consist of only the first 5 features. If for example feature number 50 occurs in the tree it means that the function is picking up noise, which will not benefit the classification performance of the Decision Tree. To get better insight we proceed to use the same algorithms but change the number of functions that they extract. In all cases, it will be that we extract three functions. Then we proceed to fit a Decision Tree on the train set with these three functions, the Decision Tree will have a max depth of 2. Then we evaluate the accuracy on the test set, the whole procedure is then repeated 100 times. This leads to the following results

algorithm	avg_acc$\pm std$
GP	0.880 $\pm$0.0270
Random_uni	0.859 $\pm$0.0326
Random_prop	0.817 $\pm$0.0163

Table 4: Results of the search algorithms extracting three functions on the artificial (sklearn) dataset.

In the table, we see similar behaviour to the results obtained earlier. Although we must note that the GP algorithm now has way less deviation, as the standard deviation of the

results has decreased.

Since we know that the artificial dataset created with the sklearn library has certain benefits for the random uniform search method, we will proceed to also introduce a self-created artificial dataset. This is so that we can have more influence on which instances get which label. We create the following data

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,5} \\ & \cdots & \\ x_{n,1} & \cdots & x_{n,5} \end{bmatrix}, \quad X' = \begin{bmatrix} x'_{1,1} & \cdots & x'_{1,95} \\ & \cdots & \\ x'_{n,1} & \cdots & x'_{n,95} \end{bmatrix},$$

where each $x_{i,j}$ is sampled from a discrete uniform distribution $X_{i,j} \sim U[0, 10]$ and each $x'_{i,j}$ is sampled from a normal distribution $X'_{i,j} \sim \mathcal{N}(0, 1)$. Let $x_i = [x_{i,1}, \ldots, x_{i,5}]$. We then proceed by creating a new matrix $\hat{X}$ which will be the following

$$\hat{X} := \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad or \quad \hat{X} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,5} & x'_{1,1} & \cdots & x'_{1,95} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n,1} & \cdots & x_{n,5} & x'_{n,1} & \cdots & x'_{n,95} \end{bmatrix}.$$

We then proceed by defining the labeling function

$$y_i = \mathbf{1}\left\{\sum_{j=1}^{5} x_{i,j} > 25\right\}.$$

Hence we get a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. On $\mathcal{D}$, we will apply the aforementioned algorithms to compare them in a more controlled environment. We note that the matrices X and X' were created using a predetermined seed. The seed in question was 'np.random.seed(1)', which is needed to reproduce the results. If we then run the algorithms on this new dataset, where we use the standard parameters and create only one split we get the following table

algorithm	avg_acc$\pm std$
GP	0.925 $\pm$0.0750
Random_uni	0.743 $\pm$0.0255

Table 5: Results on the artificial dataset with self-created labeling function. Average is taken over 10 runs.

In many cases, the GP algorithm was able to correctly find the rule. However again in some cases, the algorithm got stuck in some local optima and was not able to escape. On average most runs found the correct labeling rule. It does show that the GP algorithm can build functions from the ground up. Hence across the board, we see that it indeed makes sense to use our algorithm as opposed to the random search algorithm. This is however specific to these datasets, and thus there is no guarantee that it will always outperform the random search algorithm.

Furthermore, we want to shed a light on the functions that were extracted by the algorithms. In particular, the functions extracted on the first dataset we discussed in this section. The random uniform search algorithm often found functions containing 1 or 2 informative features, but in addition to this often other features were present as well. An example of functions that were extracted in different runs on the first split are

$$f_1(x) = -x_2 + x_4 + x_{41},$$
$$f_2(x) = -x_2 + x_4 + x_{59}x_{91} - 1,$$

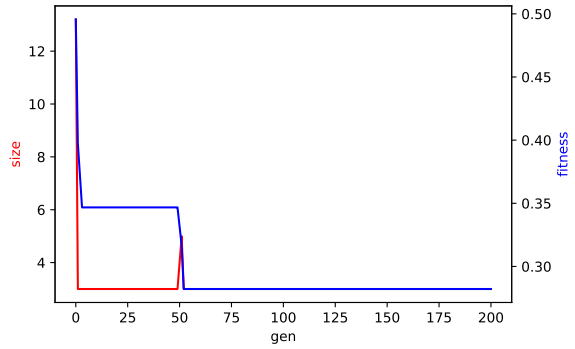
whereas the pick-up of the noise features was never present in the GP algorithm. This means that the features x_i, where $6 \leq i \leq 100$, were never present in the functions extracted by the GP algorithm. This is due to the fact when the GP algorithm picks up a function like $f_1(x)$ it will search for functions that are close to $f_1(x)$, where close refers to similar. In this particular situation, it would disregard the 41-st feature and obtain the function $g(x) = -x_2 + x_4$, which is not prone to any noisy variables. Namely, we see that none of the noisy variables x_i, where $6 \leq i \leq 100$, are present in the function. Thus in the context of this particular dataset it would mean that the extracted function better describes the interactions between the variables.

Another benefit of using the GP algorithm is that it takes way fewer calculations. Remember that every individual that ever gets created needs to be scored with a fitness. In the GP algorithm fewer individuals had to be scored compared to the random search algorithm which caused the run-time of the GP algorithm to be lower.

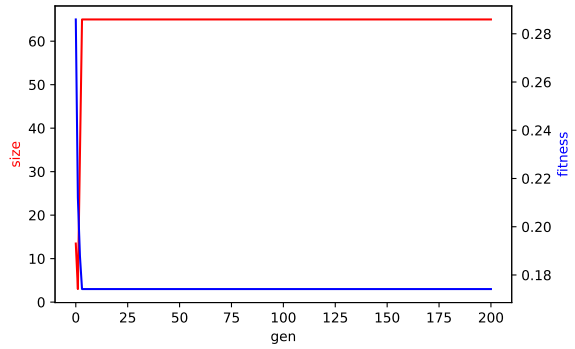
5.2 Artificial Data

5.2.1 The effect of selection methods

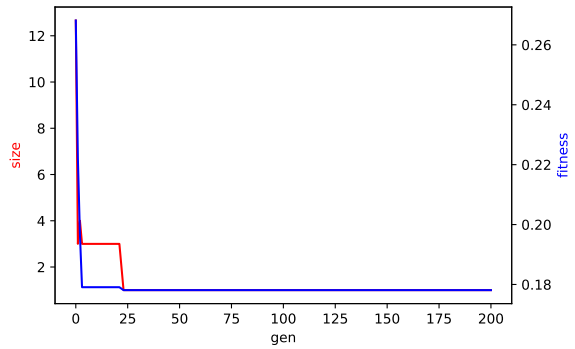
In this subsection, we will compare the effect of various selection methods introduced in Section 3.5.5. These tests will be conducted on the artificial dataset. First, we ran a few tests with the 'selBest' method using the $(\mu + \lambda)$-algorithm. We did this by selecting 2 individuals with the highest fitness and created 200 children using our standard crossover and mutation probabilities. We ran the evolution for 200 generations. The initial population had size 1000, and was initialized with the RHH method with a min depth of 1 and max depth of 5. The results are shown in Figure 14 and Figure 15.



(a) Graph of average fitness per generation and average size of the functions per generation on the first split.



(b) Graph of average fitness per generation and average size of the functions per generation on the second split.



(c) Graph of average fitness per generation and average size of the functions per generation on the third split.

Figure 14: Overview of the 'selbest' selection method.

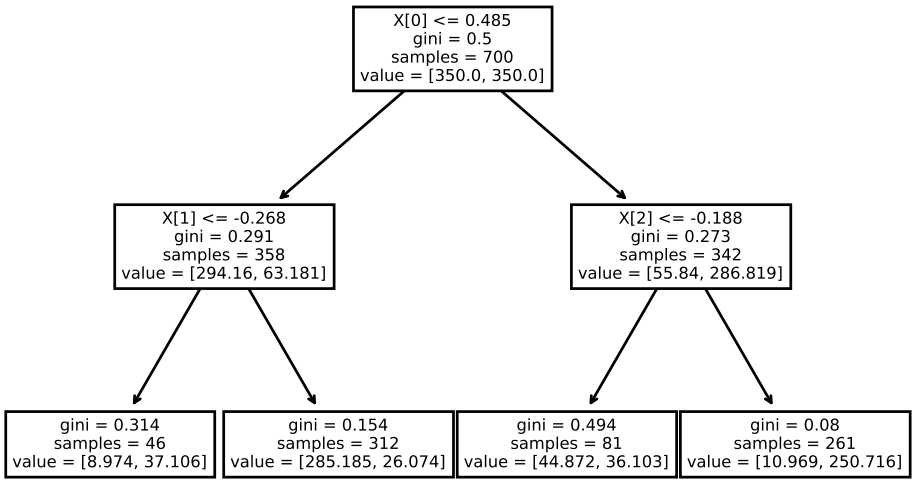
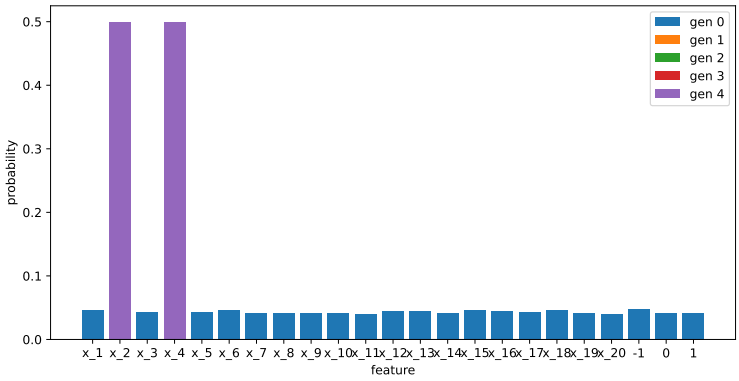


Figure 15: A Decision Tree that got created by using the features created by the algorithm.

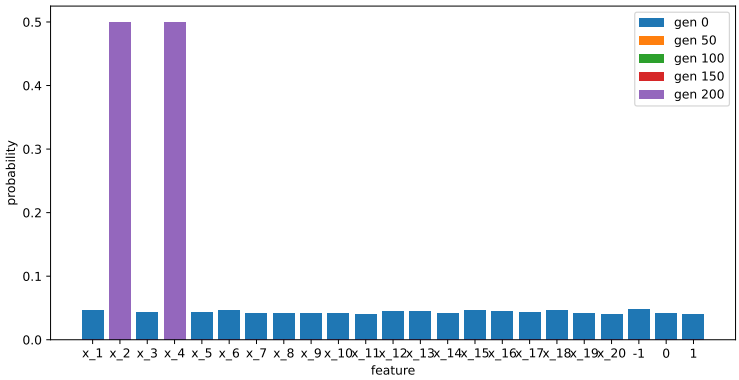
The test performance of the Decision Tree was trained on 3 extracted functions. The test accuracy of the Decision Tree with max depth 2 was equal to 0.84. We must note that this particular run, for which we received the above graphs was done with a random seed. The seed used was the number 64. Doing the run without the seed results in different graphs and results. Although the graphs were different every time, the behaviour was the same. The population converged really fast to an optimum. The genetic diversity, that is the unique count of terminals and functions in the population, became very low. This in itself made it very difficult for the population to escape the local optimum. If you only have two possible parents and create 200 children there is a high likelihood that the children will be similar. This is especially true when the size of the parents is small. The only possibility to create children that are completely new to the population is by mutation. As we have set this mutation rate to 0.1, mutation will not happen often. In fact, we expect it only to happen 2 times. Since the mutation operator creates a random subtree in a parent, there is a very high likelihood that it will contain multiple of the 295 noise variables (terminals). Hence the low probability of escaping the local optimum. Although it might sound like this approach is not useful at all we must note that the accuracy was decent ranging from an accuracy as low as 0.80 to somewhere around 0.88, with an average of 0.85. The features extracted were short and could easily be explained. Only the function for the first split contained multiple terminals, namely $f_1(x) = x_4 + x_5$. The other two splits were just terminals, in this case $f_2(x) = x_2$ and $f_3(x) = x_4$. The function f_i corresponds to the variable $X[i]$ in the Decision Tree in Figure 15. This means that only the first split was not able to be achieved from the original dataset.

In this specific case, this leads to an 0.02 test performance increase. On average a marginal improvement of 0.03 was achieved this way. This indicates to us that indeed the 'selBest' selection method, where $k = 2$, is heavily susceptible to the initial population and so the local optima where it converges towards. As the initialization of the population using the RHH method mostly creates functions with a shorter length for this dataset. The convergence to the local optimum is most of the time of shorter size, resulting in shorter functions to be extracted. Sometimes the population, which had size k, was able to escape the local optimum via mutation or crossover but we did not see this happen after 50 generations. In Appendix A we can find another figure which shows that indeed there is similar behaviour for independent runs.

Another selection method we discussed was tournament selection as defined in Definition 19. We will mainly discuss this selection method to elaborate on the fact that the 'selBest' selection method has strong selection pressure which leaves almost no room for genetic diversity. To illustrate this we first limit the number of features in the artificial dataset to be 20 in total. This is purely an aesthetic choice. Otherwise, the figures get cluttered. The rest of the parameters are not changed. So the population size where the features are counted is still equal to 2.



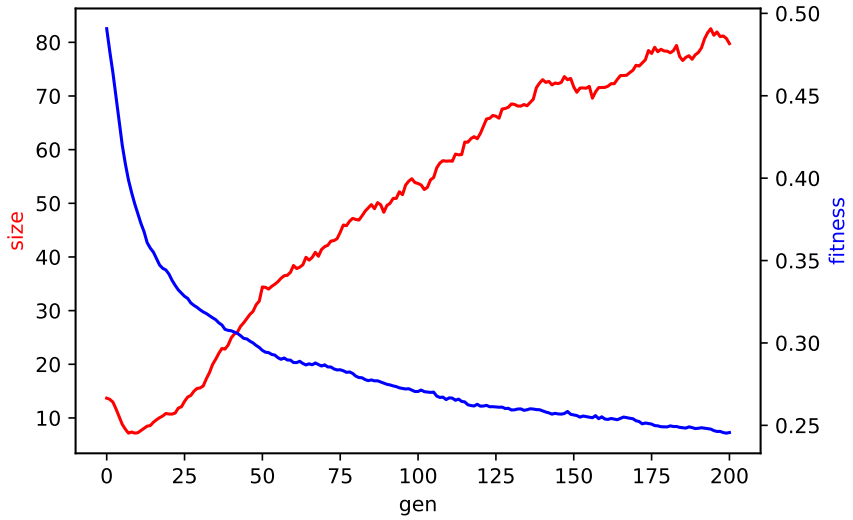
(a) Occurrence of features in generations 0,1,2,3,4 (The non-visible colors are behind the purple lines).



(b) Occurrence of features in generations 0,50,100,150,200 (The non-visible colors are behind the purple lines).

Figure 16: Visualization of the low diversity of 'selBest'.

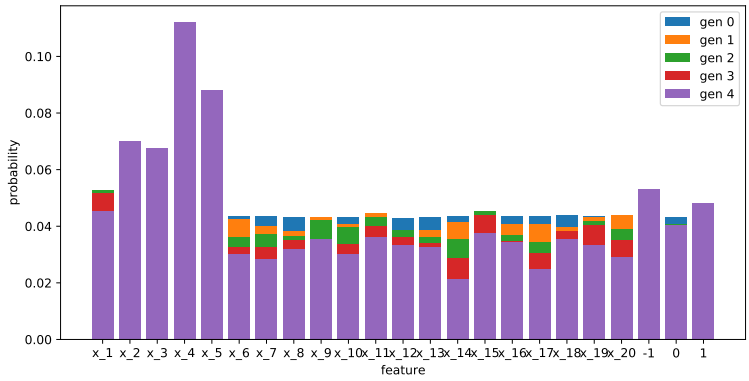
For this analysis, we will limit ourselves to the first split only. We see that in this figure the diversity of gen 0 is completely gone in any of the other generations. For a second run, done with a random seed of 128, we see that early on in the generation there is still a -1 in some of the functions, see Figure 28 in the appendix. This is easily explained by the fact that multiplying or adding/subtracting -1 has little to no influence on the split quality. This is because -1, just flips or moves the data points equally in the space. In the picture regarding the 50 delta generational information, we see that eventually by chance the functions mutate to contain an x_2, as this then leads to a lower Gini it directly swamps the population. If we now compare this selection method with the tournament selection we find the following plots.



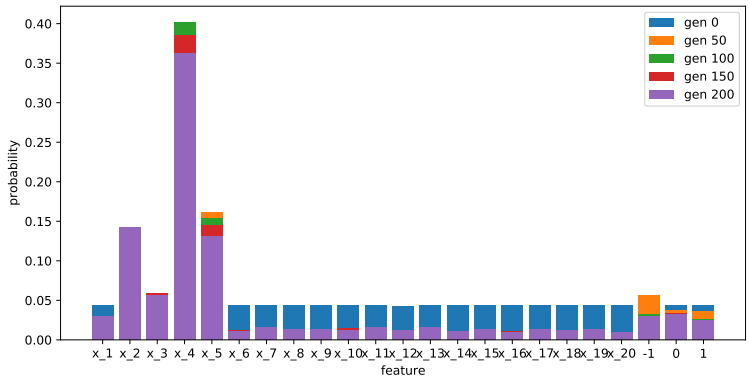
(a) Graph of average fitness per generation and average size of the functions per generation on the first split.

Figure 17: Figure of average size of the population vs average fitness.

We took an average of 100 runs, and a tournament size of 2. So in this graph, we can find the average fitness of a population in generation x averaged over 100 runs and the average size of the population. This would give us an idea of how the selection method would behave on average. We see that this selection method behaves differently from the selBest method which is shown in Figure 14. As we already illustrated below Definition 19, mediocre individuals also have a chance at survival. This effectively makes the average fitness improve over time (=generations). Simultaneously we see that, as the fitness improves, the average size of the individuals increases. This illustrates another problem that occurs in genetic programming which is the effect of bloat. Bloat is a phenomenon in Genetic Programming where the size of the individuals gradually increase. In addition to this, the fitness of the individual improves way slower or even not at all. This is problematic as it hinders the efficiency of the individuals. Furthermore, the individuals are less likely to be interpretable. For more information on bloat we refer to [50].



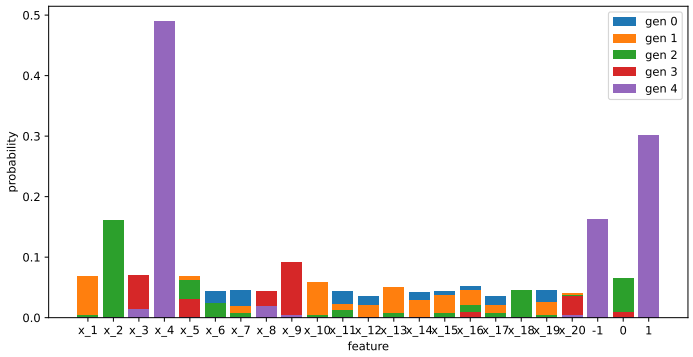
(a) Occurrence of features in generations 0,1,2,3,4 for tournament selection.



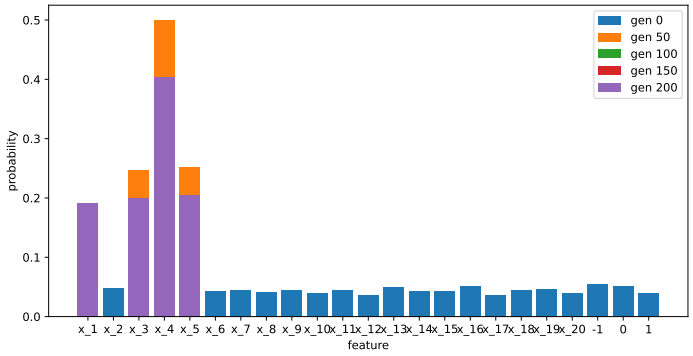
(b) Occurrence of features in generations 0,50,100,150,200 for tournament selection.

Figure 18: Visualization of the diversity for tournament selection.

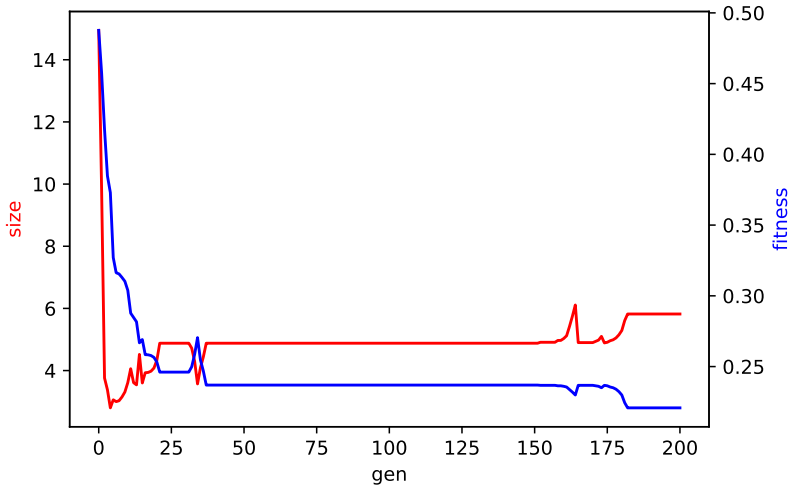
As we can see in the earlier parts of the evolution we have a lot more genetic diversity compared to the 'selBest' method. This way, we expect the tournament selection to be more robust over different runs. That is, more likely to find the same solution over different runs. The selection methods can be compared in the following way. The 'best' selection method can be viewed as the population jumping from local optimum to local optimum, where the tournament selection can be viewed as the population moving more 'smoothly' from local optimum to local optimum. This behaviour coincides with what we expected from it in Section 2. As we already mentioned the occurrence of bloat in the tournament selection method is the reason why we introduced the NSGA-II selection method in Section 3.5.5. In the figure below we find an overview of the NSGA-II selection method on the before discussed concepts.



(a) Occurrence of features in generations 0,1,2,3,4 for NSGA-II selection.



(b) Occurrence of features in generations 0,50,100,150,200 for NSGA-II selection.



(c) Size vs fitness over 100 runs for NSGA-II.

Figure 19: Visualization of the diversity for NSGA-II.

We see that NSGA-II is indeed less prone to bloat. In addition to this, we see that the NSGA-II selection method also seems to have a bit of diversity after 3 generations. But in general, this selection method was already in a local optimum shortly after the evolution started. From the figure, however, we see that it was also able to escape this local optimum. Namely, the feature x_1 came back into the population. But being less prone to bloat was the most important characteristic as to why we choose to use the NSGA-II selection method.

5.2.2 The impact of the parameters μ and λ and the initial population size

When setting parameters the most important parameters are μ, λ, and the initial population size. The value for μ and λ are important as they play a big part in the evolutionary process. Therefore we decided to perform a big comparison. We compared the artificial dataset with 20, 40, and 100 features for various combinations of μ, λ, and the initial population size. We ran the algorithm 100 times and took the average accuracy over these 100 runs.

parameter	**value**
max_gen	200
initial_pop_size	n
μ	μ
λ	λ
cxpb	0.9
mutpb	0.1
early_stopping	100
#_functions	1

Table 6: Parameters used in obtaining the data for comparison.

We note that the max_gen parameter is set to 200 because of run time constraints. Furthermore, the values of μ' and λ' are from the pair (μ', λ') where the pair is from the set

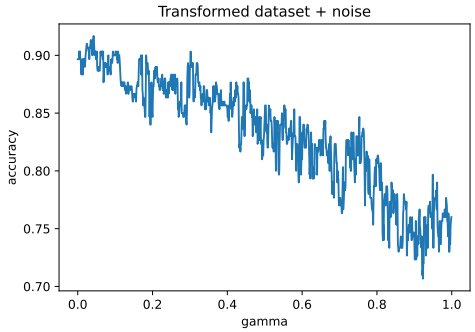
$$\{(\mu, \lambda) : \mu \in [10, 20, 40, 60, 80, 100, 150, 200], \lambda \in [10, 20, 40, 60, 80, 100, 150, 200, 300, 400]\}.$$

For every combination of (μ, λ) we ran the algorithm with $n \in \{100, 200, 400, 800\}$. The results for 20 features follow from Figures 29,30,31,32 in Appendix B. The results for 40 features follow from Figures 33,34,35,36 in Appendix B. The results for 100 features follow from Figures 37,38,39,40 in Appendix B. There are a few things that stand out when viewing the results. The triangle, in Figures 29 to 40, represent the average accuracy whereas the corresponding colored line is the standard deviation of the accuracies. We see across all figures that when λ increases the average accuracy increases with it. Remember that λ was the amount of offspring created from the population at a certain time in the evolution. So from the results, it seems that creating more offspring will result in a higher average accuracy of the algorithm. In addition to this if λ is small, i.e. $\lambda \in \{10, 20\}$, the results are very inconsistent and on average way lower compared to higher values. This behaviour seems to be independent of the value for μ. Furthermore, we conclude that when λ is high, i.e. $\lambda = 400$, the performance is about the same regardless of the initial population size. Since our Rabobank dataset will contain a lot of features, we will mainly base our conclusions on the results from the artificial dataset with 100 features. We conclude that higher λ on average

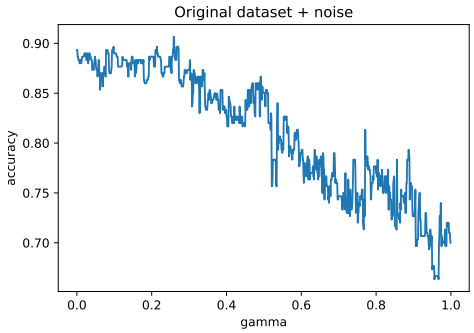
increases the accuracy of the created feature. Where the choice of μ has little to no impact. We also conclude that there is no harm in choosing a high initial population size. This is how we concluded upon most of the standard values in Table 1 where the value of λ is mostly determined by run-time restrictions.

5.2.3 Performance of features on noisy data

We created a dataset X and a label set Y using the function $make\_classification()$[1]. We proceed in the following way. First, we apply our standard form of GP on the dataset X, and obtain a set of transformations τ. We then define a new dataset $X_\gamma := X + \gamma Z$ where $\gamma \in [0, 1]$ and Z is a matrix of the same dimension as X, and where the values $z_{ij} \sim \mathcal{N}(0, 1)$. We then apply our transformations from the set τ on the new dataset X_γ for various $\gamma \in [0, 1]$. We then obtain a new dataset X_γ^τ, which contains all the transformed rows of X_γ. We then do a 70/30 split on X_γ^τ and train a Decision Tree classifier on the training set. We do this for every $\gamma \in [0, 1]$ and plot the corresponding accuracy on the test set. We do the same on the 70/30 split of the non-transformed dataset X_γ and again plot its test accuracy. This results in the following plot



(a) Out of the box DT performance on X_γ^τ.



(b) Out of the box DT performance on X_γ.

From the data, we conclude that the GP features are more sensitive to noise. This can be seen by the spike behaviour compared to the original datasets performance on the noise. It also looks like it is decreasing a bit earlier, around $\gamma = 0.1$ compared to $\gamma = 0.3$ for the original dataset. In general, the behaviour of both datasets is similar in the sense that the accuracy decreases at the same rate. The transformed dataset had 9 features.

5.3 Time series data

One of the initial goals for developing this algorithm would be to automate the feature extraction for time series. Where we want the features to either already be very discriminative or give us insights into which types of features are interesting. Therefore we will consider an artificially created time-series dataset. We will proceed as follows, let X be the dataset,

[1]The parameters are as following $n\_samples = 1000, n\_features = 30, n\_informative = 5, n\_redundant = 0, shuffle = False, random\_state = 420$

consisting of time series x_i. Where each $x_i = [x_{i,1}, \ldots, x_{i,100}]$ such that we have that

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad or \quad X = \begin{bmatrix} x_{1,1} & \ldots & x_{1,100} \\ & \ldots & \\ x_{n,1} & \ldots & x_{n,100} \end{bmatrix},$$

Furthermore, each $x_{i,j}$ is sampled from a discrete uniform distribution such that $X_{i,j} \sim \mathcal{U}[0, 42]$ and thus $x_{i,j}$ is a realisation of $X_{i,j}$. Now we define the vector Y to be the vector containing the labels of each time series. This label will be based on statistical information of the time series. We define

$$y_i := \mathbf{1}\left\{ \frac{1}{100} \sum_{j=1}^{100} x_{i,j} > 21 \right\},$$

where y_i is the label associated with time series x_i. Hence we have now created a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. Below we find a figure of two differently labeled artificially created time series

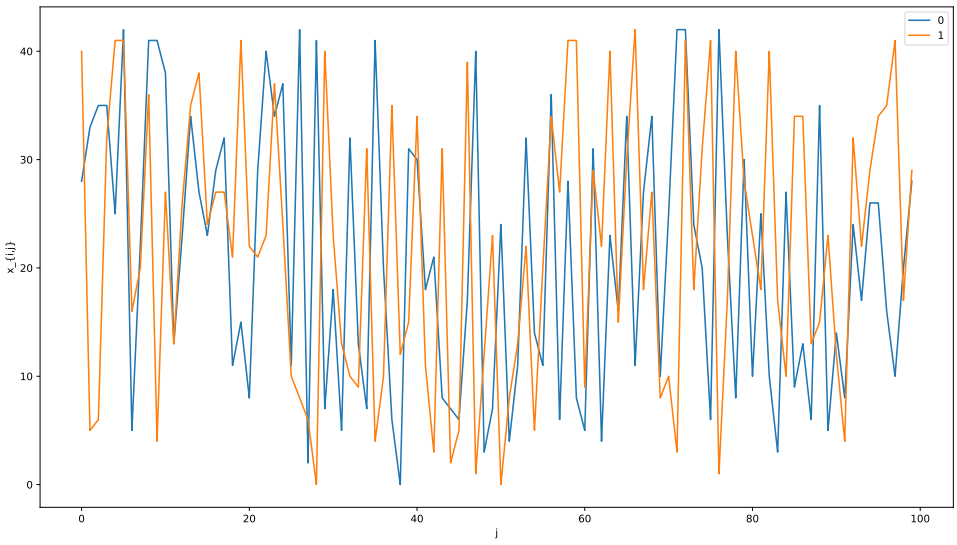


Figure 21: Picture of two time series, with different labels. The orange is label 1, whereas the blue is label 0.

Let us take $n = 10000$, and apply our standard version of the GP algorithm. That is, let the maximum of generations be equal to 1000, $\mu = 100$, $\lambda = 200$, the crossover probability equal to 0.9, and the mutation probability equal to 0.1. Besides that we apply various early stopping criteria, we extract 1 feature and stop the evolution when for 100 generations the best individual has not been changed. We also stress that our function set $\mathcal{F} = \{+, -, *, /\}$,

with our specifically defined division operator. Ideally, when we run the algorithm we want to obtain a function of the form

$$f(x_i) = \sum_{j=1}^{100} x_{i,j}.$$

If this were the case, the Decision Tree will be able to split the data, perfectly, by using the rule $f(x) < 2100$. When running the algorithm the function that is returned is precisely $f(x) = x_1 + x_{12} + x_{13} + x_{18} + x_2 + x_{20} + x_{21} + x_{23} + x_{24} + x_{45} + x_{48} + x_5 + x_{50} + x_{51} + x_{54} + x_{56} + x_{60} + x_{62} + x_{66} + x_{70} + x_{80} + x_{81} + x_{88} + x_{89} + x_9 + x_{91} + x_{92} + x_{95} + x_{99} + \frac{x_{52}}{x_{45}}$. Of course we must note that this varies from run to run. The idea, however, is clear. The algorithm finds that adding the original time points/features results in better fitness, but unfortunately is not able to correctly obtain the whole function. This is far from ideal, as the fitness of the function $f(x)$ was only 0.40 and the length of the individual was 61. This is an illustration of the problems that can occur with GP. The problem of not being able to find the correct describing function can have various reasons. The reasons can be limited function size, too high or too low selection pressure, and many more. In this case, we think we can do better by leveraging the knowledge we have about the type of data we are dealing with. Since we are working with time series, and in this case the classification of said time series, there are various ways to use statistical measures for time series to classify them [51]. One of them is, as maybe expected, moment measures. In particular, the first moment is exactly the function we are searching for in this example dataset. We now want to leverage these moment functions on the time series, and luckily there is an easy way to achieve this. We can just add a new function to the set $\mathcal{F}$, in this case, the consecutive mean. We will consider a function $m_{(a,b)}(x_i)$ defined on a time series x_i as follows. Let $a, b \in \mathbb{N}$ then $m_{(a,b)}(x_i)$ is given by

$$m_{(a,b)}(x_i) := \frac{1}{|a-b|+1} \sum_{j=\min(a,b)}^{\max(a,b)} x_{i,j}.$$

We note that in this specific situation with $\mathcal{D}$, the values of a and b are limited to $\mathbb{N} \cap [1, 100]$. With this new function added to our function set we can define new individuals, that were hardly possible before. See the figure below as an example

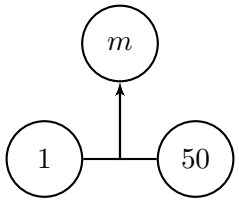


Figure 22: A new individual making use of the new function.

The individual given in Figure 22 represents the function $m(1, 50)$, which for a time series x_i corresponds to the $m_{(a,b)}(x_i) := \frac{1}{50} \sum_{j=1}^{50} x_{i,j}$, which exactly represents the first 50 time points average. We must note that in addition to the enlarged set $\mathcal{F}$, we note that the set $\mathcal{T}$ is also enlarged. By the addition of function m, we must have a way to select the random

integers, to give as input to m. Therefore, we extend $\mathcal{T}$ with a discrete uniform random variable $U \sim \mathcal{U}[1, 100]$. We choose $U \sim \mathcal{U}[1, 100]$ because we have 100 features total. If we now apply GP with the newly defined function set $\mathcal{F} = \{+, -, *, /, m\}$. We obtain the following function

$$g(x) = m_{(1,100)}(x).$$

Hence we see that GP is able, with the correct function set, to extract the true learning rule. This is an important observation, for GP to work well we need to have appropriate function and terminal sets. By working well, we mean that the algorithm can extract well performing features and is fast. We illustrate this with an example

Example 5 (GP working well)**.** *Normally to prevent overfitting and bloat we would limit the number of nodes that a certain tree can have in Genetic Programming. Let us say this limit is set to 21, then it is clearly impossible to achieve a function of the form $f(x) = \sum_{j=1}^{100} x_{i,j}$, as when representing this function in tree form it will have far more than 21 nodes. So assume for now that the tree size is unlimited, or at least set high enough such that the function $f(x)$ can be represented in tree form where we assume the latter to be the case. As we have shown in this section the algorithm was not able to extract the true labeling rule with the given parameters λ, μ, and number of maximum generations. Although since we assume that the function $f(x)$ can be represented in tree form, there is a probability which is greater than 0, that the said function $f(x)$ will eventually occur in the population. One of the easiest ways to see this is that the probability that $f(x)$ is created by subtree mutation is greater than 0. Hence when performing subtree mutation, which has a probability of 0.1 of happening, there is a probability that the newly created individual will exactly be $f(x)$. So eventually with enough time, the algorithm will find the correct rule. Compared to the case where we add the function m to $\mathcal{F}$ though, the process of finding the function $f(x)$ or equivalent in terms of the Gini fitness is way easier. As we said, which can be checked by running the program yourself, with the expanded $\mathcal{F}$ the algorithm can find the correct labeling rule with the same parameters. Which in this case we refer to as GP working well.*

Now that we have seen that the algorithm is able to construct the correct labeling function using the dataset $\mathcal{D}$ we ask ourselves how well the algorithm will perform when we add noise to the dataset $\mathcal{D}$. We proceed in the following way, again consider the matrix X created in the same manner. Again we have that each label for a time series x_i is given by

$$y_i := \mathbf{1}\left\{ \frac{1}{100} \sum_{j=1}^{100} x_{i,j} > 21 \right\}.$$

Now we define a new matrix X_γ in the following way

$$X_\gamma := X + \gamma Z,$$

where Z is a matrix of the same dimensions as X, but where each z_{ij} is a realization of $Z_{ij} \sim \mathcal{N}(0, 1)$. In other words, we can write that

$$Z = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}, \quad or \quad Z = \begin{bmatrix} z_{1,1} & \dots & z_{1,100} \\ & \dots & \\ z_{n,1} & \dots & z_{n,100} \end{bmatrix}.$$

Thus it follows that

$$X_\gamma = \begin{bmatrix} x_1 + \gamma z_1 \\ \vdots \\ x_n + \gamma z_n \end{bmatrix}, \quad or \quad X_\gamma = \begin{bmatrix} x_{1,1} + \gamma z_{1,1} & \dots & x_{1,100} + \gamma z_{1,100} \\ & \dots & \\ x_{n,1} + \gamma z_{n,1} & \dots & x_{n,100} + \gamma z_{n,100} \end{bmatrix}.$$

Compared to 21 the two time series, with $\gamma = \sqrt{10}$, now look like the following

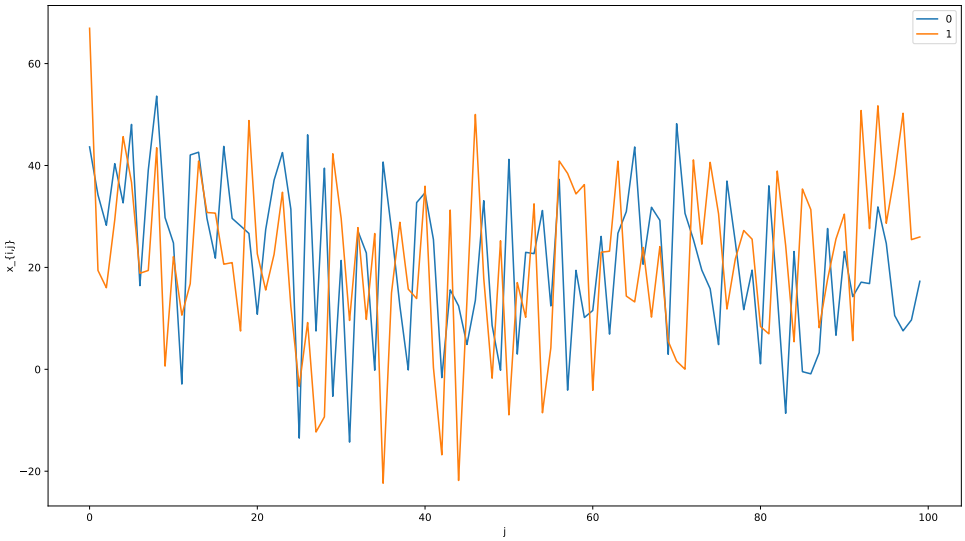


Figure 23: Picture of two timeseries, with different labels where $\gamma = \sqrt{10}$. The orange is label 1, whereas the blue is label 0.

In fact, doing a small analysis for this specific value of $\gamma = \sqrt{10}$. If we were to relabel the time series using our labeling function the time series corresponding to the blue line would actually get label 1 instead of 0. So for γ larger, we expect the classification problem to become tougher. Namely, we feed the algorithm data that contains noise. Thus to test the robustness of the Genetic Programming algorithm. We will let the algorithm extract a feature and compare how well this function compares to the 'real' labeling function. The new dataset we use will be defined in the following way

$$\hat{\mathcal{D}} = \{(x_1 + \gamma z_1, y_1), \dots, (x_n + \gamma z_n, y_n)\}.$$

Furthermore, we stress that the parameters for the algorithm have not been changed. We have only used the enhanced function set $\mathcal{F}$. We have tested various values of γ, and for every $\gamma \in [0, 10]$ the algorithm was able to extract the same function being

$$f(x_i) = m_{(0,100)}(x_i) = \frac{1}{100} \sum_{j=1}^{100} x_{i,j}.$$

After this, we tried more extreme values of γ. For example, the value of $\gamma = 40$ returned the function $h(x) = m(3, 100)$. So it did not quite find the true function. Nonetheless what we can take away from this experiment is that when the data is induced with some Gaussian noise the algorithm is still able to find the correct labeling function/splitting function. It does this by finding a function that isolates, or creates a split, on a specific subset of the dataset. This subset is given by the x_i for which it holds that

$$y_i = \mathbf{1}\left\{\frac{1}{100}\sum_{j=1}^{100} x_{i,j} + \gamma z_{i,j} > 21\right\}.$$

In other words, the time series for which relabeling would not differ from their original label. Hence this shows that the GP algorithm is able to still find rules on noisy data, as long as the noise is not too extreme and causes the input to be labeled differently. When we tried to train a Decision Tree just on the time series data we obtained an accuracy of 0.57. It shows that the GP algorithm can extract better features that really do benefit the Decision Tree performance-wise. Furthermore, this is a great example of how Decision Tree need feature transformations to have a good performance on time series data.

5.4 Rabobank data

5.4.1 Construction of the Rabobank data

The models are trained on different types of data. These types of data are commodity prices, meteorological data, and transactional data. For each category, these data are made out of time series for every client. So for client 1 who operates in the cattle market, we know what the prices of cattle have been over time. Then, based on time series we try to predict whether or not a client will go into default (or financial difficulty) in the next twelve months. First, all the clients are segmented based on their main North American Industry Classification System (NAICS) [52] activity. In short, NAICS can be thought of as giving the industry type a client will be belonging to. Examples of this would be CattleRanchingandFarming, OilseedandGrainFarming, etc. Then these industry types are grouped by industry sectors. CattleRanchingandFarming will belong to Animal Production and Aquaculture, whereas OilseedandGrainFarming will belong to Crop Production. How these groups are composed can be seen in Figure 24.

Industry Sector	Industry type
AnimalProduction Aquaculture	Aquaculture
	CattleRanchingandFarming
	HogandPigFarming
	OtherAnimalProduction
	PoultryandEggProduction
	SheepandGoatFarming
CropProduction	FruitandTreeNutFarming
	GreenhouseNurseryandFloriculureProduction
	OilseedandGrainFarming
	OtherCropFarming
	VegetableandMelonFarming
FishingHuntingandTrapping	Fishing
	HuntingandTrapping
	Logging
	TimberTractOperations
SupportActivitiesforAgriculture and Forestry	SupportActivitiesforAnimalProduction
	SupportActivitiesforCropProduction
	SupportActivitiesforForestry

Figure 24: Overview of NAICS segmentations with Industry Sector.

Furthermore, in addition to this segmentation, the clients will also be grouped in which rural area they operate. These rural areas are Australia, Brazil, Chile, Rabo Agrifinance (RAF), and New Zealand. Hence our final grouped segmentations will be Australia Animal Production Aquaculture, Brazil Crop Production, Australia Crop Production, etc. We will now explain how the time series are created. The time series are, as aforementioned, created using meteorological data and commodity price data. Consider the meteorological data. Suppose we have a certain area in which we have clients. For example, a region in New Zealand, this area is divided into a partition of smaller regions r_i, where each r_i has a middle point m_{r_i}. A lot of clients in the portfolio for which this EWS model is created are from New Zealand. Hence the choice for New Zealand.

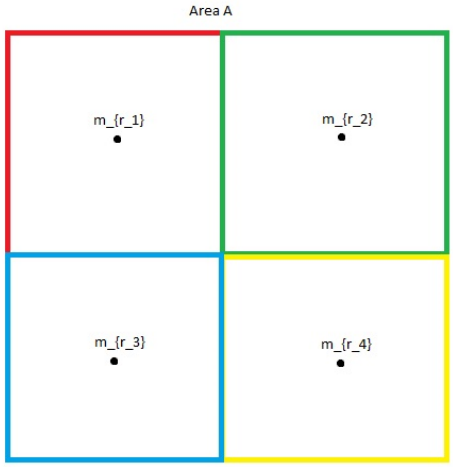


Figure 25: Example of a grid being divided up in different sub grids.

A partition of the area is created such that every client in the specific area belongs to exactly one grid/square. The middle points m_{r_i} correspond to coordinates given in longitude and latitude. Hence for the meteorological time series, every client will belong to a square r_i and will then be identified by the corresponding middle point m_{r_i}. Furthermore, we have a collection of time stamps which we call $\mathcal{T} = \{t_1, \ldots, t_n\}$. This collection (or set) of timestamps corresponds to all the time points for which we have the meteorological data. This is roughly data from 2012 to 2018. Hence the meteorological data looks like

$$\begin{bmatrix} m_{r_1} & t_1 & T2M_{t_1}^{m_{r_1}} & PET_{t_1}^{m_{r_1}} & PRECTOT_{t_1}^{m_{r_1}} & SPEI_{t_1}^{m_{r_1}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{r_1} & t_n & T2M_{t_n}^{m_{r_1}} & PET_{t_n}^{m_{r_1}} & PRECTOT_{t_n}^{m_{r_1}} & SPEI_{t_n}^{m_{r_1}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{r_n} & t_1 & T2M_{t_1}^{m_{r_n}} & PET_{t_1}^{m_{r_n}} & PRECTOT_{t_1}^{m_{r_n}} & SPEI_{t_1}^{m_{r_n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{r_n} & t_n & T2M_{t_n}^{m_{r_n}} & PET_{t_n}^{m_{r_n}} & PRECTOT_{t_n}^{m_{r_n}} & SPEI_{t_n}^{m_{r_n}} \end{bmatrix},$$

where for every middle point m_{r_i} we have, for every timestamp t_j, information about the meteorological circumstances. Typical dimensions of this dataset are, for all clients in New Zealand, 124925×91. First follows an explanation of what these values represent.

- T2M represents the average air temperature at 2 meters above the surface of the earth.
- PET is an index that measures the potential evapotranspiration. That is, the evaporation from the ground that would occur when a nearby crop has enough access to water. The evapotranspiration is modelled from the Thornthwaite equation. We will omit the location subscript for readability, keep in mind that all these values are location bound. For a month t_i we have that

  $$PET_{t_i} = PET(t_i) := 16 \left(\frac{L(t_i)}{12}\right) \left(\frac{N(t_i)}{30}\right) \left(\frac{10T(t_i)}{I(t_i)}\right)^{\alpha(t_i)},$$

  Where:

  - $L(t_i)$ is the average day length in hours for a month t_i.
  - $N(t_i)$ is the number of days in month t_i.
  - $T(t_i)$ is the average daily temperature in month t_i.
  - $\alpha(t_i) := \sum_{j=0}^{3} C_j \left(I(t_i)\right)^j$ where C_j are numerical constants.
  - $I(t_i) = \sum_{j=0}^{11} \left(\frac{T(t_i - j)}{5}\right)^{1.514}$.

  For more information and detail about the numerical constants, we refer to [53, 54].
- PRECTOT is exactly the total precipitation per month, which is calculated as the sum of precipitations per day in that specific month M.

- SPEI is an index measure for drought/water surplus in a month t_i. It is defined from the PET index and the monthly rainfall.

$$SPEI_{t_i} = SPEI(t_i) = \Gamma^{-1}\left(s, D(t_i)\right).$$

  Where

  - Γ^{-1} is the inverse regularised Gamma function.
  - s is a shape parameter determined by fitting a Gamma distribution to $D(t_i)$ for each location m_{r_i}
  - $D(t_i) = P(t_i) - PET(t_i)$ with $P(t_i)$ the total precipitation in month M.

  For more information about the SPEI index we refer to [55].

Now the prediction we are trying to make is, at a certain timepoint, if a client will go into default in the next 12 months. For each client, we have some information. We already discussed that each client k belongs to a middle point m_{r_i}, we also have a subset of timestamps $\mathcal{T}_k$ which consists of timestamps t_i^k where for each timepoint we know if a client goes into default within the next 12 months. This is visualized in the following way

$$\begin{bmatrix} \text{client} & \text{timestamp} & \text{default12} \\ 1 & t_1^1 & 0 \\ 1 & t_2^1 & 0 \\ 1 & t_3^1 & 1 \end{bmatrix}.$$

So from the matrix above, we know that client number 1 is not going into default in the next 12 months counting from t_1^1 and t_2^1, but when measured from t_3^1 client 1 does go into default in the next 12 months. Thus, from a classification standpoint, we want to label client 1 at timestamps t_1^1 and t_2^1 with a 0. Whereas, we want to label him with a 1 at timestamp t_3^1. Therefore to create a classification problem we need information for each client. This information will be the time series based on the meteorological data. We proceed in the following, let us return to considering client 1 at a certain timepoint t_j^1. For client 1 we know that they belong to a certain grid with middle point m_{r_k}. Furthermore, we have that a certain timestamp t_j^1 belongs to a timestamp in $\mathcal{T}$ say t_k. This is because $\mathcal{T}$ is the collection of all time stamps. Then we create the following feature vector for client 1 at time point t_k.

$$\left[\ T2M_{t_{k-24}}^{m_{r_k}},\ \cdots,\ T2M_{t_k}^{m_{r_k}},\ \cdots,\ SPEI_{t_{k-24}}^{m_{r_k}},\ \cdots,\ SPEI_{t_k}^{m_{r_k}}\ \right].$$

In words, this means that for every timestamp corresponding to a client in a certain area we create a time series going back $n = 24$ months for the specific middle point a client belongs to. So in this specific case $\left[\ T2M_{t_{k-24}}^{m_{r_k}},\ \cdots,\ T2M_{t_k}^{m_{r_k}}\ \right]$ is a time series based on the $T2M$ values of 24 consecutive months in the grid with middle point m_{r_k}. Each of those created time series, where the number of consecutive months can change, are then stacked after each other in the feature vector. Each feature vector then has a corresponding label based on the information for client 1 at timestamp t_i^1. For example in the case of t_1^1 the label of the feature vector would be 0. Then, as mentioned in the introduction of this thesis, new features will be created based on these time series. We will be applying the GP algorithm on a training dataset $\mathcal{D}$ which will consist of the aforementioned feature vectors for clients.

Remark 11. *We must note that since we have made a partition of the location such that every client belongs to a middle point m_{r_i} it has as a consequence that every client belonging to that middle point has the same time series and thus instances vector. That means that if clients have an overlapping collection of timestamps, the dataset will have instances that are exactly the same. However, it can be that the labels of these instances are different. Think of this as your neighbour farmer have financial probabilities in the same period whereas you have not. Hence instances that are the same should be labeled differently. So all clients belonging to the same grid with the same timestamp t will get the same output. All will be in default or all are not in default. Obviously, this is a serious problem. This problem can be circumvented by adding more client-specific data to the instances. However, at the time of running the algorithm this was not yet fully possible. For example, the commodity price data was not yet fully tested and had some calibration problems. This is why we ended up not using the commodity price data in our algorithm. Hence the results on this dataset should be treated carefully.*

5.4.2 Results on the data

For this subsection, we created a dataset based on meteorological data for the New Zealand animal production and aquaculture clients. This dataset was created based on the procedure described in Section 5.4.1. Thus for every client in the segmentation of New Zealand animal production and aquaculture, we have time series consisting of meteorological data at certain time points. In the total dataset, that is, before the split. There are 124925 rows of data, and all the data is labeled. For the labels, we have that 124187 of the rows have label 0, whereas 738 rows have label 1. This means that about 0.6% of the instances are going into default in the next 12 months. Imbalanced data is known to cause issues in various machine learning techniques. Not addressing for this would cause issues for the GP-algorithm. This can be seen by the impact of imbalanced data on the Gini fitness. Rabobank will also be using the algorithm on datasets that have a higher percentage of defaults. We expect the GP-algorithm on imbalanced data to pick up more noise in the created features. Unfortunately, while writing this thesis the datasets with a higher percentage of defaults was not yet available. Hence the data is very imbalanced, and thus we must use the weights given in Remark 9 on page 41. Furthermore, we note that the 70/30 split is created in such a way that a client can only belong to either of the two. The GP algorithm is used with our standard sets for $\mathcal{F}$ and $\mathcal{T}$. We do enlarge them so that we can add the function m to $\mathcal{F}$ as we did in Section 5.3. For running the algorithm the following parameters were used

parameter	**value**
max_gen	1000
initial_pop_size	1000
μ	100
λ	500
cxpb	0.9
mutpb	0.1
early_stopping	100
#_functions	31

Table 7: Overview of what we refer to as the standard parameters for our GP algorithm.

These parameters are based on the parameters in Table 1. We have increased the value for λ because we have more processing power in the dataiku environment. Using the parameters in Table 7 we create 31 new features. We choose 31 as that is the maximum amount of splits that occur in a Decision Tree of depth 4. Depth 4 is chosen because of the interpretability discussed in the next paragraph and from a run-time constraint. These features are then used to create two new datasets $\mathcal{D}'_{\text{train}}$ and $\mathcal{D}'_{\text{test}}$. Let f_i be equal to feature i created from the algorithm. Then we define

$$\mathcal{X}'_{\text{train}} := \begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_{31}(x_1) \\ \vdots & \vdots & \vdots & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_{31}(x_n) \end{bmatrix},$$

where x_i is equal to x_i from the training dataset. Now the training dataset

$$\mathcal{D}'_{\text{train}} = \{(\xi_1, y_1), \cdots, (\xi_n, y_1\},$$

where ξ_i is the i-th row from $\mathcal{X}'_{\text{train}}$. Similarly, the test dataset is created. Then a Decision Tree is trained on the train dataset, this Decision Tree is trained for various depths, namely depth 3, 4, and 5. These limits are chosen because a route in the Decision Tree must be interpretable. In general, it holds that the longer the route (which happens more often with a higher depth) the less interpretable the decision rules are. This means a longer route has more decisions. More decisions make the reason why a client gets a certain label more complicated. Hence they are less interpretable for the customer relations office. Since the run-time of the algorithm was high, 6+ hours, it was very tough to get statistical results. In all our runs the AUC-ROC accuracy of the Decision Trees created with the newly defined training set $\mathcal{D}'_{\text{train}}$ was at least as high as the results obtained before by Rabobank. For reference the ROC-AUC score of the Decision Tree created with expert-based features scored 0.656, whereas the Decision Tree created from the $\mathcal{D}'_{\text{train}}$ dataset had a score in the range of $0.657 - 0.703$. The results of our algorithm are not necessarily better, they are different. We noted very interesting behaviour when comparing the confusion matrix of the Decision Tree created by Rabobank and the Decision Tree that was eventually created by our algorithm. For this, we used a Decision Tree that scored 0.703, but the behaviour was similar between the runs

	predicted 1	**predicted** 0	**Total**
actually 1	39	83	122
actually 0	4054	18172	22226
Total	4093	18255	22348

(a) Confusion matrix for Rabobank DT.

	predicted 1	**predicted** 0	**Total**
actually 1	13	109	122
actually 0	859	24475	25334
Total	872	24584	25456

(b) Confusion matrix for the GP algorithm.

Figure 26: Overview of confusion matrices.

From Figure 26 it becomes clear what we mean by the word 'different' from the last paragraph. The fraction of clients that get triggered by the GP algorithm Decision Tree is about 1.4% vs about 1 % in the Rabobank Decision Tree. The difference is most likely due to run-to-run variance. Although, the GP algorithm Decision Tree achieves this precision while classifying way fewer clients as 1. This behaviour was constant over different runs of the GP algorithm. We think that this is because of dataiku doing under the hood optimizations to achieve optimal ROC-AUC scores. However, more research should be done if this is really a consequence of using the GP algorithm with dataiku. It could be beneficial to achieve higher precision while labeling fewer clients as a 1. Although, it can be considered to be a double-edged sword. If the GP algorithm Decision Tree would get used to flag clients, there is less pressure on the local office for checking every flagged client. On the contrary, more clients go unnoticed and so fewer clients might get the help they need. It is also worth noting that the original dataset on which we applied the GP algorithm had a few more entries compared to the dataset that was originally used by Rabobank. This is since the original datasets get updated with new data often. In addition to this, we note that to use the GP algorithm no features had to be created. However, the values in Table 7 had to be chosen before the runs. This can be compared with the choices Rabobank had to make while creating their features from the time series data.

Now that we have seen that the results of the Decision Tree created by the new algorithms were interesting we proceed with the features that the algorithm extracted. We first have to define some notation. For example, for the variable PET we have in total 12 months of data. Where PET_0, is the value at a certain place 12 months before the timestamp and PET_{11} is the value of the last month before the timestamp of the client. All other variables have 24 months of data. Some examples of the types of functions that got extracted by the algorithm

on the first split are

$$f(x) = \frac{-x_{12} - x_{39} + 2x_{42} + x_{61} - x_{74}}{-x_{12} + x_{38} + 2x_{39} + x_{63}},$$
$$g(x) = \frac{x_{74} m(62, 64)}{x_{12} m(62, 64) - x_{54} x_8},$$
$$h(x) = x_{47} + x_{49} - x_{54} - x_{67} + x_{70}.$$

We stress again that these functions were obtained from different runs. We highlight the first split because it is the most important split of the Decision Tree. Namely, all instances have to go through this decision rule. We have consistently seen that the feature importance of the first 3 functions extracted by the GP algorithm has accounted for 50% of the feature importance. Remember, this means that 50% of the total decrease of Gini has been reached after only 3 decision rules. Therefore we have added a section with various examples of the first 3 features that were extracted by the GP algorithm in Appendix D. The cut-off of three features was chosen because of its correspondence to 50% of the total decrease in Gini. Moreover, we think it gives an interesting snapshot of the types of functions that get extracted. Now for the interpretation of the extracted feature, we highlight the function $h(x)$. Using Table 8 we can read which meteorological variables correspond to a certain x_i. We see that $h(x)$ consists of two different components which are added together. We see that a part where

$$h_1(x) = SPEI_{10} + SPEI_{12} - SPEI_{17},$$

and a part where

$$h_2(x) = T2M_9 - T2M_6.$$

In the case of $h(x)$ the $SPEI$ value of 7 months before the time of prediction, is subtracted from the sum of two $SPEI$ values at 12 months and 14 months before the prediction. In addition to this, the increase (or decrease) of the temperature at 2 meters is also evaluated. This evaluation takes place over the 15th and 18th months before the prediction. This procedure of interpreting can then be repeated for every function extracted by the algorithm. This helps gain more insight into what functions are interesting for modelling on the time series data. The finding of the function $h(x)$ is particularly interesting when we consider what functions Rabobank used in constructing the Decision Tree. In their Decision Tree the first split was created on the function $T2M_{\text{diff}_4}$ which corresponds to the difference between the latest $T2M$ value and 4 months before. The $T2M$ values are also occurring in the functions $f(x)$ and $g(x)$. Thus Rabobank might want to explore a more in-depth analysis on features containing $T2M$. For example, looking at the lagged differences.

For the last part of this section, we want to highlight why the functions are so different. First of all, this is due to the fact that the GP search algorithm is stochastic. Secondly, it is due to the data. The functions $f(x), g(x)$ and $h(x)$ all have very similar fitness. They achieved a similar quality of split while using, mostly, different variables. This leads us to conclude that there are many different ways to achieve these types of splits. This however makes sense as we have a lot of information in the data to make a function to use for the split.

6 Future Research

At the time of writing this thesis, we only had access to meteorological data. In the future, one could research the use of the algorithm on a combination of multiple types of time series data. For example, the transaction data or commodity price data. We expect that this would benefit the algorithm in different ways. More information on clients specific would alleviate the problem where multiple clients had the same feature vector. It could also be that this addition would help with the inconsistency of the features extracted. Hence why exploring this addition is interesting for future research.

The addition of the time series discussed above could introduce a run-time problem. In its current state, the algorithm run-time scales with the number of columns the training dataset has. Of course, the rows also influence the run-time of the algorithm. However, way less than the columns. This is due to how DEAP is implemented in Python. One of the future research that could be done on this subject would be to make DEAP more efficient for dataset with lots of columns. Furthermore, one of the most taxing parts of the algorithm is the computation of the fitness for individuals. Currently, this is implemented to use a CPU. It would be interesting to research whether or not the fitness evaluation can be offloaded to a GPU. Then more individuals can be evaluated in parallel, which would improve the run-time of the algorithm.

As we have mentioned various times in the thesis, the features are originally created using expert-based knowledge. In the future, it would be interesting to be able to seed the initial population using this expert-based knowledge. Let us say that a data scientist thinks that a certain combination of features is interesting, it would be interesting to see what would happen when the GP algorithm starts with a population that is heavily biased towards these combinations of features. In this way, the search of the GP algorithm can be combined with expert knowledge. Predetermining what population a GP algorithm will start with is called seeding. Seeding a population based on expert knowledge would be an interesting future research question.

In Section 3.5.5 we introduce the NSGA-II selection method. We introduced this because the tournament selection method was prone to bloat. This was shown in Section 5.2.1. In this Section, we also saw that the tournament selection had high diversity over multiple generations. It would be interesting to see if the occurrence of bloat can be limited in tournament selection. There are various ways this can be done. One could penalize the fitness if the individual gets too big. Also one could limit the size of all individuals appearing in the evolution. What would be the best way to apply these size limiting procedures? How would this influence the features extracted by the algorithm? These are all questions that can be answered in future research.

In Section 5 we have seen that the GP algorithm each run extracts different features. Hence the high standard deviation in the artificial tests and the different functions found in the Rabobank dataset. It would be interesting to see how the GP algorithm can be made more consistent. For example, how would one extract the most prominent functions from 100 different runs? In most datasets we expect this to return more consistent features. Doing 100

different runs would take a lot of time, so either the run-time of the algorithm needs to be improved, or a different approach has to be taken. One could also do the runs in parallel, but then often a lot of machines are needed.

We think that there is still a lot of research to be done before the GP algorithm we introduced in this thesis can be used on its own. For now, we would advise using the algorithm to get an idea of functions and features that benefit classification in a DT.

7 Conclusion

While creating predictive features for time series, Rabobank was wondering if their approach was complete, in the sense that they did not overlook any important predictive features. To answer this question we developed a search algorithm that would search for good predictive features. Our approach was based on our knowledge of the concept of Decision Tree introduced in Section 2. Together with the concept of Genetic Programming introduced in Section 3 we created a search algorithm which was introduced in Section 4. In Section 4 we also highlighted the most important choice we had to make for the algorithm. We concluded that a fitness based on the Gini was less likely to overfit compared to the Fisher fitness. Which was beneficial for the interpretability of the extracted features. The first part of Section 5 functioned as an exploration of how the algorithm behaved in certain circumstances. It was also a setup for the second part of the Section. This second part was written in Section 5.4.2. In Section 5.4.2 we ran the algorithm on data provided by Rabobank. This again highlighted the issues corresponding to a stochastic search algorithm. The extracted features are different for each run. However, the functions that were extracted by the algorithm gave an indication of types of features that are interesting to explore more in-depth. Where we specifically mentioned that features with meteorological variable $T2M$ could be explored more in-depth as they came up multiple times in the extracted features. Our created algorithm can be used to create new features for many types of data, not only time series. We conclude that the algorithm is useful in determining which features of the original dataset are of interest and can help indicate how the original features relate to each other. The algorithm can be used to automatically create features that can be used to create better-performing Decision Trees. Furthermore, it can serve as a guide for a data scientist or modeller in the feature extraction part of modelling to create better predictive models.

References

[1] Padmavathi Kora and Priyanka Yadlapalli. Crossover operators in genetic algorithms: A review. *International Journal of Computer Applications*, 162(10), 2017.

[2] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[3] Dominic P Searson, David E Leahy, and Mark J Willis. Gptips: an open source genetic programming toolbox for multigene symbolic regression. In *Proceedings of the International multiconference of engineers and computer scientists*, volume 1, pages 77–80. Citeseer, 2010.

[4] Kourosh Neshatian, Mengjie Zhang, and Peter Andreae. A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation*, 16(5):645–661, 2012.

[5] Mohammed Alweshah, Omar A Alzubi, Jafar A Alzubi, and Sharihan Alaqeel. Solving attribute reduction problem using wrapper genetic programming. *International Journal of Computer Science and Network Security (IJCSNS)*, 16(5):77, 2016.

[6] Mohammed Muharram and George D Smith. Evolutionary constructive induction. *IEEE transactions on knowledge and data engineering*, 17(11):1518–1528, 2005.

[7] Mr Brijain, R Patel, MR Kushik, and K Rana. A survey on decision tree algorithm for classification. 2014.

[8] Hong Guo and Asoke K. Nandi. Breast cancer diagnosis using genetic programming generated feature. *Pattern Recogn.*, 39(5):980–987, May 2006.

[9] Kourosh Neshatian and Mengjie Zhang. Genetic programming and class-wise orthogonal transformation for dimension reduction in classification problems. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming*, pages 242–253, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] Afsaneh Mahanipour and Hossein Nezamabadi-pour. A multiple feature construction method based on gravitational search algorithm. *Expert Systems with Applications*, 127:199–209, 2019.

[11] M. L. Raymer, W. F. Punch, E. D. Goodman, and L. A. Kuhn. Genetic programming for improved data mining: Application to the biochemistry of protein interactions. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 375–380, Cambridge, MA, USA, 1996. MIT Press.

[12] Pedro G. Espejo, Sebastián Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *Trans. Sys. Man Cyber Part C*, 40(2):121–144, March 2010.

[13] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[14] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009.

[15] Wikipedia contributors. Korfball — Wikipedia, the free encyclopedia, 2021. [Online; accessed 16-August-2021].

[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[17] Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[18] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[19] William B Langdon and Riccardo Poli. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

[20] Michael O'Neill. Riccardo poli, william b. langdon, nicholas f. mcphee: a field guide to genetic programming, 2009.

[21] Spencer CH Barrett and Spencer Charles Hilton Barrett. *Major evolutionary transitions in flowering plant reproduction*. University of Chicago Press, 2008.

[22] Nigel J Dimmock, Andrew J Easton, and Keith N Leppard. *Introduction to modern virology*. John Wiley & Sons, 2015.

[23] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[24] Karl Sims. Artificial evolution for computer graphics. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, 1991.

[25] Eva Alfaro-Cid, Euan W McGookin, David J Murray-Smith, and Thor I Fossen. Genetic programming for the automatic design of controllers for a surface ship. *IEEE transactions on intelligent transportation systems*, 9(2):311–321, 2008.

[26] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[27] Santosh S Rathore and Sandeep Kumar. Predicting number of faults in software system using genetic programming. *Procedia Computer Science*, 62:303–311, 2015.

[28] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[29] Frank Neumann and Carsten Witt. *Stochastic Search Algorithms*, pages 21–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[30] Wolfgang Banzhaf, A Lakhtakia, and RJ Martin-Palma. Evolutionary computation and genetic programming. *Engineered Biomimicry*, pages 429–447, 2013.

[31] T. Baeck, D.B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press, 2018.

[32] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, 2020.

[33] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pages 123–137. Springer International Publishing, 2016.

[34] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[35] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

[36] John H. Holland. Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314, July 1962.

[37] R. M. Friedberg. A learning machine: Part i. *IBM J. Res. Dev.*, 2(1):2–13, January 1958.

[38] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[39] Markus Brameier, Wolfgang Banzhaf, and Wolfgang Banzhaf. *Linear genetic programming*, volume 1. Springer, 2007.

[40] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[41] Laurent Alonso and Rene Schott. *Random generation of trees: random generators in computer science*. Springer Science & Business Media, 2013.

[42] Thalles Santos Silva. An illustrative introduction to fisher's linear discriminant. *https://sthalles.github.io*, 2019.

[43] Yongsheng Fang and Jun Li. A review of tournament selection in genetic programming. In *International Symposium on Intelligence Computation and Applications*, pages 181–192. Springer, 2010.

[44] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[45] Binh Tran, Bing Xue, and Mengjie Zhang. Genetic programming for multiple-feature construction on high-dimensional classification. *Pattern Recognition*, 93:404–417, 2019.

[46] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[48] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 13: receiver operating characteristic curves. *Critical care*, 8(6):1–5, 2004.

[49] I. Guyon. Design of experiments for the nips 2003 variable selection benchmark. 2003.

[50] Anuradha Purohit and Narendra S Choudhari. Code bloat problem in genetic programming. 2013.

[51] Florian Mormann, Ralph G. Andrzejak, Christian E. Elger, and Klaus Lehnertz. Seizure prediction: the long and winding road. *Brain*, 130(2):314–333, 09 2006.

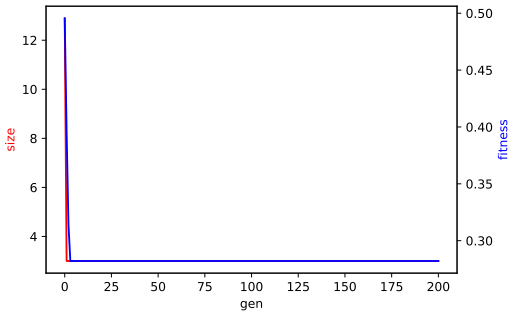
[52] John B Murphy. Introducing the north american industry classification system. *Monthly Lab. Rev.*, 121:43, 1998.

[53] James Adams. climate_indices, an open source python library providing reference implementations of commonly used climate indices, may 2017–.

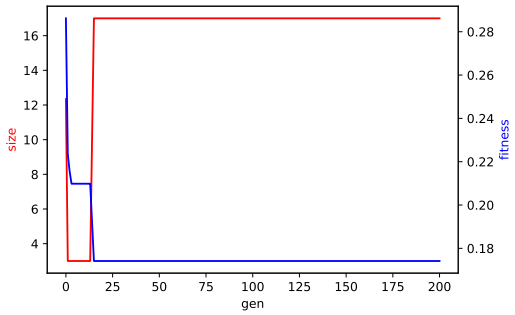
[54] Charles Warren Thornthwaite. An approach toward a rational classification of climate. *Geographical review*, 38(1):55–94, 1948.

[55] Sergio M. Vicente-Serrano, Santiago Beguer?a, and Juan I. L?pez-Moreno. A multiscalar drought index sensitive to global warming: The standardized precipitation evapotranspiration index. *Journal of Climate*, 23(7):1696 – 1718, 2010.

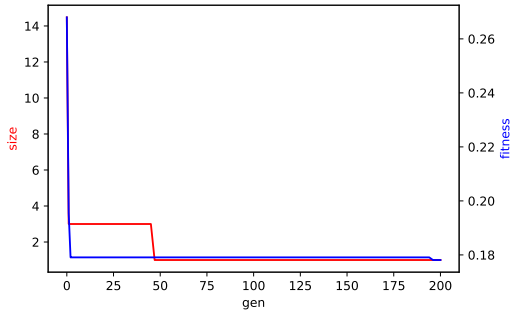
A Appendix on Selection methods



(a) First feature fitness vs size.

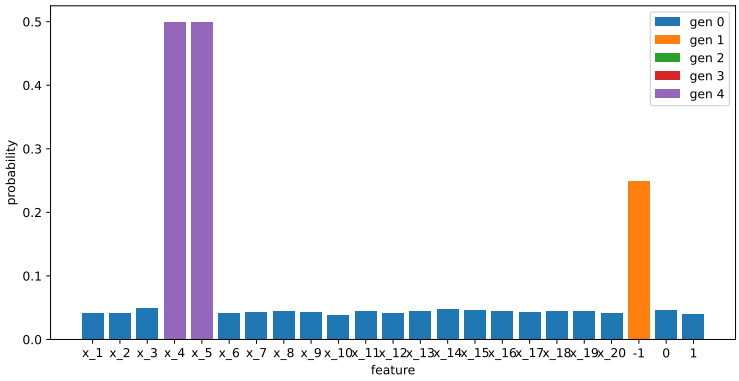


(b) Second feature fitness vs size.

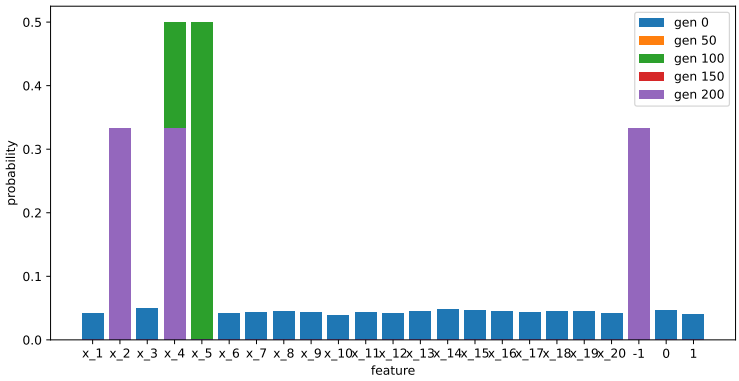


(c) Third feature fitness vs size.

Figure 27: Second run for 'selbest' method.



(a) Occurrence of features in generations 0,1,2,3,4.



(b) Occurrence of features in generations 0,50,100,150,200.

Figure 28: Visualization of the low diversity of 'selBest' in a second run.

B Appendix images from Section 5.2.2

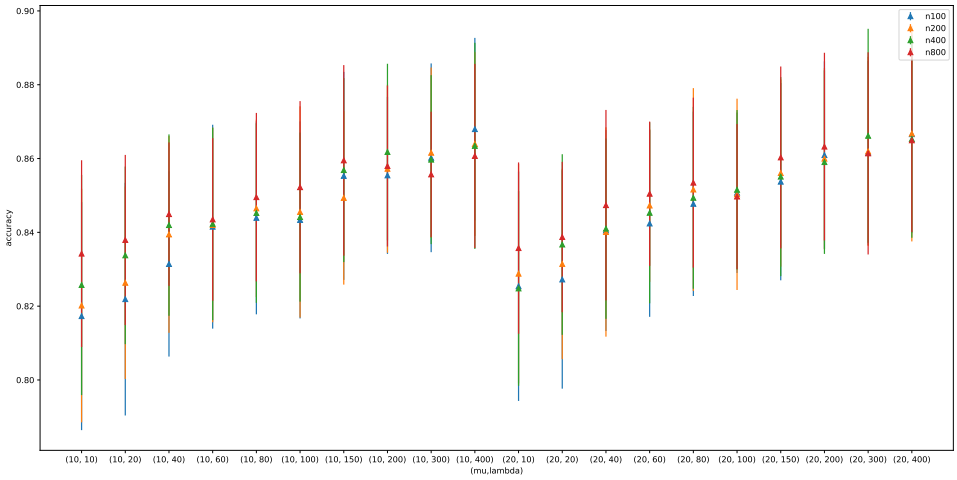


Figure 29: Accuracy of the GP algorithm on 20 features for the first 20 combinations.

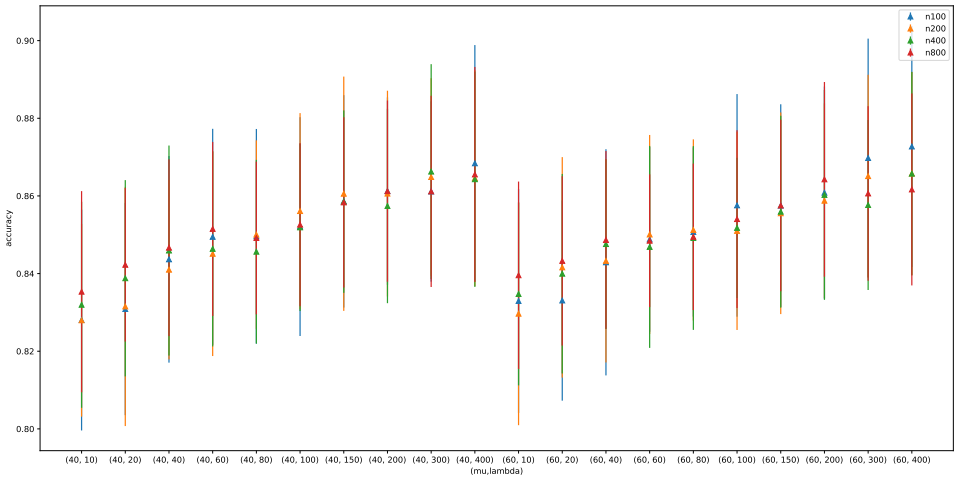


Figure 30: Accuracy of the GP algorithm on 20 features for combinations 20-40.

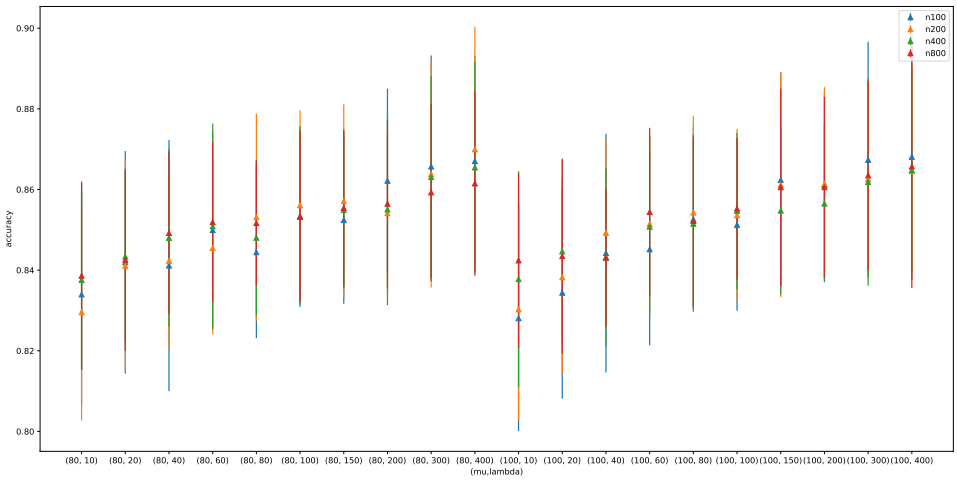


Figure 31: Accuracy of the GP algorithm on 20 features for combinations 40-60.

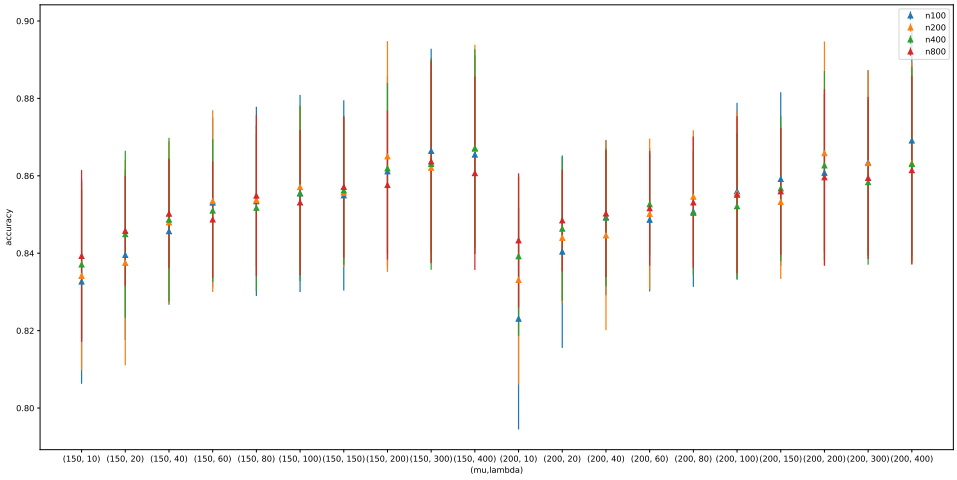


Figure 32: Accuracy of the GP algorithm on 20 features for combinations 60-80.

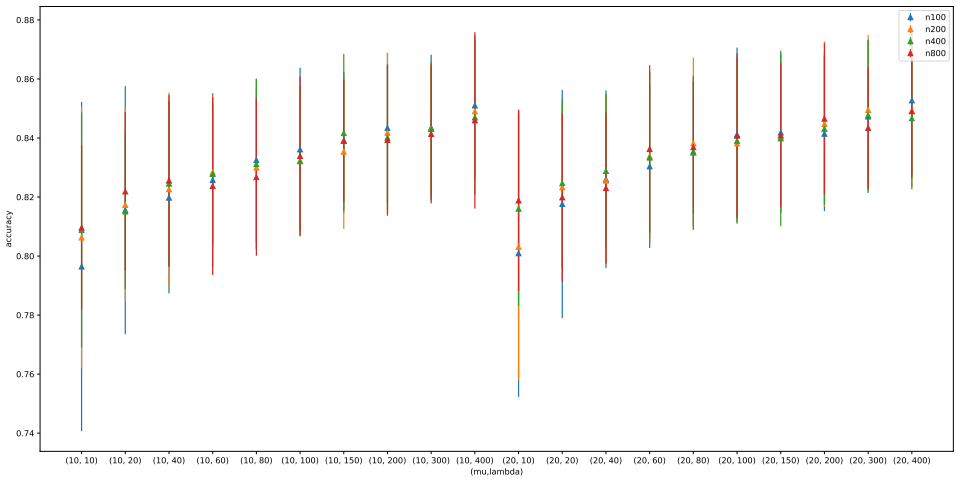


Figure 33: Accuracy of the GP algorithm on 40 features for the first 20 combinations.

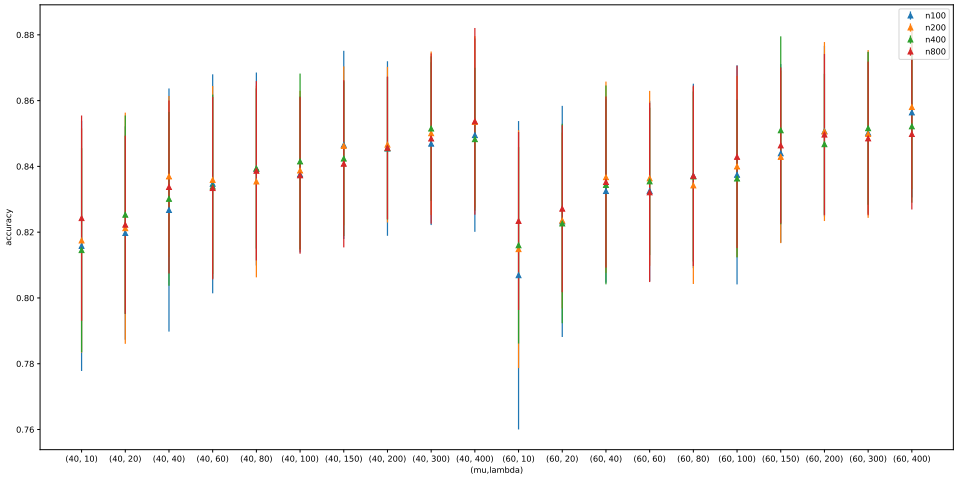


Figure 34: Accuracy of the GP algorithm on 40 features for combinations 20-40.

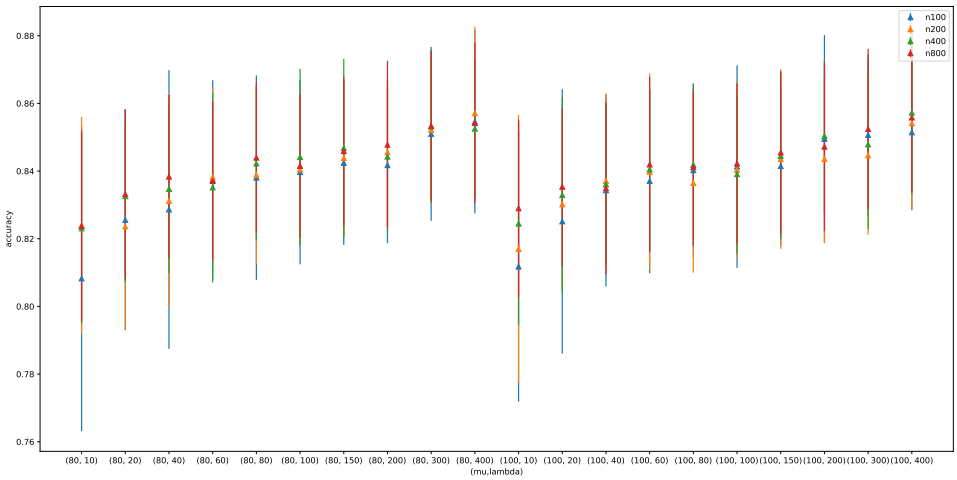


Figure 35: Accuracy of the GP algorithm on 40 features for combinations 40-60.

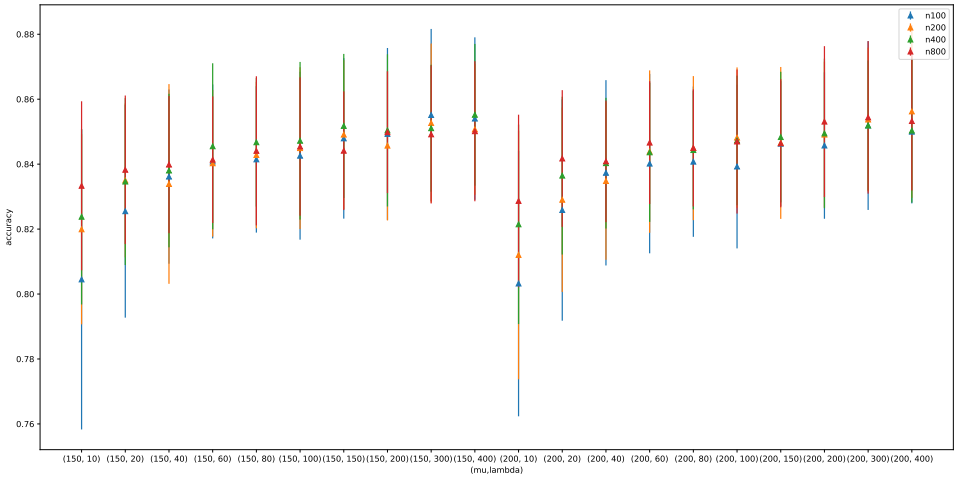


Figure 36: Accuracy of the GP algorithm on 40 features for combinations 60-80.

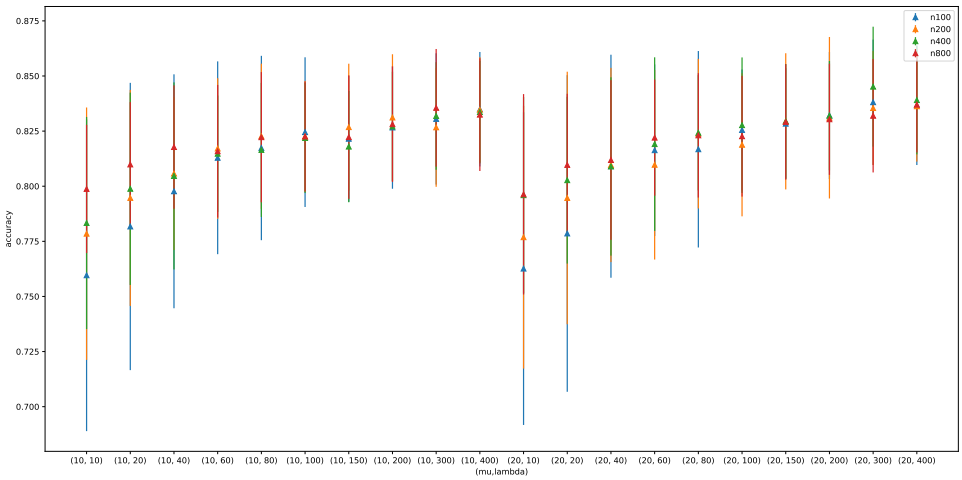


Figure 37: Accuracy of the GP algorithm on 100 features for the first 20 combinations.

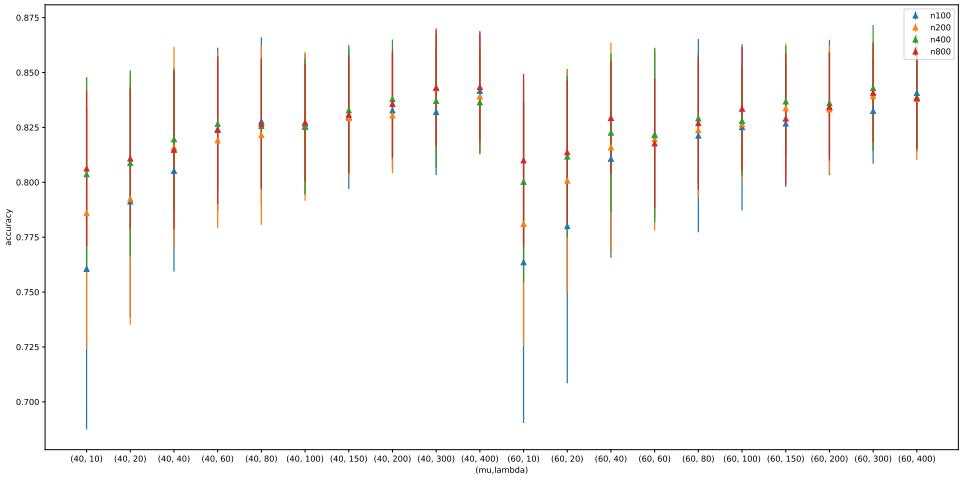


Figure 38: Accuracy of the GP algorithm on 100 features for combinations 20-40.

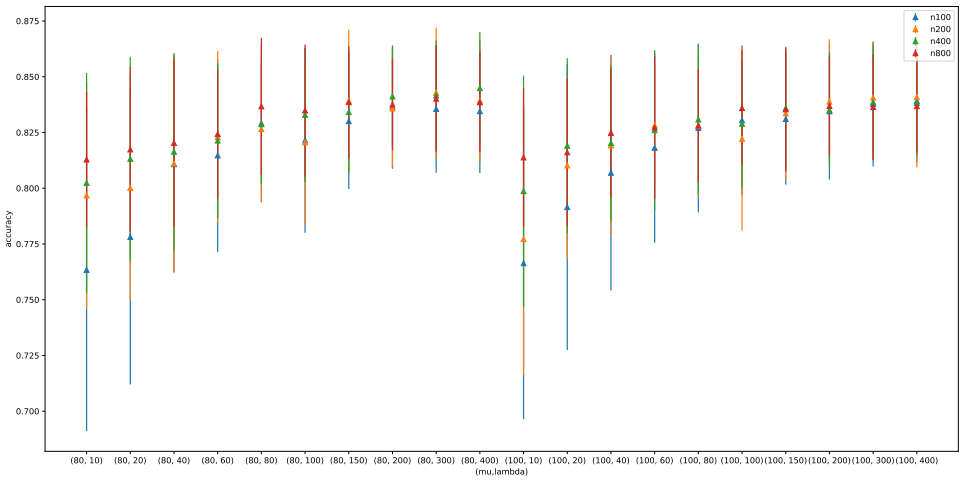


Figure 39: Accuracy of the GP algorithm on 100 features for combinations 40-60.

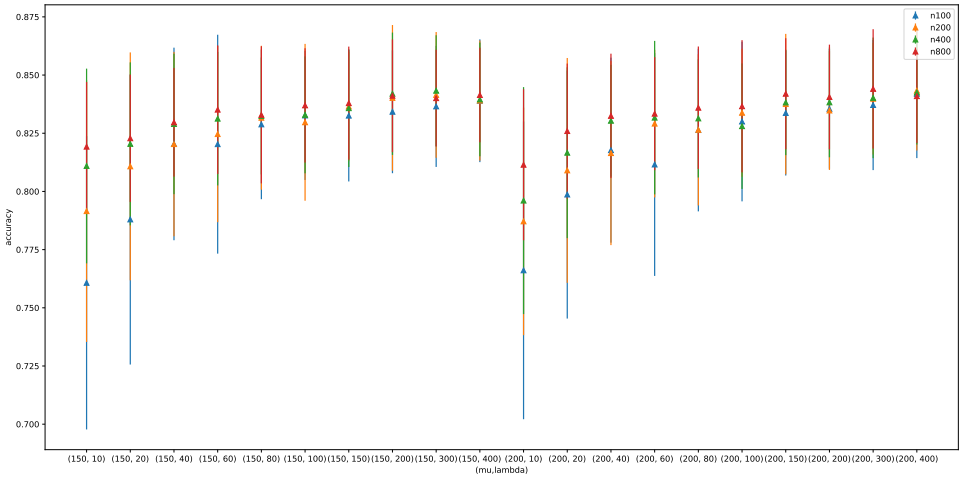


Figure 40: Accuracy of the GP algorithm on 100 features for combinations 60-80.

C Overview of name conversion

theenumivcc

name	variable
PET_0	x_1
PET_1	x_2

PET_2	x_3
PET_3	x_4
PET_4	x_5
PET_5	x_6
PET_6	x_7
PET_7	x_8
PET_8	x_9
PET_9	x_{10}
PET_10	x_{11}
PET_11	x_{12}
$PRECTOT\_0$	x_{13}
$PRECTOT\_1$	x_{14}
$PRECTOT\_2$	x_{15}
$PRECTOT\_3$	x_{16}
$PRECTOT\_4$	x_{17}
$PRECTOT\_5$	x_{18}
$PRECTOT\_6$	x_{19}
$PRECTOT\_7$	x_{20}
$PRECTOT\_8$	x_{21}
$PRECTOT\_9$	x_{22}
$PRECTOT\_10$	x_{23}
$PRECTOT\_11$	x_{24}
$PRECTOT\_12$	x_{25}
$PRECTOT\_13$	x_{26}
$PRECTOT\_14$	x_{27}
$PRECTOT\_15$	x_{28}
$PRECTOT\_16$	x_{29}
$PRECTOT\_17$	x_{30}
$PRECTOT\_18$	x_{31}
$PRECTOT\_19$	x_{32}
$PRECTOT\_20$	x_{33}
$PRECTOT\_21$	x_{34}
$PRECTOT\_22$	x_{35}
$PRECTOT\_23$	x_{36}
$SPEI\_0$	x_{37}
$SPEI\_1$	x_{38}
$SPEI\_2$	x_{39}
$SPEI\_3$	x_{40}
$SPEI\_4$	x_{41}
$SPEI\_5$	x_{42}
$SPEI\_6$	x_{43}
$SPEI\_7$	x_{44}
$SPEI\_8$	x_{45}

$SPEI\_9$	x_{46}
$SPEI\_10$	x_{47}
$SPEI\_11$	x_{48}
$SPEI\_12$	x_{49}
$SPEI\_13$	x_{50}
$SPEI\_14$	x_{51}
$SPEI\_15$	x_{52}
$SPEI\_16$	x_{53}
$SPEI\_17$	x_{54}
$SPEI\_18$	x_{55}
$SPEI\_19$	x_{56}
$SPEI\_20$	x_{57}
$SPEI\_21$	x_{58}
$SPEI\_22$	x_{59}
$SPEI\_23$	x_{60}
$T2M\_0$	x_{61}
$T2M\_1$	x_{62}
$T2M\_2$	x_{63}
$T2M\_3$	x_{64}
$T2M\_4$	x_{65}
$T2M\_5$	x_{66}
$T2M\_6$	x_{67}
$T2M\_7$	x_{68}
$T2M\_8$	x_{69}
$T2M\_9$	x_{70}
$T2M\_10$	x_{71}
$T2M\_11$	x_{72}
$T2M\_12$	x_{73}
$T2M\_13$	x_{74}
$T2M\_14$	x_{75}
$T2M\_15$	x_{76}
$T2M\_16$	x_{77}
$T2M\_17$	x_{78}
$T2M\_18$	x_{79}
$T2M\_19$	x_{80}
$T2M\_20$	x_{81}
$T2M\_21$	x_{82}
$T2M\_22$	x_{83}
$T2M\_23$	x_{84}

Table 8: Overview of features and corresponding x_i.

D More functions extracted by the algorithm

For the second split various functions were

$$
\begin{aligned}
f_2(x) &= x_{62} - x_{73} - m(47, 48), \\
g_2(x) &= m(67, 74) - m(61, 73), \\
h_2(x) &= x_3 - x_6 - \frac{x_2}{x_{63}} - \frac{x_3}{x_{61}}.
\end{aligned}
$$

The functions correspond each to $f(x), g(x)$ and $h(x)$ being their second split. For the third split the algorithm obtained

$$
\begin{aligned}
f_3(x) &= \frac{x_6 + x_{37} + 2x_{51} + x_{57} + 4 * x_{78}}{x_{57} + x_{65}}, \\
g_3(x) &= m(44, 51)m(75, 80), \\
h_3(x) &= \left(x_{20} + x_{24}\left(x_{56} + x_{77}\right)\right) \frac{\left(x_{62} - x_{75}\right)}{x_{24}}.
\end{aligned}
$$

Again various of these functions can be interpreted and give an insight in what meteorological variables are interesting to explore more in-depth.