INTERPRETABLE ARTIFICIAL INTELLIGENCE USING NONLINEAR DECISION TREES

By

Yashesh Deepakkumar Dhebar

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Mechanical Engineering – Doctor of Philosophy

2020

ABSTRACT

INTERPRETABLE ARTIFICIAL INTELLIGENCE USING NONLINEAR DECISION TREES

By

Yashesh Deepakkumar Dhebar

The recent times have observed a massive application of artificial intelligence (AI) to automate tasks across various domains. The back-end mechanism with which automation occurs is generally black-box. Some of the popular black-box AI methods used to solve an automation task include decision trees (DT), support vector machines (SVM), artificial neural networks (ANN), etc. In the past several years, these black-box AI methods have shown promising performance and have been widely applied and researched across industries and academia. While the black-box AI models have been shown to achieve high performance, the inherent mechanism with which a decision is made is hard to comprehend. This lack of interpretability and transparency of black-box AI methods makes them less trustworthy. In addition to this, the black-box AI models lack in their ability to provide valuable insights regarding the task at hand. Following these limitations of black-box AI models, a natural research direction of developing interpretable and explainable AI models has emerged and has gained an active attention in the machine learning and AI community in the past three years. In this dissertation, we will be focusing on interpretable AI solutions which are being currently developed at the Computational Optimization and Innovation Laboratory (COIN Lab) at Michigan State University. We propose a nonlinear decision tree (NLDT) based framework to produce transparent AI solutions for automation tasks related to classification and control. The recent advancement in non-linear optimization enables us to efficiently derive interpretable AI solutions for various automation tasks. The interpretable and transparent AI models induced using customized optimization techniques show similar or better performance as compared to complex black-box AI models across most of the benchmarks. The results are promising and provide directions to launch future studies in developing efficient transparent AI models.

Dedicated to my *parents* and elder brother *Paurav*.

ACKNOWLEDGEMENTS

The PhD journey for me has been really fulfilling and transformational and I would be like to acknowledge those who have played instrumental role during this endeavour. I thought that writing this *Acknowledgements* section of my dissertation would be easy and quick, however its not the case. While I was typing this, I came to a realization that this expression of gratitude would be very limited and always incomplete for certain people including my parents, my brother Paurav, my friends with whom I spent most of my PhD duration with and Prof. Kalyanmoy Deb.

The role of my family has been extremely important and encouraging in bringing me to the point I am at right now. Contributions and sacrifices made by my father Mr. Deepak Dhebar and my mother Mrs. Deepika Dhebar are among the things which cannot be quantified and most of which I believe are still outside my conscious realm. If it would not have been my brother Paurav, who spotted my technical aptitude and envisioned me getting into the IIT, my trajectory of life would be have been completely different. The vision, the support and the encouragement provided to me by my family has played a big part in turning this dream into reality.

I would like to thank my friends who I see as my family far from my hometown, the friends with whom I stayed and created a permanent bond with during my PhD – thank you Tarang, Swati, Mayank, Kokil, Ashish, Saptarshi, Sabhyasachi, Thrilok and Vikram Prajapati. I would like to also thank Abhinav and Kamala for showing their concern and warmth and being like elder siblings in the town. The stay has been made equally joyous because of the indoor board games amidst the harsh weather of Michigan and I would like to thank Kanchan, Aritra, Rahul and Bakul for being the part of the clan.

I would like to thank Pratap Bhanu Solanki for helping me during my transition from IIT Kanpur to Michigan State University. I would like to thank Nilay Kant for his valuable inputs during my recent job search and providing me with some fundamental insights across numerous aspects during my PhD.

I was really fortunate to have great lab-mates as my colleagues at the Computational Optimiza-

tion and Innovation Laboratory. I got to learn a lot from them. Thank you Abhinav, Haitham, Proteek, Rayan, Julian, Zhichao, Khaled, Airuddh, Ali, Abhiroop, Yash, Shuvei, Mohamed and visiting scholars viz. Ankur, Flavio, Pablo, Sukrit, Sparsh and Hemant. I would like to thank Mrs. Debjani Deb for organizing lab gatherings and enriching the social culture of the lab.

It was indeed my pleasure to interact with Prof. Ranjan Mukherjee and Mrs. Moushumi Mukherjee and am thankful to them for introducing me and involving me in the Durga Pooja festivals and other social gatherings. It was indeed great to experience the Indian traditional culture on the foreign land.

I would like to thank Prof. Bhaskar Dasgupta and Prof. Bishakh Bhattacharya of IIT Kanpur and Prof. G. K. Ananthasuresh of IISc Bangalore who wrote my recommendation letter for PhD. I would like to thank Prof. Niraj Sinha, Prof. Anindya Chatterjee and Prof. Harish Karnick of IIT Kanpur for giving valuable advice regarding how to choose a career after bachelors and what to consider while opting for a PhD. I still remember this line from Prof. Chatterjee's homepage and it reads: *The PhD advisor is more than merely a source of information and funding. Ideally, your PhD advisor will profoundly influence not only how you view your subject, but how you view the world.* I could experience this my case. Working under Prof. Kalyanmoy Deb shaped me fundamentally and his guidance showed me the way to conduct the life: both professional and personal.

Thank you Prof. Deb for everything and thank you Prof. Erik Goodman, Prof. Ronald Averill and Dr. Vishnu Boddeti for being the part of my PhD committee and for providing valuable feedback and suggestions in regards to my research.

Warm Regards,

Yashesh Dhebar

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

THE NEED AND THE START

The year was 2016 and our lab was offered with a project from the Dow Chemical Company. The goal of that project was to develop an automation system to automatically predict the harmonized tariff schedule code (HS-code) given the chemical recipe of a product which the Dow Chemical Company imports or exports. Prof. Deb, Prof. Goodman and I operated as a team from MSU and collaborated with concerned experts from the Dow who were involved in manually solving the task of HS-code assignment. At the end of the project, we were able to successfully develop an automation system to predict the HS-codes of chemical products with a human-level accuracy. This work attracted attention from various corporate organizations and is currently under the process of getting patented. During the process of developing a classifier, the team at Dow used to casually intervene into those slides where I used to show them how the AI doing HS-code prediction looks like. They were intrigued by the fact that the task being handled manually at Dow was now getting done by the AI, with the same performance accuracy as that of manual HS-code assignment. More than that, they were interested to know *how* the AI is thinking in the background and is coming up with a HS-code for a supplied chemical product. The AI developed however was fairly complicated and no such human interpretable logic could be borrowed through the *visual inspection* of AI, but we got the signal that *a human mind is simply not satisfied with answers an AI is providing, and it longs to understand "**why**" the thing worked.*

The implicit signal to develop an interpretable AI from Dow was made explicit a few months later. Our lab was approached by General Motors (GM), which wanted to develop a classifier to distinguish between *good* and *bad* designs. The task here was to develop a classifier which could be *read* by design engineers at GM. The *readable* AI would then serve as a tool to develop better insights regarding the design process and could be used as a *recipe* by design engineers to develop and innovate future car designs.

Later, a similar problem of developing an explainable AI was floated to us by Ford Motors,

where the task was to decipher the complicated logic learnt by a deep neural network (DNN) which is controlling a car for autonomous driving. The research presented in this dissertation is a byproduct of the process we went through in answering the questions laid out by two automotive giants. It has triggered many possibilities and we believe that foundation set by our work would lead to a further advancement in the field of interpretable and explainable AI.

1.1 Our Work

The field of interpretable and explainable AI has received active attention in the past four years. Broadly speaking, the interpretable AI approaches are categorized into two main groups

- Intrinsic (or model based) and
- Post-hoc.

Within the framework of *intrinsically* interpretable AI, the AI model itself is transparent and interpretable. The task of generating a less complex AI is generally realized using decision trees, rule-sets [2, 3] or additive models [4, 5, 6]. The aim here is to express an AI (or a subpart of an AI) in as simple format as possible.

The post-hoc interpretable methods are aimed towards analyzing the behavior of an already trained AI, mostly using visualization techniques such as partial dependence plots, histogram plots, heat-maps, attention maps, etc. A more comprehensive explanation to these is provided in [7, 8].

In this dissertation, we focus at developing an intrinsically interpretable AI solutions in form of a *nonlinear decision trees* (NLDT). The main criteria we consider in our research is to reduce the complexity of the transparent AI while ensuring it has a good performance for the machine learning task under consideration. Additionally, the approach being developed assumes that the features (or attributes) in a given automation task are interpretable. The optimization algorithm is then developed to induce an AI which is visually simple, involves less number of terms or functionals and can be expressed in a format which is humanly comprehensible.

The dissertation is organized as follows. First, a birds-eye view of the overall interpretable AI framework is provided in Chapter 2. In Chapter 3, we formally introduce the algorithm to derive NLDT for classification problems. Next, the approach developed to induce NLDT in Chapter 3 is extended to solve control problems involving discrete actions. In the same chapter, a reinforcement learning algorithm in form of *closed-loop training* is proposed to enhance the control performance of NLDT. The approach to arrive at interpretable controllers is improvised and tested for scalability on custom designed benchmarks in Chapter 5. A methodology to extend the concept of NLDT to solve regression and continuous control problems is discussed in Chapter 6. Concluding remarks and possible directions for future work are provided in Chapter 7.

CHAPTER 2

A HIGH LEVEL VIEW OF OUR INTREPRETABLE AI MODEL

Before we dig into the details, I would like to provide you with a bird's eye view of the overall approach. The interpretable and explainable AI we are developing assumes the framework of a Nonlinear Decision Tree (NLDT). A high-level perspective of NLDT for a classification task and regression task is provided in Figures 2.1 and 2.2 respectively.

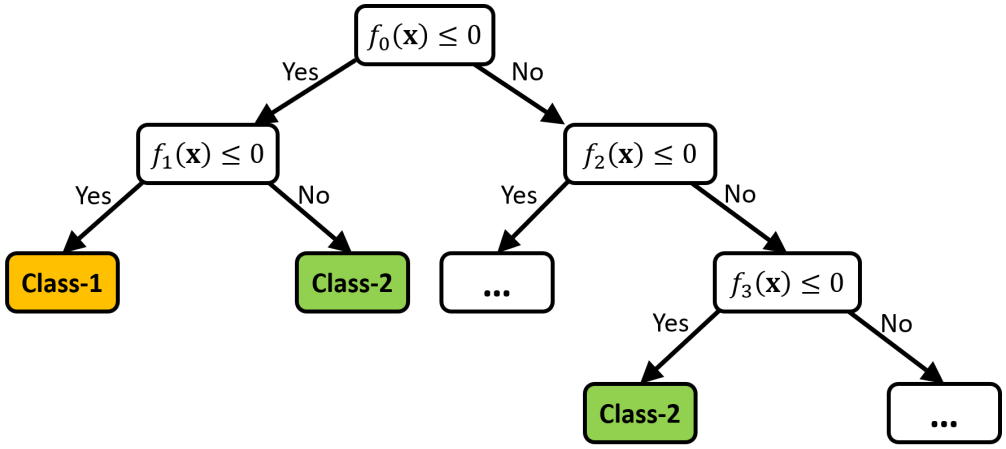


Figure 2.1: NLDT for classification: At each conditional node (white colored), a nonlinear condition $f_i(\mathbf{x}) \leq 0$ is checked for a datapoint $\mathbf{x}$. The datapoint $\mathbf{x}$ traverses the NLDT by following conditions $f_i(\mathbf{x})$ at each conditional node and reaches a leaf node (colored) where it is assigned to a *class* represented by the leaf node.

Here $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$ can be any linear or non-linear functions on the input feature vector $\mathbf{x}$.

Conceptually, all AI models can be represented in the NLDT format. For instance, artificial neural networks (ANNs) or support vector machines (SVMs) designed for binary classification tasks will have one single condition $f_0(\mathbf{x}) \leq 0$ which will be used to predict if the supplied input datapoint $\mathbf{x}$ belongs to *Class 1* or *Class 2*. A natural extension to handle multi class problems is possible either by increasing number of conditions $f_i(\mathbf{x})$ or by increasing number of splits per condition by allowing more branches from a conditional node. We shall visit the topic of handling multiple classes with our proposed NLDT approach later in the thesis. A complicated axis parallel

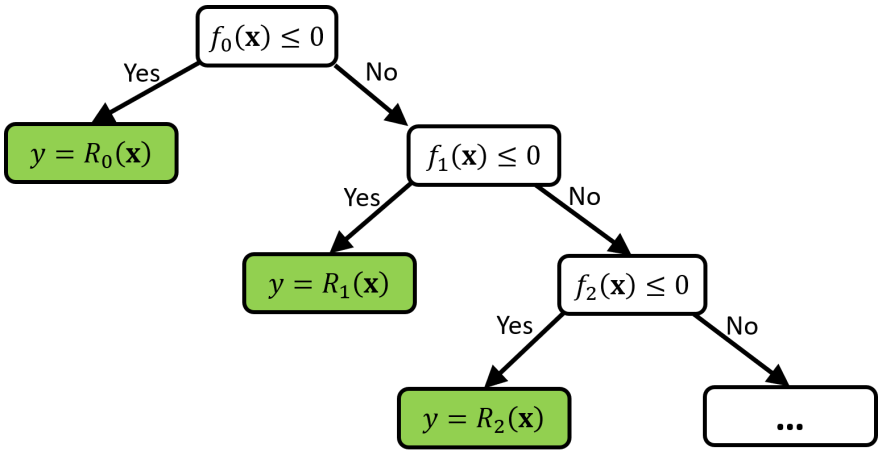


Figure 2.2: NLDT for Regression: Each terminal leaf node (green colored) here represents a regression equation $R_i(\mathbf{x})$ to predict the value of a dependent variable y. However, these regression rules ($y = R_i(\mathbf{x})$) are valid only for certain part of the feature space. These regions of the feature space are defined using the partition rules $f_i(\mathbf{x})$ at each condition node (white).

decision tree (decision tree involving only rules such as $x_i \leq \tau_i$, where x_i is the i-th component of the feature vector $\mathbf{x}$) are by default in the NLDT format.

Similar to classification problems, all the AI models developed to handle regression problems can be represented with the NLDT framework. In case of ANNs, the NLDT representation will involve only one node (which will be also a terminal node) with regression rule $y = R_0(\mathbf{x})$, where $R_0(\mathbf{x})$ will represent an ANN.

While ANNs, SVMs or traditional CART decision trees can be used to model classification decision boundaries or regression surfaces, they are expressed with either a very complicated expression for functions $f_i(\mathbf{x})$ or $R_i(\mathbf{x})$ (like in ANNs or SVMs) but simpler topology (only upto 1 layer of NLDT is sufficient), *OR* are topologically very complex (like CART based decision trees) but have simple expressions for $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$. Hence, existing AI models lie in either of the two extremes:

- Either they have a very complicated function representation for $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$, OR
- They are topologically complicated and involves many nodes in the overall structure of the NLDT.

The above mentioned two aspects make these traditional AI models non-comprehensible and thus non-interpretable.

However, what if we allow a controlled non-linearity for $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$? Maybe allowing some degree of non-linearity for $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$ can help us induce a decision tree with fewer nodes while also ensuring the simplicity of rule expression for $f_i(\mathbf{x})$ and $R_i(\mathbf{x})$ as compared to black-box AI counterparts like ANN, DNN, SVM 0r CART based DT[1].

In this dissertation, we develop algorithms which navigate through the search space of equations to obtain visually simple linear or non-linear rules ($f_i(\mathbf{x})$, $R_i(\mathbf{x})$) as compared to black-box AI counterparts and eventually induce decision tree with fewer nodes. A more in-depth discussion will follow in subsequent chapters where we design dedicated algorithms to solve *classification* and *regression* problems and apply them to generate a relatively interpretable translator to policy networks for reinforcement learning tasks. The interpretable AI (IAI) models developed are relatively simple, easy to *read* and thus humanly more comprehensible thank black-box AI models.

[1] In this work, we will use *black-box* term for AI models represented as DNN, ANN, SVM and topologically complex CART Based trees.

CHAPTER 3

CLASSIFICATION

In a classification task, usually a set of labelled data involving a set of features and its belonging to a specific class are provided. The task in a classification problem solving is to design an algorithm which works on the given dataset and arrives at one or more classification rules (expressed as a function of problem features) which are able to make predictions with maximum classification accuracy. A classifier can be expressed in many different ways suitable for the purpose of the application. It can be a *procedure* in which a feature vector is supplied to the procedure as an input and the procedure determines the class in which the feature vector belongs. It can also be a mathematical function of the feature vector, considering only a few features, instead of all features, that when takes a value within a non-overlapping range in the real space, it belongs to a specific class. It can also assume the form of a rule-set system with several “if-then-else" conditional statements. Each rule in the rule-set system helps to generate an overall classification logic. Decision Trees (DT) fall under this category, wherein the rule-set is represented in an inverted tree based structure. The non-terminal conditional nodes comprise of conditional statements and the terminal leaf-nodes have a class label associated with them. A datapoint traverses through the DT by following the rules at conditional nodes and lands at a particular leaf-node.

The classifier under consideration involves several variables which are supposed to be learned (or optimized) using an optimization algorithm to achieve an optimal classification performance. In case of mathematical single-rule based classifiers like artificial neural networks (ANNs) or support-vector-machines (SVM), these variables are associated with connection-weights or coefficients and location of support-vectors respectively. If the classifier is represented as a rule-set like in DT, each of the conditional split-rules involve some variables which dictate the equation of the split-rule. The optimization problem of determining these variables of classifier involves one or more objectives specifying the quality of classification.

Due to their importance in many practical problems involving design, control, identification,

and other machine learning related tasks, researchers have spent a lot of attention to develop efficient optimization-based classification algorithms [9, 10, 11]. While most algorithms are developed for classifying two-class data sets, the developed methods can be extended to multi-class classification problems as a hierarchical two-class classification problem or by extending them to constitute a simultaneous multi-class classification algorithm. In this chapter, we will consider the binary classification task. An extension to multi-class classification will be discussed later in Chapter 4.

In most classification problem solving tasks, maximizing classification accuracy or minimizing classification error on the labelled dataset is usually used as the sole objective. However, besides the classification accuracy, in many applications, users are also interested in finding an easily interpretable classifier for various practical reasons: (i) it will help identify most important rules which are responsible for the classification task, (ii) it will help provide a more direct relationship of features to have a better insight for the underlying classification task for knowledge enhancement and future developmental purposes. The definition of an easily interpretable classifier will largely depend on the context. In terms of mathematically expressed classifier, this may mean a linear, polynomial, or *posynomial* function involving only a few features. In the case of a DT-based classifier, this may additionally mean a low-depth tree involving only a few branches of the tree.

In our work, we propose a number of novel ideas. First, instead of a DT, we propose to develop a nonlinear decision tree (NLDT) as a classifier, in which every non-terminal conditional node will represent a nonlinear function of features ($f(\mathbf{x})$) to express a split-rule. Each of these split-rules will split the data into two non-overlapping subsets. Successive hierarchical splits of these subsets are carried out and the tree is allowed to grow until one of the termination criteria is met. We argue that flexibility of allowing nonlinear split-rule at conditional nodes (instead of a single-variable based rule, which is found in tradition ID3 based DTs [12]) will result in a more compact DT (i.e. it will have fewer nodes). *Second*, to derive the split-rule at a given conditional node, a dedicated bilevel-optimization algorithm is applied. The upper level optimization focuses at determining the *structure* of the split-rule, while the lower level optimization searches for the necessary coefficients (weights) and biases of the corresponding rule-structure as supplied by the upper level. *Third*,

our proposed methodology uses some generic classification problem information to make the overall bilevel optimization algorithm computationally efficient. *Fourth*, we emphasize simplistic rule structures in our bilevel optimization method so that obtained rules are also relatively more interpretable than the black-box AI counterparts like SVM or ANN.

In the remainder of this chapter, we provide a brief survey of the existing approaches of inducing DTs in Section 3.1. A detailed description about the problem of inducing interpretable DTs from optimization perspective and a high-level view on proposed approach is provided in Section 3.2. Next, we provide an in-depth discussion of the bilevel-optimization algorithm which is adopted to derive interpretable split-rules at each conditional node of NLDT in Section 3.3. Section 3.6 provides a brief overview on the post-processing method which is used to prune the tree to simplify its topology. Compilation of results on standard classification and engineering problems is provided in section 3.7. The chapter ends with concluding remarks and some highlights on future work in Section 3.9.

3.1 Past Studies

There exist many studies involving machine learning and data analytics methods for discovering rules from data. Here, we provide a brief mention of some studies which are close to our proposed study.

Traditional induction of DTs is done in an axis-parallel fashion [12], wherein each split-rule is of type $x_i \leq \tau_i$ or $x_i \geq \tau_i$. Algorithms such as ID3, CART and C4.5 are among the popular ones in this category. The work done in the past to generate *non-axis-parallel* trees can be found in [13, 14, 15], where the researchers rely on randomized algorithms to search for multi-variate hyperplanes. Work done in [16, 17] use evolutionary algorithms to induce oblique DTs. The idea of OCI [15] is extended in [18] to generate nonlinear quadratic split-rules in a DT. Bennett Et al. [19] uses SVM to generate linear/nonlinear split-rules.

However, these works do not address certain key practical considerations, such as the *complexity* of the combined split-rules and handling of biased data. Some works which take the aspect of

complexity of rule into consideration are discussed next.

In [20], ellipsoidal and interval based rules are determined using the set of support-vectors and prototype points/vertex points. The authors there primarily focus at coming up with compact set of if-then-else rules which are comprehensible. Despite its intuitiveness, the approach proposed in [20] doesn't result into comprehensible set of rules on high-dimensional datasets. Another approach suggested in [21] uses the decompositional based technique of deriving rules using the output of a linear-SVM classifier. The rules here are expressed as hypercubes. The method proposed is computationally fast, but it lacks in its scope to address nonlinear decision boundaries and its performance is limited by the performance of a regular linear-SVM. On the other hand, this approach has a tendency to generate more rules if the data is distributed parallel to the decision boundary. The study conducted in [22] uses a trained neural-network as an oracle to develop a DT of *at least m-of-n* type rule-sets (similar to the one described in ID2-of-3 [23]). The strength of this approach lies in its potential to scale up. Its pedagogical approach of inducing DT by referring to the oracle empowers it to create as many synthetic datapoints as desired using the oracle neural-network. However, its accuracy on unseen dataset usually falls by about 3% from the corresponding oracle neural-network counterpart. Authors in [24] use a pedagogical technique to evolve comprehensible rules using genetic-programming. The algorithm G-REX proposed in this work considers the *fidelity* (i.e. how closely can the evolved AI agent mimic the behaviour of the oracle NN) as the primary objective and the compactness of the rule expression is penalized using a parameter to evolve interpretable set of *fuzzy rules* for a classification problem. The approach is good enough to produce comprehensible rule-set, but it needs tweaking and fine tuning of the penalty parameter. A nice summary of the above mentioned methods is provided in [25]. Ishibuchi et al. in [26] implemented a three-objective strategy to evolve fuzzy set of rules using a multi-objective genetic algorithm. The objectives considered in that research were the classification accuracy, number of rules in the evolved rule-set and the total number of antecedent conditions. In [27], an artificial neural network (ANN) is used as a final classifier and a multi-objective optimization approach is employed to find simple and accurate classifier. The simplicity is expressed by the number of nodes.

Hand calculations are then used to analytically express the neural network with a mathematical function. This procedure however becomes intractable when the evolved neural network has a large number of nodes.

Genetic programming (GP) based methods have been found efficient in deriving relevant features from the set of original features which are then used to generate a classifier [28, 29]. In some studies, the entire classifier is encoded with a genetic-representation and the genome is then evolved using GP. Some works conducted in this regard also simultaneously consider complexity of the classifier [30, 31, 32, 33], but those have been limited to axis-parallel or oblique splits. Application of GP to induce nonlinear multivariate decision tree can be found in [34, 35]. Our approach of inducing nonlinear decision tree is conceptually similar to the idea discussed in [34], where the DT is induced in the top-down way and at each internal conditional node, the nonlinear split-rule is derived using a GP. Here, the fitness of a GP solution is expressed as a weighted-sum of misclassification-rates and generalizability term. However, the *interpretability* aspect of the split-rule does not get captured anywhere in the fitness assignment and authors do not report the complexity of the final classifier. No further extension of this study is found in the literature.

In our proposed approach, we attempt to evolve nonlinear DTs which are robust to different biases in the dataset and simultaneously target in evolving nonlinear yet simpler polynomial split-rules at each conditional node of NLDT with a dedicated bilevel genetic algorithm (shown pictorially in Figure 3.1). In oppose to the method discussed in [34], where the fitness of GP individual doesn't capture the notion of complexity of rule expression, the bilevel-optimization proposed in our work deals with the aspects of *interpretability* and *performance* of split-rule in a logically hierarchical manner (conceptually illustrated in Figure 3.2). Results indicate that the proposed bilevel algorithm for evolving nonlinear split-rules eventually generates classifiers which are simpler than other black-box AI and traditional machine learning (ML) based classifiers and have high or comparable classification accuracy on all the test problems used in this study.

3.2 Proposed Approach

3.2.1 Classifier Representation Using Nonlinear Decision Tree

As mentioned before, the classifier developed in this work is represented in the form of a nonlinear decision tree (NLDT) as depicted in Figure 3.1.

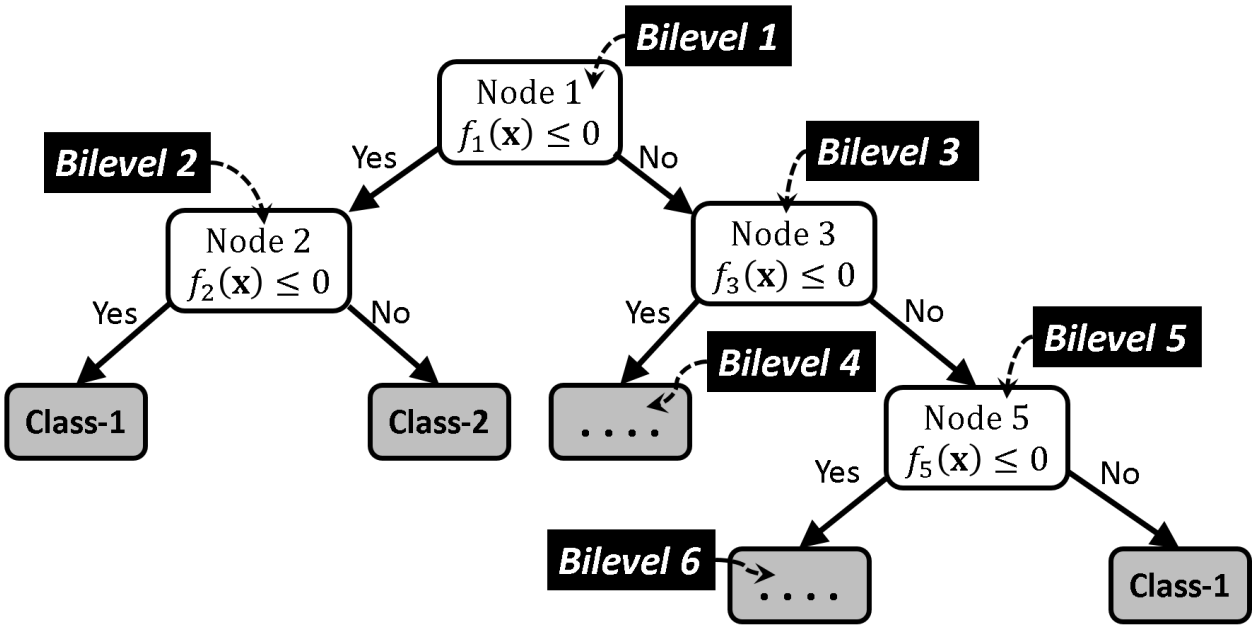


Figure 3.1: An illustration of a Nonlinear Decision Tree (NLDT) for a classification problem. For a given conditional node, the split-rule function $f_i(\mathbf{x})$ is derived using a dedicated bilevel-optimization procedure. At first, the algorithm is applied to the entire data on Node 1 to obtain split-rule $f_1(\mathbf{x}) \leq 0$. If this split-rule is not able to partition the data perfectly, then a similar bilevel optimization is invoked at Node 2 and Node 3 to determine $f_2(\mathbf{x})$ and $f_3(\mathbf{x})$, respectively, on the subset of data present in Node 1 (wherein the Node 2 will have the data satisfying the split-rule $f_1(\mathbf{x}) \leq 0$ and Node 3 will have data of Node 1 which violates split-rule $f_1(\mathbf{x}) \leq 0$). The process continues until a certain termination criteria is met. Terminal leaf-nodes are assigned with a class-label based on the distribution of data in that node.

The decision tree (DT) comprises of conditional (or non-terminal) nodes and terminal leaf-nodes. Each conditional-node of the DT has a rule associated to it. In NLDT, we allow this rule to assume a nonlinear equation. A datapoint $\mathbf{x}$ traverses the DT based on conditions defined by split-rules at conditional nodes and eventually lands at a particular terminal leaf node. To make the DT more interpretable, two aspects are considered:

1. Simplicity of split-rule $f_i(\mathbf{x})$ at each conditional nodes (see Figure 3.2) and

2. Simplicity of the topology of overall DT, which is computed by the counting total number of conditional-nodes in the DT.

Under an ideal scenario, the *simplest split-rule* will involve just one attribute (or feature) and can be expressed as $f(\mathbf{x}) : x_k - \tau \leq 0$. Here, the split occurs based on the k^{th} component of the overall feature vector $\mathbf{x}$. Since for most of the problems, just one such simple split-rule is not sufficient to partition the data into two-classes, many such splits are used in hierarchical fashion to partition the dataset, wherein the first split is done on the entire training-dataset and the subsequent splits are conducted on the subsets of the original training-dataset. This exercise usually resolves into a topologically complicated DT. DTs induced using algorithms such as ID.3 and C4.5 fall under this category. In the present work, we refer to these trees with CART.

On the other extreme, a *topologically-simplest* tree will correspond to the DT involving only one conditional node, but the associated rule is complex to interpret. SVM based classifiers fall in this category, wherein decision-boundary is expressed in form of a complicated mathematical equation. Another way to represent a classifier is through an artificial neural network (ANN), which when attempted to express analytically, will resort into a complicated nonlinear function $f(\mathbf{x})$ without any easy interpretability.

In this work, we propose a novel and compromise approach to above two extreme cases so that the resulting DT is not deep and associated split-rule function at each conditional node $f_i(\mathbf{x})$ assume a nonlinear form with controlled complexity and is easily interpretable as compared to the rule equation corresponding to SVM (or ANN). Allowing the flexibility to have nonlinear split-rules is believed to induce the tree which is topologically simpler than the CART counterpart, while simultaneously ensuring simplicity of split-functions $f_i(\mathbf{x})$. If the split at the root-node is not sufficient to partition the dataset into two classes, subsequent nonlinear splits are determined in a hierarchical fashion, similar to popular ID3 and C4.5 approaches of inducing CART based

decision trees. This process continues until one of the termination criteria is met[1]. The challenging task of obtaining nonlinear split-rule $f(\mathbf{x}) \leq 0$ for each conditional-node is carried out using a dedicated bilevel-optimization, which we discuss in Section 3.2.2. A high-level perspective of this is provided in Figure 3.1. During the training phase, NLDT is induced using a recursive algorithm as shown in Algorithm 1. Brief description about subroutines used in Algorithm 1 is provided below and relevant pseudo codes for *ObtainSplitRule* subroutine and an *evaluator* for upper-level GA are provided in Algorithm 2 and 3 respectively.

- *ObtainSplitRule(Data)*
  - **Input**: $(N \times (d+1))$-matrix representing the dataset comprising of N datapoints with d features. The last column indicates the class-label.
  - **Output**: Nonlinear split-rule $f(\mathbf{x}) \leq 0$.
  - **Method**: Bilevel optimization is used to determine $f(\mathbf{x})$. Details regarding this are discussed in Sec 3.3.
- *SplitNode(Data, SplitRule)*
  - **Input**: Data, split-rule $f(\mathbf{x}) \leq 0$.
  - **Output**: LeftNode and RightNode which are node-data-structures, where LeftNode.Data represents datapoints in the input Data satisfying $f(\mathbf{x}) \leq 0$ while RightNode.Data represents datapoints in the input Data violating the split-rule.

3.2.2 Split-Rule Discovery Using Bilevel Optimization

In this chapter, we restrict the expression of *split-rule* at a conditional node of the decision tree operating on a feature vector $\mathbf{x}$ to assume the following structure:

$$\text{Rule}: \quad f(\mathbf{x}, \mathbf{w}, \boldsymbol{\Theta}, \mathbf{B}) \leq 0, \tag{3.1}$$

[1] Details regarding termination criteria can be found in the Section 3.10

Algorithm 1: Pseudo Code to Recursively Induce NLDT.

```
Input: Dataset
Function UpdatedNode = InduceNLDT(Node, depth):
    Node.depth = depth;
    if TerminationSatisfied(Node) then
        Node.Type = 'leaf';
    else
        Node.Type = 'conditional';
        Node.SplitRule = ObtainSplitRule(Node.Data);
        [LeftNode, RightNode] = SplitNode(Node.Data, Node.SplitRule);
        Node.LeftNode = InduceNLDT(LeftNode, depth + 1);
        Node.RightNode = InduceNLDT(RightNode, depth + 1);
    end
    UpdatedNode = Node;
// NLDT Induction Algorithm
RootNode.Data = Dataset;
Tree = InduceNLDT(RootNode, 0)
```

where $f(\mathbf{x}, \mathbf{w}, \mathbf{\Theta}, \mathbf{B})$ can be expressed in two different forms depending on whether a `modulus` operator m is sought or not:

$$f(\mathbf{x}, \mathbf{w}, \mathbf{\Theta}, \mathbf{B}) = \begin{cases} \theta_1 + w_1 B_1 + \ldots + w_p B_p, & \text{if } m = 0, \\ \left|\theta_1 + w_1 B_1 + \ldots + w_p B_p\right| - \left|\theta_2\right|, & \text{if } m = 1. \end{cases} \tag{3.2}$$

Here, w_i's are coefficients or weights of several power-laws (B_i's), θ_i's are biases, m is the *modulus-flag* which indicates the presence or absence of the *modulus* operator, p is a user-specified parameter to indicate the maximum number of *power-laws* (B_i) which can exist in the expression of $f(\mathbf{x})$, and B_i represents a power-law rule of type:

$$B_i = x_1^{b_{i1}} \times x_2^{b_{i2}} \times \ldots \times x_d^{b_{id}}, \tag{3.3}$$

Algorithm 2: *ObtainSplitRule* subroutine. Implements a *Bilevel* optimization algorithm of determine a split-rule. The UpperLevel searches the space of Block Matrix **B** and modulus-flag m. Constraint violation value for an upper level individuals comes by executing lower-level GA (LLGA) as shown in Algorithm 3.

Input: Data
Output : SplitRule `// split-rule` $f(\mathbf{x})$
Function *SplitRule = ObtainSplitRule(Data)*:
  Initialize: P_U `// Upper Level population`
  `// Execute LLGA (Algorithm 3)`
  P_U = *EvaluateUpperLevelPop*(P_U);
  `// Upper Level GA Loop`
  for *gen = 1:MaxGen* **do**
    P_U^{Parent} = *Selection*(P_U);
    P_U^{Child} = *Crossover*(P_U^{Parent});
    P_U = *Mutation*(P_U^{Child});
    `// Execute LLGA (Algorithm 3)`
    P_U = *EvaluateUpperLevelPop*(P_U);
    `// Elite preservation`
    P_U = *SelectElite*(P_U^{Parent}, P_U);
    if *TerminationConditionSatisfied* **then**
      break;
    end
  end
  `// Extract best solution in` P_U
  SplitRule = *ObtainBestInd*(P_U);

B is a block-matrix of exponents b_{ij}, given as follows:

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1d} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & b_{p3} & \dots & b_{pd} \end{bmatrix}. \tag{3.4}$$

Exponents b_{ij}'s are allowed to assume values from a specified discrete set **E**. In this work, we set $p = 3$ and $\mathbf{E} = \{-3, -2, -1, 0, 1, 2, 3\}$ to limit the maximum complexity of the rule, however value of p and set **E** can be changed by the user. Parameters w_i and θ_i are real-valued variables in $[-1, 1]$. The feature vector **x** is a datapoint in a d-dimensional space. Another user-defined parameter $a_{\max}$

Algorithm 3: *EvaluateUpperLevelPop* subroutine. A dedicated *LowerLevel* optimization is executed for each upper level population member.

Input: P_U // Upper Level population
Output P'_U // Evaluated Population
:
Function $P'_U = EvaluateUpperLevelPop(P_U)$**:**
  for $i = 1{:}PopSize$ **do**
    // Execute LLGA (see Section 3.3.3)
    $[F_L, \mathbf{w}, \mathbf{\Theta}] = LLGA(\text{Data}, \mathbf{B}, m)$;
    $[P_U[\text{i}].\mathbf{w}, P_U[\text{i}].\mathbf{\Theta}] = [\mathbf{w}, \mathbf{\Theta}]$;
    // Constraint and Fitness Value
    $P_U[\text{i}].\text{CV} = F_L - \tau_I$;
    $P_U[\text{i}].F_U = MaxActiveTerms(\mathbf{B})$;
  end
  $P'_U = P_U$

controls the maximum number of variables that can be present in each power-law B_i. The default is $a_{\max} = d$ (i.e. dimension of the feature space).

3.3 Bilevel Approach for Split-Rule Identification

3.3.1 The Hierarchical Objectives to derive split-rule

Here, we illustrate the need of formulating the problem of split-rule identification as a bilevel-problem using Figure 3.2.

The geometry and shape of split-rules is defined by exponent terms b_{ij} appearing in its expression (Eq. 3.2 and 3.3) while the orientation and location of the split-rule in the feature-space is dictated by the values of coefficients w_i and biases θ_i (Eq. 3.2). Thus, the above optimization task of estimating split-rule $f(\mathbf{x})$ involves two types of variables:

1. Discrete: B-matrix representing exponents of B-terms (i.e. b_{ij} as shown in Eq. 3.4) and the modulus flag m indicating the presence or absence of a modulus operator in the expression of $f(\mathbf{x})$, and

2. Continuous: weights $\mathbf{w}$ and biases $\mathbf{\Theta}$ in each rule function $f(\mathbf{x})$.

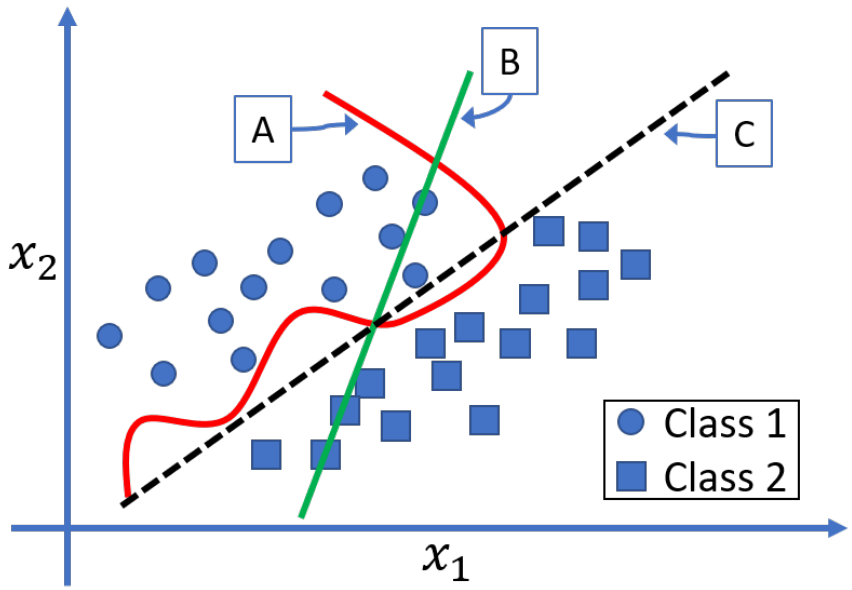


Figure 3.2: In this illustration, a two-class data comprising of two features x_1 and x_2 is provided. A, B and C are three different split-rules. Split-rule A is able to optimally partition the data, but is complex and may not be interpretable. Rule B is simple, however it does not produce accurate classification. Rule C is simple as well as classifies the data accurately.

Identification of a good structure for B terms and value of m is a more difficult task, compared to the weight and bias identification. We argue that if both types of variables are concatenated in a single genome, a good $(\mathbf{B}, m)$ combination may be associated with a not-so-good $(\mathbf{w}, \mathbf{\Theta})$ combination (like split-rule B in Fig. 3.2), thereby making the whole solution vulnerable to deletion during evolution. It may be better to separate the search of a good structure of $(\mathbf{B}, m)$ combination from the weight-bias search at the same level, and search for the best weight-bias combination for every $(\mathbf{B}, m)$ pair as a detailed task. This hierarchical structure of the variables motivates us to employ a *bilevel* optimization approach [36] to handle above variables. The upper level optimization algorithm searches the space of $\mathbf{B}$ and m. Then, for each $(\mathbf{B}, m)$ pair, the lower level optimization algorithm is invoked to determine the optimal values of $\mathbf{w}$ and $\mathbf{\Theta}$. Referring to Fig. 3.2, the upper-level will search for the structure of $f(\mathbf{x})$ which might have a nonlinearity (like in rule A) or might be linear (like rule B). Then, for each upper-level solution (for instance a solution corresponding to a linear rule structure), the lower-level will adjust its weights and biases to determine its optimal location C.

The bilevel optimization problem can be then formulated as shown below:

$$
\begin{aligned}
\text{Min.} \quad & F_U(\mathbf{B}, m, \mathbf{w}^*, \boldsymbol{\Theta}^*), \\
\text{s.t.} \quad & (\mathbf{w}^*, \boldsymbol{\Theta}^*) \in \operatorname{argmin} \left\{ F_L(\mathbf{w}, \boldsymbol{\Theta})|_{(\mathbf{B},m)} \right| \\
& \quad F_L(\mathbf{w}, \boldsymbol{\Theta})|_{(\mathbf{B},m)} \leq \tau_I, \\
& \quad -1 \leq w_i \leq 1, \ \forall i, \ \boldsymbol{\Theta} \in [-1, 1]^{m+1} \Big\}, \\
& m \in \{0, 1\}, \ b_{ij} \in \{-3, -2, -1, 0, 1, 2, 3\},
\end{aligned}
\tag{3.5}
$$

where the upper level objective F_U is quantifies the simplicity of the split-rule, and, the lower-level objective F_L quantifies quality of split resulting due to split-rule $f(\mathbf{x}) \leq 0$. An upper-level solution is considered feasible only if it is able to partition the data within some acceptable limit which is set by a parameter τ_I. A more detailed explanation regarding the upper level objective F_U and the lower-level objective F_L is provided in next sections. A pseudo code of the bilevel algorithm to obtain split-rule is provided in Alogrithm 2 and the pseudo code provided in Algorithm 3 gives on overview on the evaluation of population-pool in upper level GA. In next sections, we provide a mathematical insight on the procedure to compute lower-level and upper level objective functions, F_L and F_U respectively.

3.3.1.1 Computation of F_L

The impurity of a node in a decision tree is defined by the distribution of data present in the node. In this work, we use *gini-score* to gauge the impurity of a node. Thus, for a given parent node P in a decision tree and two child nodes L and R resulting from it, the *net-impurity of child nodes* (F_L) can be computed as follows:

$$
F_L(\mathbf{w}, \boldsymbol{\Theta})|_{(\mathbf{B},m)} = \left(\frac{N_L}{N_P} \text{Gini}(L) + \frac{N_R}{N_P} \text{Gini}(R) \right)_{(\mathbf{w},\boldsymbol{\Theta},\mathbf{B},m)}, \tag{3.6}
$$

where N_P is the total number of datapoints present in the parent node P, and N_L and N_R indicate the total number of points present in left (L) and right (R) child nodes, respectively. Datapoints in P which satisfy the split-rule at node P (i.e. $f_P(\mathbf{x}) \leq 0$, $\mathbf{x} \in P$) (Eq 3.1) go to left node, while the rest go to the right node. The objective F_L of minimizing the net-impurity of child nodes favors

the creation of purer child nodes (i.e. nodes with a low gini-score value). For an ideal split, the left and right child nodes will have completely homogeneous data, i.e. all datapoints present in the node will belong to one class (say left node will have all points from Class-1 while right node will have all points from Class-2).

3.3.1.2 Computation of F_U

The objective F_U is *subjective* in its form since it targets at dealing with a subjective notion of *generating visually simple* equations of split-rule. Usually, equations with more variables and terms seem visually complicated. Taking this aspect into consideration, F_U is set as *the total number of non-zero exponent terms* present in the overall equation of the split-rule (Eq. 3.1). Mathematically, this can be represented with the following equation:

$$F_U(\mathbf{B}, m, \mathbf{w}^*, \mathbf{\Theta}^*) = \sum_{i=1}^{p} \sum_{j=1}^{d} g(b_{ij}), \tag{3.7}$$

$$\text{where} \quad g(\alpha) = \begin{cases} 1, & \alpha \neq 0, \\ 0, & \alpha = 0. \end{cases}$$

Here, we use only $\mathbf{B}$ to define F_U, but another more comprehensive simplicity term involving presence or absence of modulus operators and relative optimal weight/bias values of the rule can also be used.

3.3.2 Upper Level Optimization (ULGA)

A genetic algorithm (GA) is implemented to explore the search space of power-laws B_i's and modulus flag m in the upper level. The genome is represented as a tuple $(\mathbf{B}, m)$ wherein $\mathbf{B}$ is a matrix as shown in Equation 3.4 and m assumes a Boolean value of 0 or 1.

The upper level GA focuses at estimating a simple equation of the split-rule within a desired value of *net impurity* (F_L) of child nodes. Thus, the optimization problem for upper level is

formulated as a single objective constrained optimization problem as shown in Eq. 3.5. The constraint function $F_L(\mathbf{w}, \mathbf{\Theta})|_{(\mathbf{B},m)}$ is evaluated at the lower level of our bilevel optimization framework. The threshold value τ_I indicates the desired value of net-impurity of resultant child nodes (Eq. 3.6). In our experiments, we set τ_I to be 0.05. As mentioned before, a solution satisfying this constraint implies creation of purer child nodes. Minimization of objective function F_U should result in a simplistic structure of the rule and the optimization will also reveal *key* variables needed in the overall expression.

3.3.2.1 Custom Initialization for Upper Level GA

Minimization of objective F_U (given in Eq 3.7) requires to have less number of active (or non-zero) exponents in the expression of split-rule (Eq 3.1). To facilitate this, the population is initialized with a restriction of having only one active (i.e. non-zero) exponent in the expression of split-rule, i.e., any-one of the b_{ij}'s in the block matrix $\mathbf{B}$ is set to a non-zero value from a user-specified set $\mathbf{E}$ and the rest of the elements of matrix $\mathbf{B}$ are set to zero. Note here that only $2d$ number of unique individuals (d individuals with $m = 0$ and d individuals with $m = 1$) can exist which satisfy the above mentioned restriction. If the population size for upper level GA exceeds $2d$, then remaining individuals are initialized with two non-zero active-terms (i.e. randomly, two b_{ij}'s are set to a non-zero value, while rest of the elements in $\mathbf{B}$ are fixed to zero) and the process continues until all population members in the upper level are initialized uniquely. As the upper level GA progresses, incremental enhancement in rule complexity is realized through crossover and mutation operations, which are described next.

3.3.2.2 Ranking of Upper Level Solutions

The binary-tournament selection operation and (μ+λ) survival selection strategies are implemented for the upper level GA. Selection operators use the following hierarchical ranking criteria to perform selection:

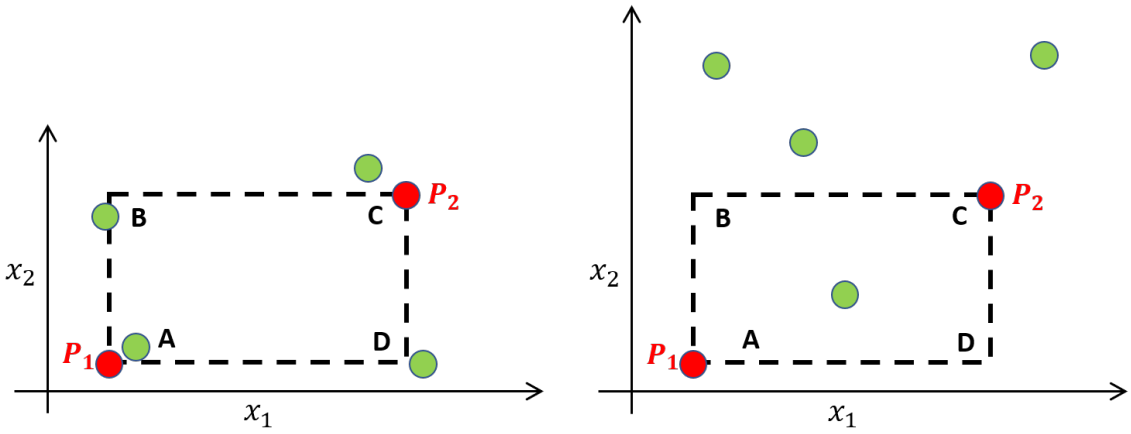
Definition 3.3.1 *For two individuals i and j in the upper level, rank(i) is better than rank(j), when any of the following is true:*

- *i and j are both infeasible AND $F_L(i) < F_L(j)$,*
- *i is feasible (i.e. $F_L(i) \leq \tau_I$) AND j is infeasible (i.e. $F_L(j) > \tau_I$),*
- *i and j both are feasible AND $F_U(i) < F_U(j)$,*
- *i and j both are feasible AND $F_U(i) = F_U(j)$*
  AND $F_L(i) < F_L(j)$,
- *i and j both are feasible AND $F_U(i) = F_U(j)$*
  AND $F_L(i) = F_L(j)$ AND $m(i) < m(j)$.

3.3.2.3 Custom Crossover Operator for Upper Level GA

The main challenging aspect of developing an evolutionary algorithm is to design meaningful genetic operators. Crossover operation is one such stage in GAs where there is an *information exchange* between two (or more) individual species in the parent population pool. The crossover operation between two (or more) participating parents creates one or more children. For a real coded GA involving only real continous variables, Simulated Binary Crossover (SBX) [37] is one such meaningful genetic operator for exchanging information between two parents to create two children. A 2D illustration of the concept of meaningful crossover is provided in Figure 3.3.

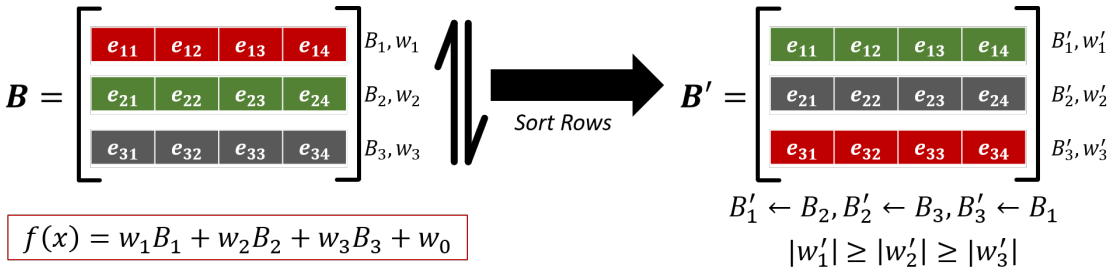
In our case, in the upper level, the crossover operator needs to be designed to meaningfully crossover two *parent equations*. To do that, population members are first clustered according to their m-value – all individuals with $m = 0$ belong to one cluster and all individuals with $m = 1$ belong to another cluster. The crossover operation is then restricted to individuals belonging to the same cluster. The crossover operation selects two parents P_1 and P_2 from the same cluster having block matrix $\mathbf{B_{P_1}}$ and $\mathbf{B_{P_2}}$ to create two *children* with block matrices $\mathbf{B_{C_1}}$ and $\mathbf{B_{C_2}}$. As mentioned before, each row of a block matrix $\mathbf{B}$ represents a power-law B_i (Eq. 3.3). The crossover



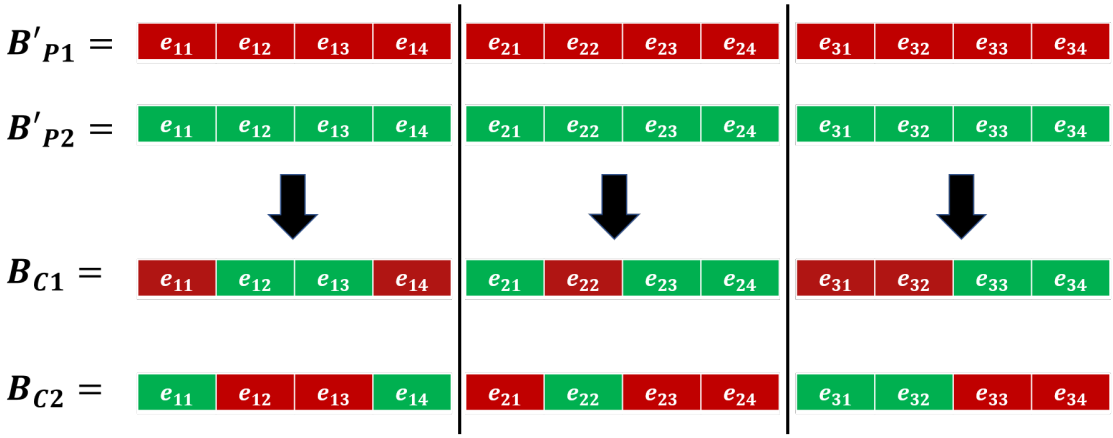
(a) Meaningful Crossover (SBX) example (b) Meaningless Crossover example

Figure 3.3: Illustrations of meaningful and meaningless crossover operations. In the meaningful crossover operation, children (green dots) will carry information of parents (red dots). Hence, they will be located with high probability at the corners of the rectangle $ABCD$. On the other hand, the meaningless crossover operation creates children at random locations, without taking leverage of parents.

operation is executed separately on each row of block matrices of participating parents to generate corresponding rows of child block matrices. First, rows of block matrices of participating parents are rearranged in descending order of the magnitude of their corresponding coefficient values (i.e. weights w_i of Eq. 3.2). This way, the most influential power-law in the equation of parent individual will be shuffled to the first row of the rearranged block matrix $\mathbf{B'}$. The second row of the rearranged block matrix $\mathbf{B'}$ will have exponents corresponding to the second most influential power-law and so on. Let the parent block matrices be represented as $\mathbf{B'_{P_1}}$ and $\mathbf{B'_{P_2}}$ after this rearrangement. The crossover operation is then conducted element wise on each row of $\mathbf{B'_{P_1}}$ and $\mathbf{B'_{P_2}}$. Doing so allows us to mate those rows of the participating parents which had similar importance in the respective equations of the split-rule. For better understanding of the cross-over operation, a psuedo code is provided in Algorithm 4 and a schematic is provided in Figure 3.4.



(a) Sorting of B-mat



(b) Crossover Operation

Figure 3.4: Upper Level Crossover Operation

3.3.2.4 Custom Mutation Operator for Upper Level GA

In genetic algorithms, the purpose of mutation operation is to conduct *local* perturbations to parent solutions. A *meaningful and controlled* mutation generates the mutated individual in the vicinity of the unmutated original value[2]. In real coded GAs, polynomial mutation [38] is popularly used to execute this task. An illustrative plot of polynomial mutation operator is provided in Figure 3.5a.

Since the upper level of our GA focuses on the search space involving discrete variables, we use the discretized version of mutation to mutate the values of exponents b_{ij} (Eq. 3.3) as shown in Figure 3.5. For a given upper level solution with block matrix **B** and modulus flag m, the probability with which b_{ij}'s and m gets mutated is controlled by parameter p^U_{mut}. From the experiments, value of $p^U_{mut} = 1/d$ is found to work well. The mutation operation then changes the value of exponents

[2]while most of the times mutation is desired to create *local* perturbations, under some scenario, a *randomized* mutation is favoured to increase the diversity of the population pool.

Algorithm 4: Crossover operation in upper level GA.

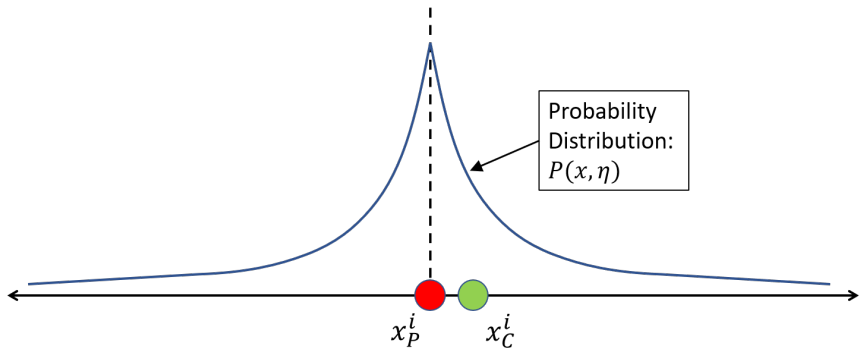
input : Block matrices $\mathbf{B_{P_1}}$ and $\mathbf{B_{P_2}}$, and weight vectors $\mathbf{w_{P_1}}$ and $\mathbf{w_{P_2}}$.
output Child block matrices $\mathbf{B_{C_1}}$ and $\mathbf{B_{C_2}}$.
:

```
B'_P1 = SortRows(B_P1, w_P1);
B'_P2 = SortRows(B_P2, w_P2);
n_rows = Size(B_P1, 1);
n_cols = Size(B_P1, 2);
for i ← 1 to n_rows do
    for j ← 1 to n_cols do
        r = rand();
        // random no.  between 0 and 1.
        if r ≤ 0.5 then
            B_C1(i, j) = B_P1(i, j);
            B_C2(i, j) = B_P2(i, j);
        else
            B_C1(i, j) = B_P2(i, j);
            B_C2(i, j) = B_P1(i, j);
        end
    end
end
```

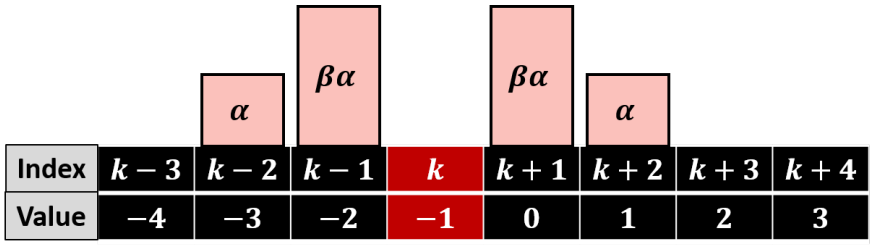
b_{ij} and the modulus flag m. Let the domain of b_{ij} be given by $\mathbf{E}$, which is a sorted finite set of allowable exponents. Since $\mathbf{E}$ is sorted, its elements can be accessed using an integer-id k, with *id-value* of $k = 1$ representing the smallest exponent and $k = n_e$ representing the largest exponent. In our case, $\mathbf{E} = [-3, -2, -1, 0, 1, 2, 3]$ (making $n_e = 7$). The mechanism with which the mutation operator mutates the value of b_{ij} for any arbitrary sorted array $\mathbf{E}$ is illustrated in Fig. 3.5.

Here, the *red* tile indicates the *index* (k) of exponent b_{ij} in array $\mathbf{E}$. *Red* shaded vertical bars indicate the probability distribution for obtaining mutated-values. The b_{ij} can be mutated to either of $k-2, k-1, k+1$, or $k+2$ id-values with a probability of α, $\beta\alpha$, $\beta\alpha$, and α respectively. The parameter β is preferred to be greater than one and is supplied by the user. The parameter α is then computed using the following equation:

$$\alpha = \frac{1}{2(1+\beta)}. \tag{3.8}$$



(a) Polynomial Mutation on the i-th component of an individual $\mathbf{x}$. The probability distribution curve $P(x, \eta)$ dictates the location where the i-th component of child (i.e. x_C^i) will be after mutating the parent component (i.e. x_P^i). Here, the mutated value x_C^i has a higher probability of getting created near the parent value x_P^i. The steepness of the probability curve $P(x, \eta)$ can be adjusted by parameter η, with higher values of η giving steeper curve.



(b) Discretized Mutation mimics the behaviour of real polynomial mutation discussed in figure above. Red vertical bars indicate the probability distribution of obtaining mutated values. The parent *unmutated* value is located at k-th index value. Here, we intentionally avoid creating mutated value on the parent value and encourage creation of mutated values at immediate two neighboring allowable values to facilitate the creation of unique solutions.

Figure 3.5: Mutation for Upper Level GA

The above formula to compute α is derived by equating the sum of probabilities to 1. In our experiments, we have set $\beta = 3$. The value of the modulus flag m is mutated randomly to either 0 or 1 with 50% probability.

In order to bias the search to create simpler rules (i.e. split-rules with a small number of non-zero b_{ij}'s), we introduce a parameter p_{zero}. The value of parameter p_{zero} indicates the probability with which a variable b_{ij} participating in mutation is set to *zero* (i.e. $b_{ij} \leftarrow 0$). In our case, we use $p_{zero} = 0.75$. Thus, $b_{ij} \rightarrow 0$ with a net probability of $p_{mut}^U \times p_{zero}$.

3.3.2.5 Duplicate Update Operator

After creating the offspring population, we attempt to eliminate duplicate population members. For each duplicate found in the child population, the block-matrix **B** of that individual is randomly mutated using the following equation

$$\mathbf{B}(i_r, j_r) = \mathbf{E}(k_r), \tag{3.9}$$

where i_r, j_r and k_r are randomly chosen integers within specified limits. This process is repeated for all members of child population, until each member is unique within the child population. This operation allows to maintain diversity and encourages the search to generate multiple novel and optimized equations of the split-rule.

$(\mu+\lambda)$ survival selection operation is then applied on the combined child and parent population. The selected elites then go to the next generation as parents and the process repeats.

The parameter setting for upper level GA can be found in Section 3.10.

3.3.3 Lower Level Optimization (LLGA)

For a given population member of the upper level GA (with block matrix **B** and modulus flag m as variables), the lower level optimization problem determines the coefficients w_i (Eq. 3.2) and biases θ_i such that $F_L(\mathbf{w}, \mathbf{\Theta})|_{(\mathbf{B},m)}$ (Eq. 3.6) is minimized. Thus, the lower level optimization problem can be stated as below:

$$\begin{aligned} \text{Minimize:} \quad & F_L(\mathbf{w}, \mathbf{\Theta})|_{(\mathbf{B},m)}, \\ & \mathbf{w} \in [-1, 1]^p, \quad \mathbf{\Theta} \in [-1, 1]^{m+1}. \end{aligned} \tag{3.10}$$

We describe the details of the real-parameter GA used for solving the lower level problem next.

3.3.3.1 *Custom Initialization for Lower Level GA*

One of the most crucial and effective operator to determine optimal values of variables of lower level optimization was the *initialization* operation. As stated before in Section 3.3.1.1, the lower level objective function F_L provides the quantification to the net impurity of child nodes (Eq. 3.6). The number of points going to left child node C_L and right child node C_R depends on the sign of the split function $f(\mathbf{x})$ of the parent node, where $f(\mathbf{x}) \leq 0 \implies \mathbf{x} \rightarrow C_L$ and $f(\mathbf{x}) > 0 \implies \mathbf{x} \rightarrow C_R$. Geometrically the split function $f(\mathbf{x}, \mathbf{B}, m, \mathbf{w}, \mathbf{\Theta})$ is linear in the transformed *B-space* since

$$f(\mathbf{x}, \mathbf{B}, m, \mathbf{w}, \mathbf{\Theta}) = \begin{cases} \sum_{i=1}^{p} w_i B_i + \theta_0 & \text{if } m = 0 \\ \left| \sum_{i=1}^{p} w_i B_i + \theta_0 \right| - |\theta_1| & \text{if } m = 1 \end{cases}$$

where $B_i = \prod_{j=1}^{d} x_j^{b_{ij}}$[3]. Hence, the lower level searches the optimal orientation (dictated by $\mathbf{w}$) and location (dictated by $\mathbf{\Theta}$) of the linear straight line in the mapped *B-space* as shown in Figure 3.6.

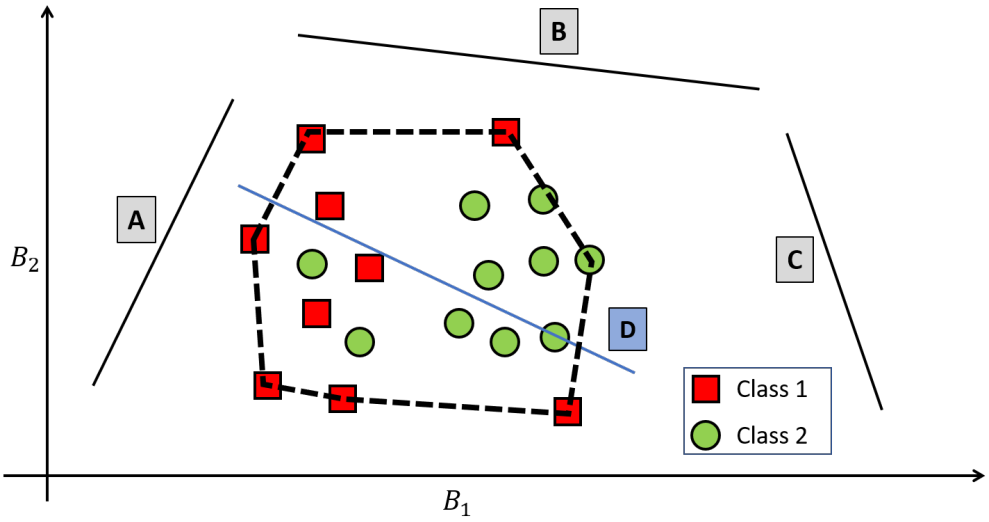


Figure 3.6: Illustration of Data in B-Space. The split function $f(\mathbf{x}, \mathbf{B}, m, \mathbf{w}, \mathbf{\Theta})$ is a straight line in *B-space*. Dotted lines represent the convex hull engulfing the data. Different possible split functions are shown by straight lines A, B, C and D.

Since no separation of data is evident if the straight line representing the split function in

[3] technically, with $m = 1$, there will be two linear lines which will represent the split fuction. However, each line will be separating two classes.

B-space lies outside of the convex hull (dotted lines in Figure 3.6), the function landscape of $F_L(\mathbf{w}, \mathbf{\Theta})|_{\mathbf{B},m}$ is *flat* in the region outside the convex hull (since either $N_L = 0$ or $N_R = 0$ in Eq. 3.6). This constitutes a very significant portion of the domain of F_L. If the initial population pool of straight lines lies in the region outside of the convex hull, then the optimization algorithm gets *no signal* and motivation to move and reorient pool of initialized straight lines. Under certain scenarios, crossover and mutation operators might generate solutions representing straight lines passing through the dataset (shown by D in Figure 3.6), thereby partitioning it into two subsets $f(\mathbf{x}) \leq 0$ and $f(\mathbf{x}) > 0$. In practice, a lot of computation is wasted while arriving at a situation where most of the individuals in the population pool represent straight lines in *B-space* which crosses the convex hull and partition the data.

In a bilevel optimization, the lower level problem must be solved for every upper level variable vector, thereby requiring a computationally fast algorithm for the lower level problem. Considering these aspects, instead of creating every population member in lower level randomly, we use the mixed dipole concept [39, 17, 40] to initialize the population pool of straight lines in *B-space* to ensure each of the initialized member partitions atleast one point in the dataset from the rest as shown in Figure 3.7. This smart initialization facilitates faster convergence towards optimal (or near-optimal) values of $\mathbf{w}$ and $\mathbf{\Theta}$.

The dipole based initialization is done in the following way:

Step 1: Randomly pick two datapoints $\mathbf{x_A}$ and $\mathbf{x_B}$ such that $\mathbf{x_A} \in$ Class-1 and $\mathbf{x_B} \in$ Class-2.

Step 2: Weights $\mathbf{w_h}$ and bias θ_h of the hyperplane H (where $H(\mathbf{w_h}, \theta_h) : \mathbf{w_h} \cdot \mathbf{x} + \theta_h = 0$) separating $\mathbf{x_A}$ and $\mathbf{x_B}$ can be computed with equation mentioned below

$$\begin{aligned} \mathbf{w_h} &= \mathbf{x_A} - \mathbf{x_B}, \\ \theta_h &= \Delta \times (-\mathbf{w_h} \cdot \mathbf{x_B}) - (1 - \Delta) \times (\mathbf{w_h} \cdot \mathbf{x_A}), \end{aligned} \tag{3.11}$$

where Δ is a random number between 0 and 1. This is pictorially represented in Fig. 3.7.

Step 3: Set values of variables $\mathbf{w}$ and $\boldsymbol{\Theta}$ of a population member of lower level GA as shown below

$$\mathbf{w} = \mathbf{w_h}, \tag{3.12}$$

$$\theta_1 = \theta_h, \tag{3.13}$$

$$\theta_2 = \min(\Delta, 1-\Delta) \quad \text{if } m = 1. \tag{3.14}$$

Step 4: Repeat Steps 1-3 until all population members of lower level GA are initialized.

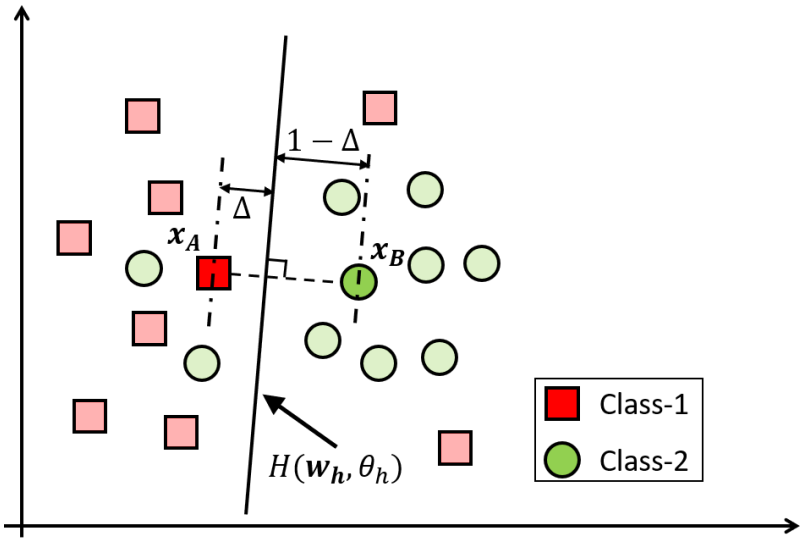


Figure 3.7: Mixed dipole ($\mathbf{x_A}$, $\mathbf{x_B}$) and a hyperplane $H(\mathbf{w_h}, \theta_h)$.

As mentioned before, this method of initialization is found to be more efficient than randomly initializing $\mathbf{w}$ and Θ since in the region beyond the convex-hull bounding the training dataset (which constitutes major volume of the domain space), the landscape of F_L is flat thereby making any optimization algorithm to stagnate.

3.3.3.2 Selection, Crossover, and Mutation for Lower Level GA

The binary tournament selection [38], SBX crossover [37] and polynomial mutation [41] were used to create the offspring solutions. $(\mu + \lambda)$ survival selection strategy was then adopted to preserve elites.

3.3.3.3 Termination Criteria for Lower level GA

The lower level GA is terminated after a maximum of 50 generations were reached or when the change in the lower level objective value (i.e. F_L) is less than 0.01% in the past 10 consecutive generations.

Other parameters setting for lower-level GA can be found in Section 3.10.

3.4 Ablation Studies and Comparison

In this section, we test the efficacy of lower-level and upper-level optimization algorithm by applying them on four customized problems (DS1-DS4) which are shown pictorially in Fig. 3.8. Their performances are compared against two standard classifiers: CART and support-vector-machines (SVM)[4].

3.4.1 Ablation Studies on Lower Level GA

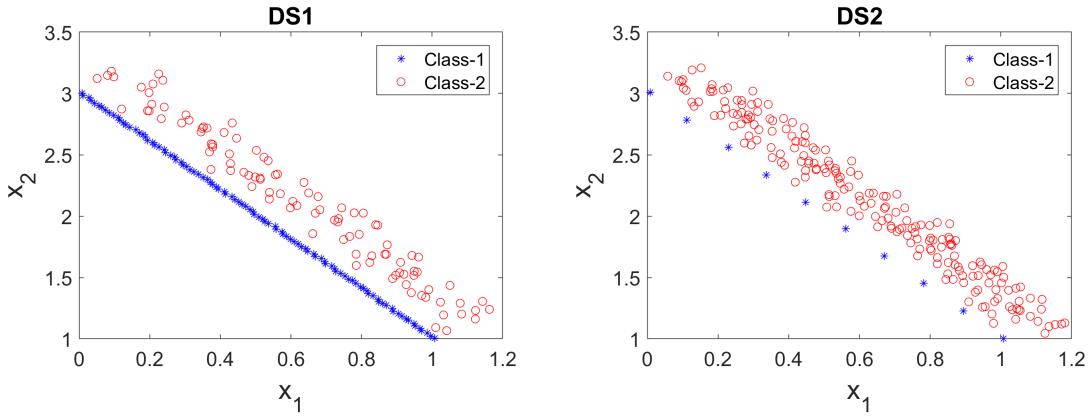
Since the lower level GA (LLGA) focuses at determining coefficients w_i and bias θ_i of linear split-rule in *B-space*, DS1 and DS2 datasets are used to guage its efficacy. The block matrix **B** is fixed to 2×2 identity matrix and the modulus-flag m is set to 0. Hence, the split-rule (classifier) to be evolved is

$$f(\mathbf{x}) = \theta_1 + w_1 x_1 + w_2 x_2,$$

where w_1, w_2 and θ_1 are to be determined by the LLGA.

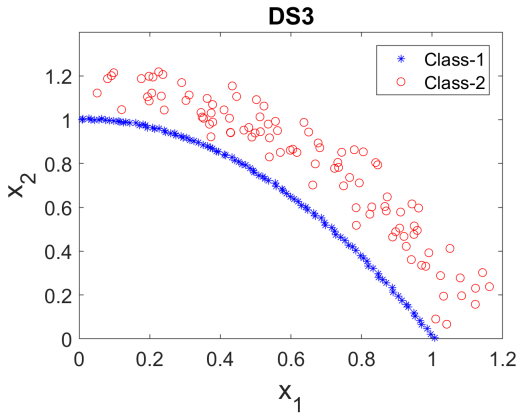
The classifier generated by our LLGA is compared with that obtained using the standard SVM algorithm (without any kernel-trick) on DS1 dataset. Since DS2 dataset is unbalanced, SMOTE algorithm [42] is first applied to over-sample datapoints of the minority class before generating the classifier using SVM. For the sake of completeness, classifier generated by SVM on DS2 dataset, without any oversampling, is also compared against the ones obtained using LLGA and

[4]Here, the support vector machine is applied without any kernal-trick.

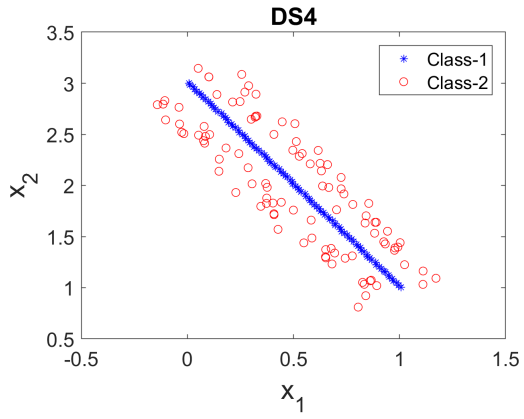


(a) DS1 Dataset.

(b) DS2 Dataset.



(c) DS3 Dataset.



(d) DS4 Dataset.

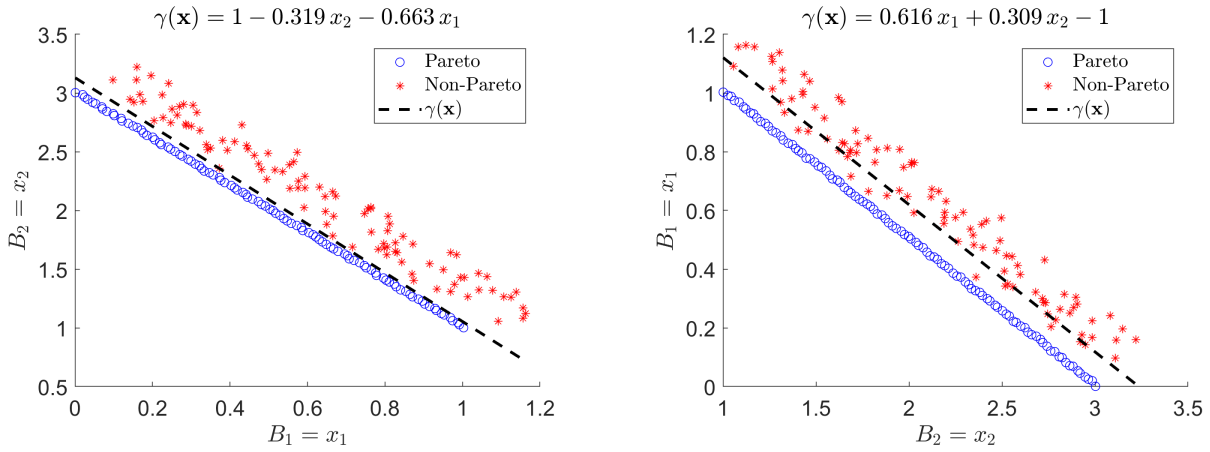
Figure 3.8: Customized datasets. DS1 is linear and balanced, DS2 is linear but unbalanced with minority class having 10x less points. DS3 is nonlinear and DS4 has a *sandwiched* distribution.

SMOTE+SVM. The results are shown in Fig. 3.9 and Fig. 3.10, respectively, for DS1 and DS2 datasets. Type-1 and Type-2 errors are reported for each experiment[5].

It is clearly evident from the results shown in Fig. 3.9 that the proposed customized LLGA is more efficient than SVM in arriving at the desired split-rule as a classifier. The LLGA is able to find the decision boundary with 100% prediction accuracy on training and testing datasets. However, the classifier generated by SVM has an overlap with the cloud of datapoints belonging to the scattered class and there-by resulting into the accuracy of 89% on the testing dataset.

On the DS2 dataset, SVM is required to rely on SMOTE to synthetically generate datapoints

[5]Type-1 error indicates the percentage of datapoints belonging to *Class-1* getting classified as *Class-2*. Type-2 error indicates the percentage of points belonging to *Class-2* getting classified as *Class-1*).



(a) LLGA: (0%, 0%); (0%, 1%)

(b) SVM: (0%, 17%); (0%, 10%)

Figure 3.9: Results on DS1 dataset to benchmark LLGA. Numbers in first parenthesis indicate *Type-1* and *Type-2* error on training data and the numbers in second parenthesis indicate *Type-1* and *Type-2* error on testing data.

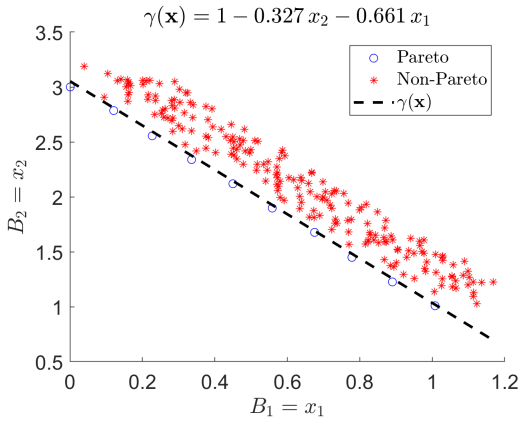
in order to achieve respectable performance as can be seen from Fig. 3.10b (without SMOTE) and 3.10c (with SMOTE). However, LLGA performs better without SMOTE, as shown in Fig. 3.10a.

Ablation study conducted above on LLGA clearly indicates that the customized optimization algorithm developed for estimating weights $\mathbf{w}$ and bias $\boldsymbol{\Theta}$ in the expression of a split-rule is reliable and efficient and could easily outperform the classical SVM algorithm.

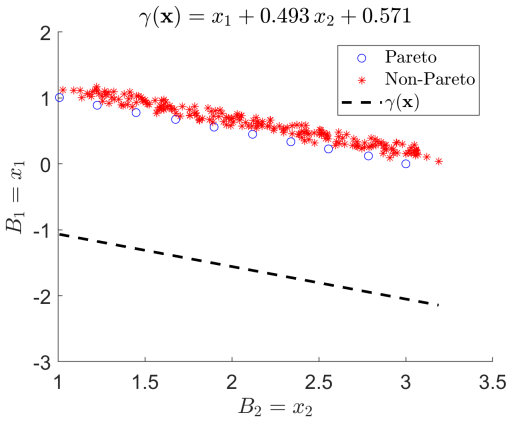
3.4.2 Ablation Studies on the Proposed Bilevel GA

As mentioned before, the upper level of our bilevel algorithm aims at determining optimal power-law structures (i.e. B_is) and the value of modulus flag m. Experiments are performed on DS3 and DS4 datasets to guage the efficacy of upper level (and thus the overall *bilevel*) GA. No prior information about the optimal values of block matrix $\mathbf{B}$ and modulus flag m is supplied. Thus, the structure of the equation of the split-rule is unknown to the algorithm. Results of these experiments are shown in Figs. 3.11a and 3.11b.

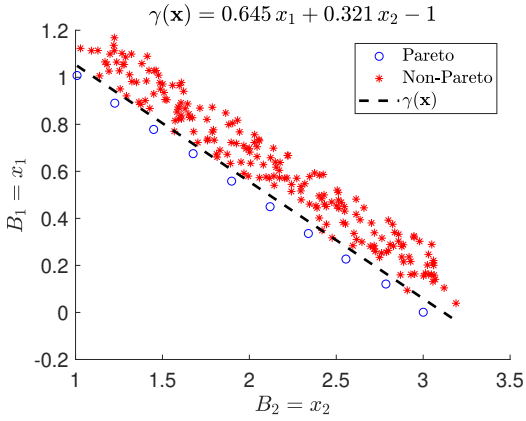
As can be seen from Fig. 3.11a, the proposed bilevel algorithm is able to find quatratic split-rule which is able to partition the DS3 dataset with no Type-I or Type-II error. Results on DS4 dataset validated the importance of involving modulus-flag m as a search variable for upper level GA in



(a) LLGA : (0%, 0%); (0%, 0%)



(b) SVM: (100%, 0%); (100%, 0%)



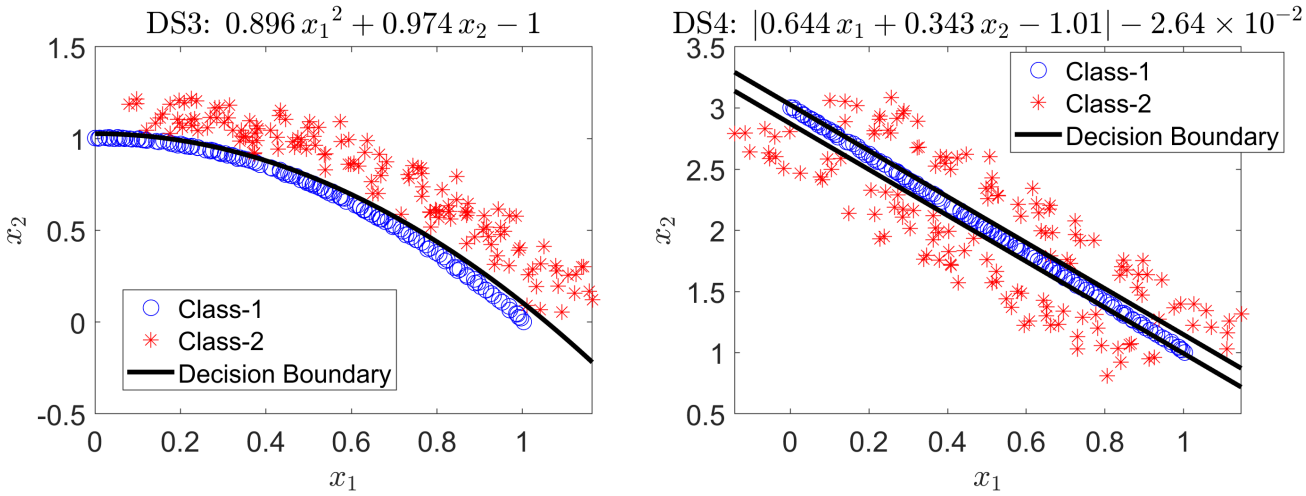
(c) SVM+SMOTE: (0%, 1.5%); (0%, 1.5%)

Figure 3.10: Results on DS2 dataset to benchmark LLGA.

addition to the block-matrix **B**. The algorithm is able to converge at the optimal tree topology (Fig. 3.11b) with 100% classification accuracy.

It is to note here that the algorithm had no prior knowledge about optimal block matrix **B** of exponents b_{ij}. This observation suggests that the upper level GA is successfully able to navigate through the search space of block-matrices **B** and modulus-flag m (which is a binary variable) to arrive at the optimal power-laws and split-rule.

Fig. 3.12 and Fig. 3.13 shows plots of a decision boundary (split-rule) obtained using our bilevel-algorithm on *DS3* and *DS4* datasets, their corresponding NLDTs and its comparison against the decision-tree obtain using traditional CART algorithm on *DS3* and *DS4* datasets.



(a) DS3: (0%, 0%); (0%, 0%)

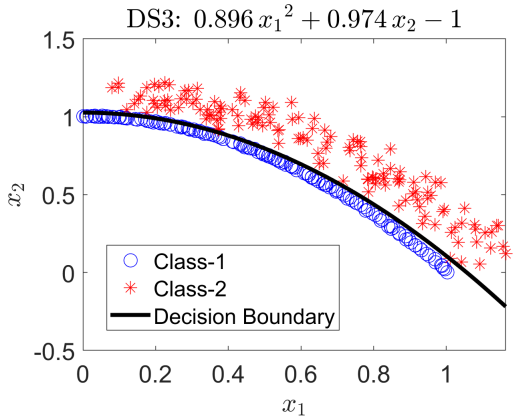
(b) DS4: (0%, 0%); (0%, 0%)

Figure 3.11: Bilevel GA results on DS3 and DS4.

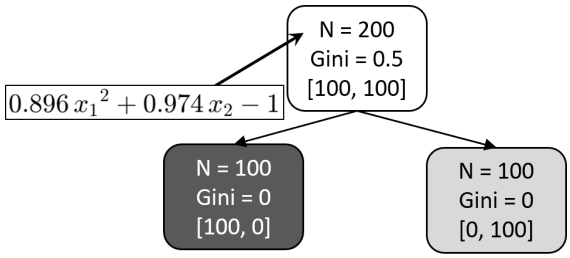
3.5 Visualization of split-rule: X-Space and B-Space

A comprehensible visualization of the decision boundary is possible if dimension of the feature space is up to three. However, if the dimension d of the original feature space (or, *X-space*) is larger than three, the decision-boundary can be visualized on the *transformed-feature-space* (or *B-space*). In our experiments, the maximum number of allowable power-law-rules (p) is fixed to three (i.e. $p = 3$). Thus, any datapoint $\mathbf{x}$ in a d-dimensional *X-space* can be mapped to a three-dimensional *B-space* (Eq. 3.3). A conceptual illustration of this mapping is provided in Fig. 3.14, where, for the sake of simplicity, a three-dimensional *X-space* (x_1 to x_3) is mapped to a two-dimensional B-space using two power-law-rules (B_1 and B_2).

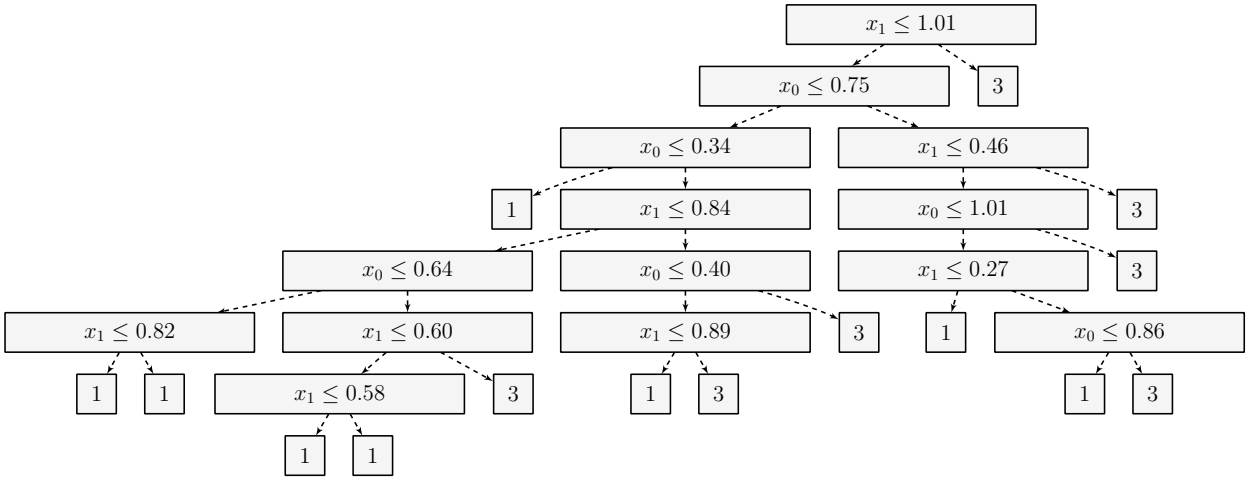
It is to note here that the power-law rules B_i's are not known a priori. The proposed bilevel optimization method determines them so that the data becomes *linearly separable* in B-space. Results and discussions provided in Section 3.7 provide clarity on this novel aspect of our nonlinear decision tree representation.



(a) Bilevel GA results: $(0\%, 0\%)$ and $(1\%, 0\%)$.



(b) Obtained NLDT for DS3 problem – One rule.



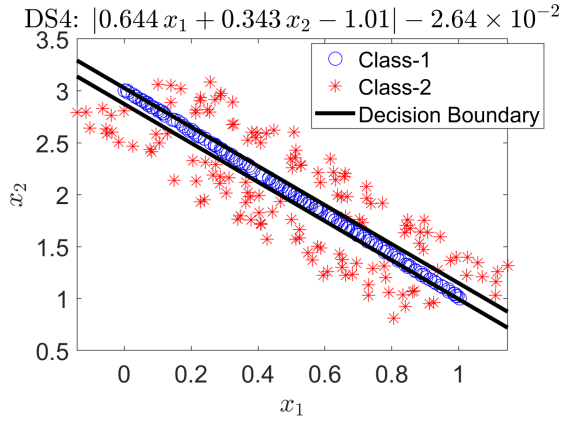
(c) CART results: $(0\%, 4\%)$ and $(4\%, 3\%)$ – 13 rules.

Figure 3.12: Results on DS3 dataset to benchmark the overall bilevel algorithm.

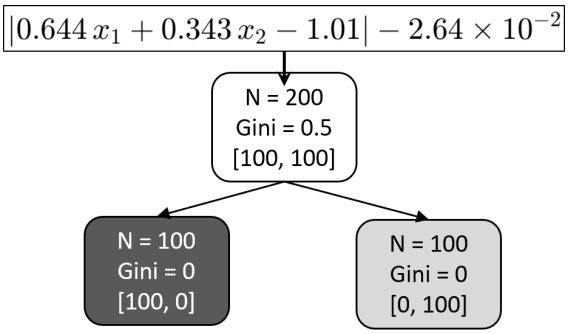
3.6 Overall Tree Induction and Pruning

Once the split-rule for a conditional node is determined using the above bilevel optimization procedure, resulting child nodes are checked for the following termination criteria:

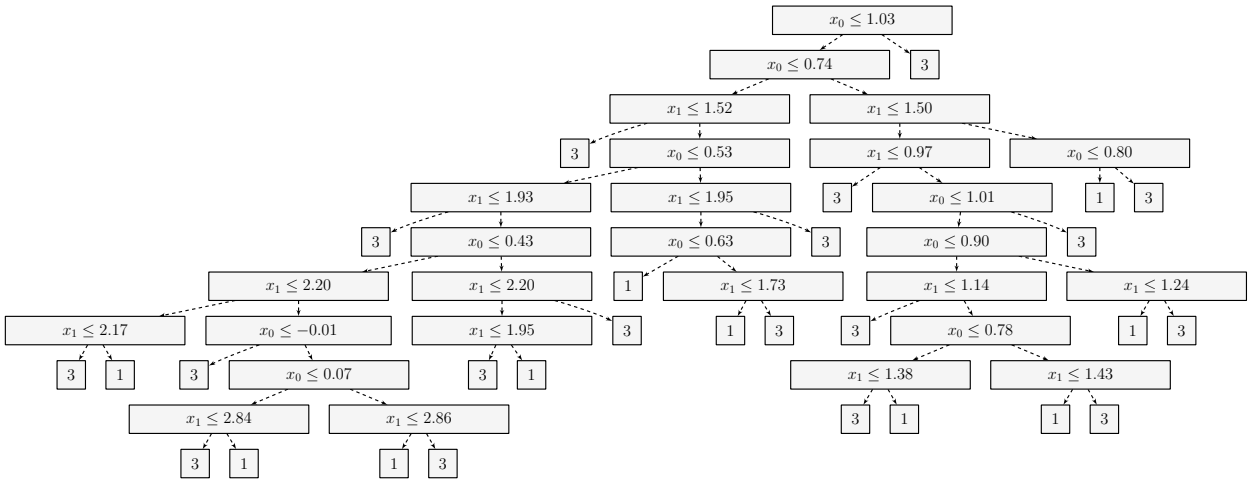
- Depth of the Node > Maximum allowable depth,
- Number of datapoints in node $< N_{min}$, and
- Node Impurity $\leq \tau_{min}$.



(a) Bilevel GA results: $(0\%, 0\%)$ and $(0\%, 0\%)$ – No error.



(b) Obtained NLDT for DS4 problem – One rule.



(c) CART results: $(4\%, 3\%)$ and $(5\%, 18\%)$, 27 rules.

Figure 3.13: Results on DS4 dataset to benchmark the overall bilevel algorithm.

If any of the above criteria is satisfied, then that node is declared as a terminal leaf node. Otherwise, the node is flagged as an additional conditional node. It then undergoes another split (using a split-rule which is derived by running the proposed bilevel-GA on the data present in the node) and the process is repeated. This overall process is illustrated in Algorithm 1. This procedure eventually produces the main nonlinear decision tree $NLDT_{main}$. The fully grown decision tree then undergoes pruning to produce the pruned decision tree $NLDT_{pruned}$.

During the pruning phase, splits after the root-node are systematically removed until the training accuracy of the pruned tree does not fall below a pre-specified threshold value of $\tau_{prune} = 3\%$

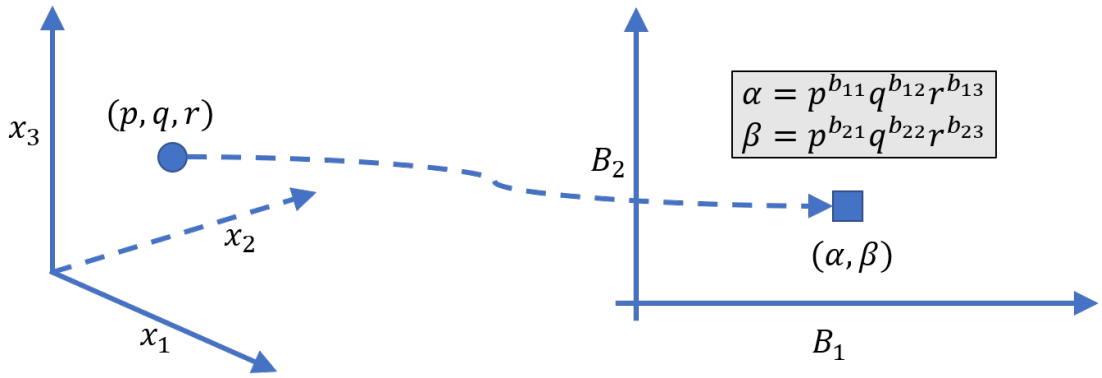


Figure 3.14: Feature Transformation. A point in three-dimensional X-space is mapped to a point in a two-dimensional B-space for which $B_i = x_1^{b_{i1}} x_2^{b_{i2}} x_3^{b_{i3}}$.

(set here after trial-and-error runs). This makes the resultant tree topologically less complex and provides a better generalizability. In subsequent sections, we provide results on final NLDT obtained after pruning (i.e. $NLDT_{pruned}$), unless otherwise specified.

3.7 Results

This section summarizes results obtained on four customized test problems, two real-world classification problems, three real-world optimization problems and eight custom designed multi-objective problems. For each experiment, 50 runs are performed with random training and testing sets. A dataset is split into training and testing sets with a ratio of 7:3, respectively. Mean and standard deviation of *training and testing accuracy-scores* across all 50 runs are evaluated to gauge the performance of the classifier. Statistics about the number of conditional nodes (given by **#Rules**) in NLDT, average number of active terms in split-rules of decision tree (i.e. F_U/Rule) and *rule length* (which gives *total number* of active-terms appearing in the entire NLDT) is provided to quantify the simplicity and interpretability of classifier (lesser this number, simpler is the classifier). The comparison is made against classical CART tree solution [12] and SVM [43] results. For SVM, the Gaussian kernel is used and along with overall accuracy-scores; statistics about the number of support vectors (which is also equal to the rule-length) is also reported. It is to note that the decision boundary computed by SVM can be expressed with a single equation, however the length

of the equation usually increases with the number of support vectors [44]. We used MATLAB's SVM routine with default parameter settings.

For each set of experiments, best scores are highlighted in bold and Wilcoxon signed-rank test is performed on overall testing accuracy scores to check the statistical similarity with other classifiers in terms of classification accuracy. Statistically similar classifiers (which will have their p-value greater than 0.05) are italicized.

3.7.1 Customized Datasets: DS1 to DS4

The compilation of results of 50 runs on datasets DS1-DS4 is presented in Table 3.1. The results clearly indicate the superiority of the proposed nonlinear, bilevel based decision tree approach over classical CART and SVM based classification methods. Bilevel approach finds a single rule with

Table 3.1: Results on DS1 to DS4 datasets (2 features).

Dataset	Method	Training Accuracy	Testing Accuracy	p-value	#Rules	F_U/Rule	Rule Length
DS1	Bilevel	99.78 ± 0.51	**99.55 ± 1.08**	–	**1.0 ± 0.0**	2.3 ± 0.6	**2.3 ± 0.6**
	CART	97.99 ± 0.96	90.32 ± 4.06	7.34e-10	14.5 ± 1.7	**1**	14.5 ± 1.7
	SVM	98.67 ± 0.31	98.50 ± 1.09	1.29e-05	**1**	71.5 ± 3.3	71.5 ± 3.3
DS2	Bilevel	99.80 ± 0.40	**99.44 ± 0.87**	–	**1.0 ± 0.0**	2.3 ± 0.7	**2.3 ± 0.7**
	CART	98.80 ± 0.50	95.43 ± 1.50	5.90e-10	11.0 ± 1.4	1	11.0 ± 1.4
	SVM	96.95 ± 0.73	95.24 ± 0.16	2.43e-10	**1**	44.7 ± 1.9	44.7 ± 1.9
DS3	Bilevel	99.91 ± 0.35	**99.77 ± 0.67**	–	**1.0 ± 0.0**	2.2 ± 0.5	**2.2 ± 0.5**
	CART	99.42 ± 0.57	95.00 ± 2.35	7.00e-10	11.5 ± 1.3	**1**	11.5 ± 1.3
	SVM	98.96 ± 0.42	98.38 ± 1.15	7.16e-09	**1**	62.1 ± 3.4	62.1 ± 3.4
DS4	Bilevel	99.34 ± 1.21	**98.88 ± 1.65**	–	1.2 ± 0.4	2.6 ± 0.6	**3.1 ± 1.4**
	CART	96.98 ± 1.19	88.68 ± 3.60	7.31e-10	31.3 ± 4.2	**1**	31.3 ± 4.2
	SVM	98.19 ± 0.44	97.28 ± 1.19	1.31e-05	**1**	89.8 ± 3.3	89.8 ± 3.3

two to three appearances of variables in the rule, whereas CART requires 11 to 31 rules involving a single variable per rule, and SVM requires only one rule but involving 44 to 90 appearances of variables in the rule. Moreover, the bilevel approach achieves this with the best accuracy. Thus, classifiers obtained by bilevel approach are more simplistic and simultaneously more accurate.

3.7.2 Breast Cancer Wisconsin Dataset

This problem was originally proposed in 1991. It has two classes: *benign* and *malignant*, with 458 (or 65.5%) datapoints belonging to *benign* class and 241 (or 34.5%) belonging to *malignant* class. Each datapoint is represented with 10 attributes. Results are tabulated in Table 3.2. The bilevel method and SVM had similar performance (*p-value*> 0.05), but SVM requires about 90 variable appearances in its rule, compared to only about 6 variable appearances in the single rule obtained by the bilevel approach. The proposed approach outperformed the classifiers generated using techniques proposed in [20, 25, 24, 21, 22] in terms of both: *accuracy* and *comprehensibility/compactness*. The NLDT classifier obtained by a specific run of the bilevel approach has five variable appearances and is presented in Fig. 3.15.

Table 3.2: Results on breast cancer Wisconsin dataset (10 features).

Method	**Training Accuracy**	**Testing Accuracy**	**p-value**	**#Rules**	**F_U/Rule**	**Rule Length**
Bilevel	98.07 ± 0.39	*96.50 ± 1.16*	*0.308*	**1.0 ± 0.0**	6.4 ± 1.7	**6.4 ± 1.7**
CART	98.21 ± 0.49	94.34 ± 1.92	8.51e-09	11.6 ± 2.4	**1**	11.6 ± 2.4
SVM	97.65 ± 0.39	**96.64 ± 1.16**	–	**1**	89.4 ± 14.8	89.4 ± 14.8

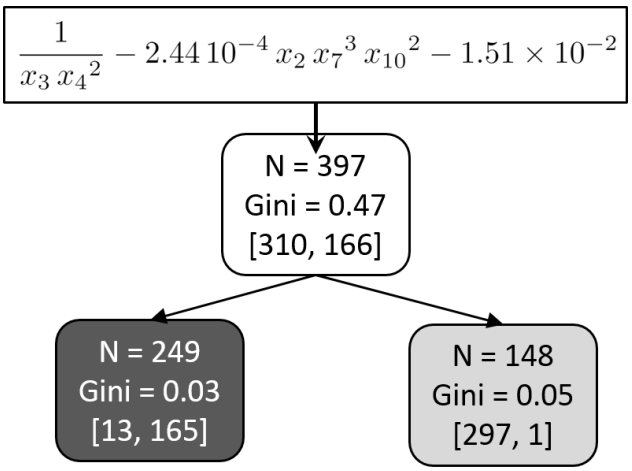


Figure 3.15: Breast Cancer Wisconsin NLDT. For each node, number of datapoints *N* present in the node, impurity of the node (*Gini*) and class distribution (in square parenthesis) is reported.

Figure 3.16 provides B-Space visualization of decision-boundary obtained by the bilevel approach, which is able to identify two nonlinear *B*-terms involving variables to split the data linearly

to obtain high accuracy.

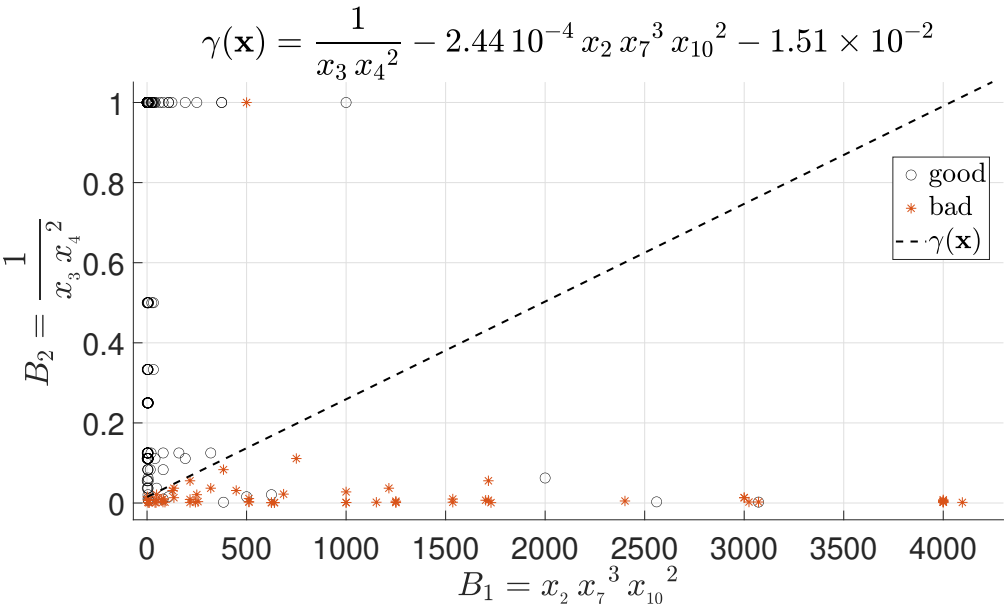


Figure 3.16: B-space plot for Wisconsin breast cancer dataset.

3.7.3 Wisconsin Diagnostic Breast Cancer Dataset (WDBC)

This dataset is an extension to the dataset of the previous section. It has 30 features with total 356 datapoints belonging to *benign* class and 212 to *malign* class. Results shown in Table 3.3 indicate that the bilevel-based NLDT is able to outperform standard CART and SVM algorithms. The NLDT generated by a run of the bilevel approach requires seven out of 30 variables (or features) and is shown in Fig. 3.17. It is almost as accurate as that obtained by SVM and is more interpretable (seven versus about 107 variable appearances).

Table 3.3: Results on WDBC dataset (30 features).

Method	**Training Accuracy**	**Testing Accuracy**	**p-value**	**#Rules**	**F_U/Rule**	**Rule Length**
Bilevel	98.24 ± 0.64	96.20 ± 1.49	4.65e-05	**1.0 ± 0.0**	9.2 ± 4.1	**9.2 ± 4.1**
CART	98.76 ± 0.60	92.11 ± 2.07	1.09e-09	10.8 ± 2.1	**1**	10.8 ± 2.1
SVM	98.65 ± 0.37	**97.39 ± 1.37**	–	**1**	106.7 ± 6.6	106.7 ± 6.6

The *B*-space plot (Fig. 3.18) shows an efficient discovery of *B*-functions to linearly classify the supplied data with a high accuracy.

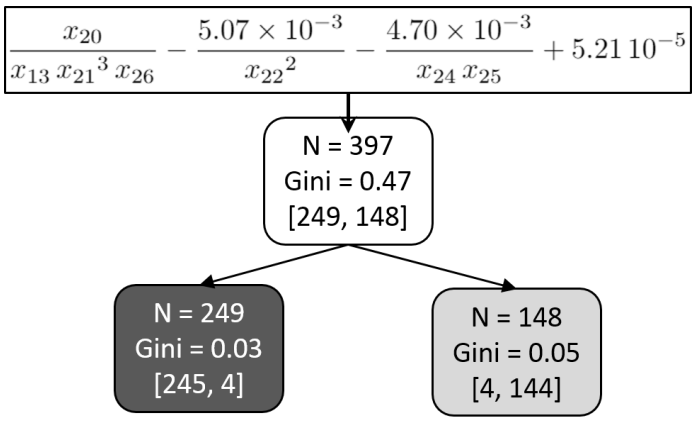


Figure 3.17: Tree for WDBC dataset. For each node, number of datapoints *N* present in the node, impurity of the node (*Gini*) and class distribution (in square parenthesis) is reported.

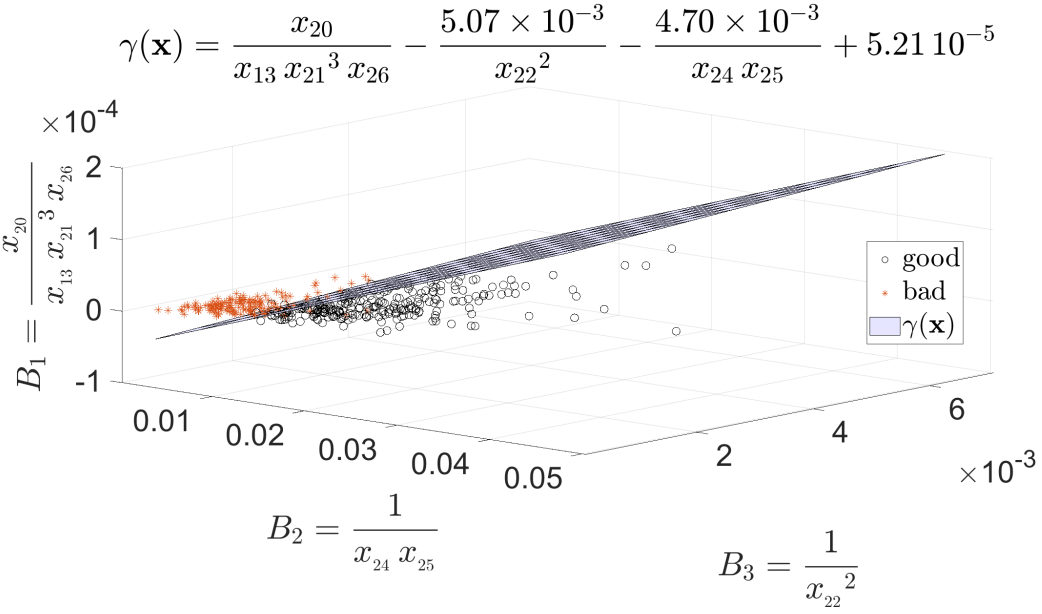


Figure 3.18: B-space plot for WDBC dataset.

3.7.4 Real World Auto-Industry Problem (RW-problem)

This real-world engineering design optimization problem has 36 variables, eight constraints, and one objective function. The dataset is highly biased, with 188 points belonging to the *good-class* and 996 belonging to the *bad-class*. Results obtained using the bilevel GA are shown in Table 3.4. The proposed algorithm is able to achieve near 90% accuracy scores requiring only two split-rules. The best performing NLDT has the testing-accuracy of 93.82% and is shown in Fig. 3.19.

SVM performed the best in terms of accuracy, but the resulting classifier is complicated with

about 241 variable appearances in the rule. However, bilevel GA requires only about two rules, each having about only 10 variable appearances per rule to achieve slightly less-accurate classification. CART requires about 30 rules with a deep DT, making the classifier difficult to interpret easily.

Table 3.4: Results on the real-world auto-industry problem (36 features).

Method	**Training Accuracy**	**Testing Accuracy**	**p-value**	**#Rules**	**F_U/Rule**	**Rule Length**
Bilevel	94.36 ± 1.47	89.93 ± 2.04	4.35e-09	1.9 ± 0.5	10.0 ± 2.9	**18.2 ± 5.9**
CART	98.00 ± 0.55	91.13 ± 1.32	9.80e-08	29.6 ± 3.9	**1**	29.6 ± 3.9
SVM	94.98 ± 0.73	**93.24 ± 1.38**	–	**1**	240.8 ± 9.4	240.8 ± 9.4

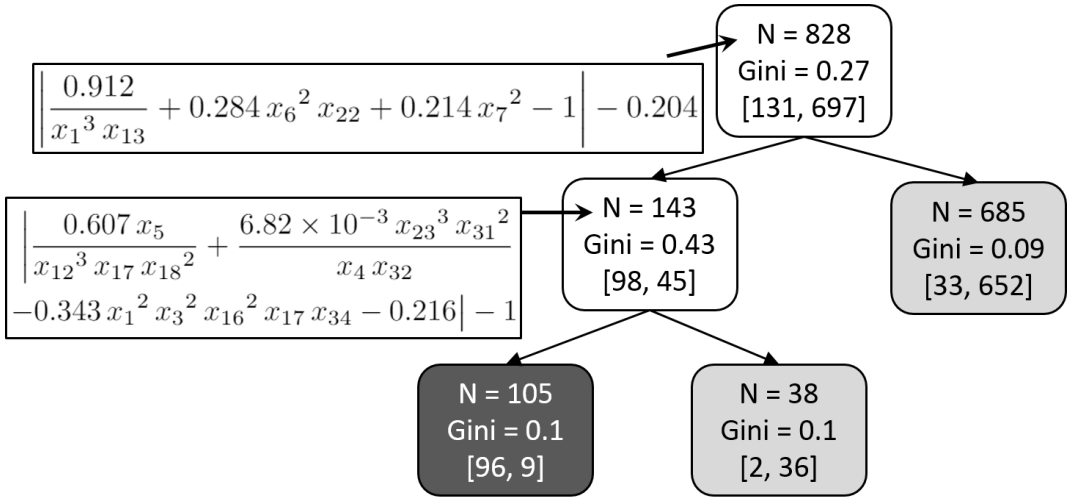


Figure 3.19: NLDT for the auto-industry problem. The first split-rule uses five variables and the second one uses 12. For each node, number of datapoints *N* present in the node, impurity of the node (*Gini*) and class distribution (in square parenthesis) is reported.

3.7.5 Results on Multi-Objective Optimization Problems

After bench-marking the proposed bilevel GA algorithm on standard classical benchmarks and a single-objective engineering problem, we now evaluate its performance on two real-world multi-objective problems: *welded-beam design* and *2D truss design* and eight modified ZDT and DTLZ problems. We will briefly discuss the procedure used to generate datasets corresponding to these problems followed by results obtained on them using our approach.

3.7.5.1 Truss 2D and Welded Beam Problems

Data creation:

For each problem, NSGA-II [45] algorithm is first applied to evolve the population of N individuals for g_{max} generations. Population for each generation is stored. Naturally, population from later generations are closer to Pareto-front than the population of initial generations. We artificially separate the entire dataset into two classes – good and bad – using two parameters g_{ref} (indicating an intermediate generation for data collection) and τ_{rank} (indicating the minimum non-dominated rank for defining the bad cluster). First, population members from g_{ref} and g_{max} generations are merged. Next, non-dominated sorting is executed on the merged population to determine their non-domination rank. Points belonging to same non-domination rank are further sorted based on their crowding-distance (from highest to lowest) [45]. For the good-class, top N_A points from rank-1 front are chosen and for the bad-class, top N_B points from τ_{rank} front onward are chosen. Increasing the τ_{rank} increases the separation between good and bad classes, while the g_{ref} parameter has the inverse effect. Parameter setting for NSGA-II algorithm which is used to generate multi-objective datasets can be found in Section 3.10.

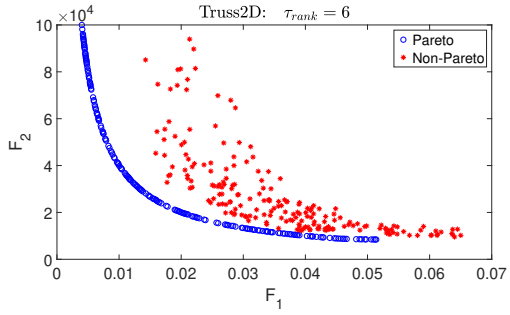
Truss 2D Results:

Truss 2D problem [46] is a three-variable multi objective problem involving one constraint. Visualization of the Truss dataset in the *F-space* and *X-space* is provided in Figure 3.20 for $\tau_{rank} = 6$ and $\tau_{rank} = 9$, with $g_{ref} = 1$.

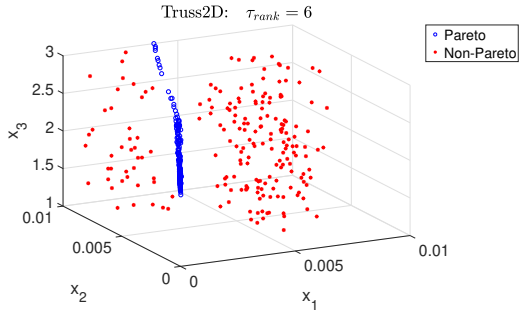
Compilation of results obtained using $\tau_{rank} = 6$ is provided in Table 3.5.

Table 3.5: Results on Truss-2D with $\tau_{rank} = 6$.

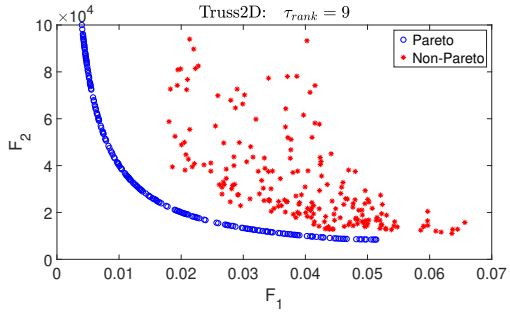
Method	**Training Accuracy**	**Testing Accuracy**	**p-value**	**#Rules**	**F_U/Rule**	**Rule Length**
Bilevel	99.77 ± 0.72	**99.54 ± 0.75**	–	1.2 ± 0.5	2.9 ± 0.5	**3.3 ± 0.9**
CART	99.34 ± 0.32	98.33 ± 1.10	6.04e-08	11.06 ± 3.15	**1**	11.06 ± 3.15
SVM	99.66 ± 0.15	*99.46 ± 0.50*	*0.135*	**1**	62.5 ± 2.9	62.5 ± 2.9



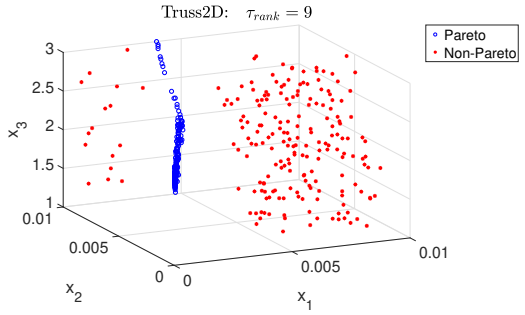
(a) F-space plot with $\tau_{rank} = 6$.



(b) X-space plot with $\tau_{rank} = 6$.



(c) F-space plot with $\tau_{rank} = 9$.



(d) X-space plot with $\tau_{rank} = 9$.

Figure 3.20: Truss design problem data visualization. $g_{ref} = 1$ is kept fixed. For a fixed value of g_{ref}, larger value of τ_{rank} implies better separation between datapoints belonging to two classes.

Clearly, the bilevel NLDT has the best accuracy and fewer variable appearances (meaning easier intrepretability) compared to CART and SVM generated classifiers.

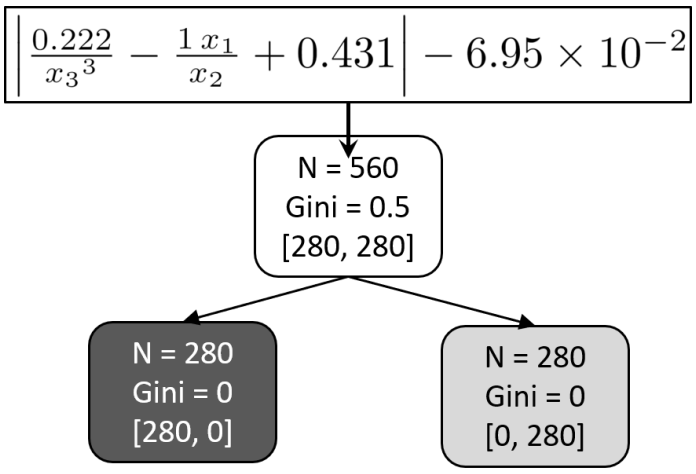
Fig. 3.21a shows a 100% correct classifier with a single rule obtained by our bilevel NLDT, whereas Fig. 3.21b shows a typical CART classifier with 19 rules, making it difficult to interpret easily.

A comparison of results obtained using NLDT approach on truss problem with $\tau_{rank} = 6$ and $\tau_{rank} = 9$ is provided in Table 3.6.

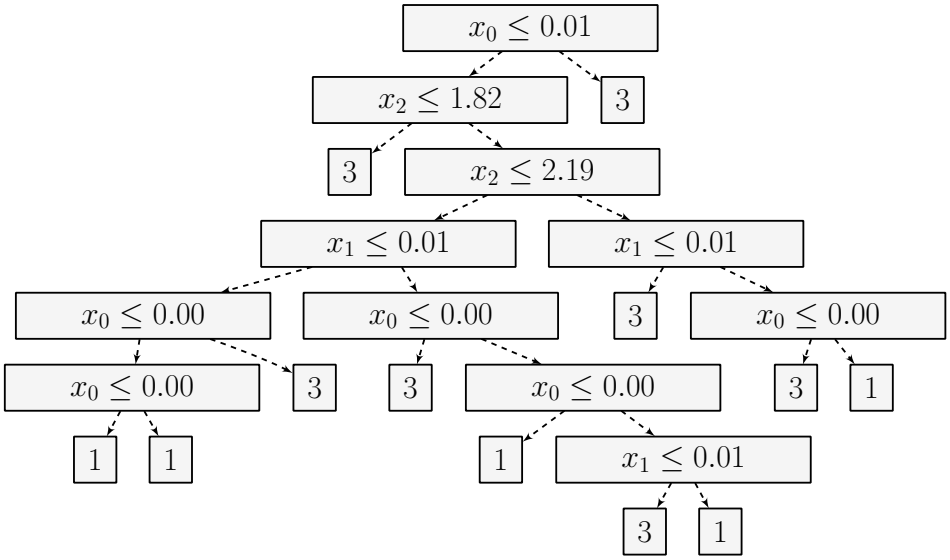
Table 3.6: 2D Truss problem with $\tau_{rank} = 6$ and $\tau_{rank} = 9$.

τ_{rank}	**Training Accuracy**	**Testing Accuracy**	**#Rules**	**F_U/Rule**
6	99.86 ± 0.36	99.6 ± 0.58	1.20 ± 0.53	2.92 ± 0.52
9	100 ± 0	99.92 ± 0.19	1.36 ± 0.48	2.26 ± 0.52

Clearly, since the data is more separated for $\tau_{rank} = 9$, the results for $\tau_{rank} = 9$ are more accurate and relatively simpler.



(a) Bilevel classifier requiring only one rule.



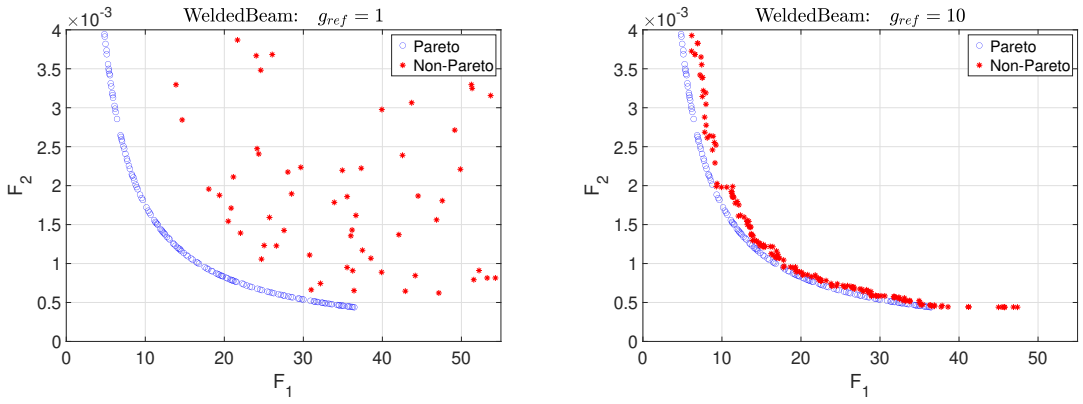
(b) CART classifier with 11 rules.

Figure 3.21: Comparison of bilevel and CART methods on truss problem with τ_{rank} = 6 dataset.

Welded Beam Design Problem Results:

This bi-objective optimization problem has four variables and four constraints [46]. Here, two sets of experiments are conducted for two different values of g_{ref}, keeping the value of τ_{rank} fixed to 3. Visualization of these datasets in the objective space (or *F-space*) is provided in Figure 3.22.

A higher value of g_{ref} results in *good* and *bad* datapoints too close to each other (Figure 3.22b), thereby making it difficult for the classification algorithm to determine a suitable classifier. This can be validated from the results presented in Table 3.7. However, bilevel NLDTs have produced



(a) F-space plot with $g_{ref} = 1$.

(b) F-space plot with $g_{ref} = 10$.

Figure 3.22: Welded Beam Design Problem data visualization. $\tau_{rank} = 3$ is kept fixed. Problem at (b) is more difficult to solve than at (a).

one rule with about 2 to 4 variable appearances compared to 39.5 to 126 (on average) for SVM classifiers. CART does well in this problem with, on average, 2.94 to 8.42 rules. The NLDT (having a single rule) obtained with $g_{ref} = 10$ is shown in Fig. 3.23. Interestingly, the bilevel NLDTs have similar accuracy to that of CART and SVM.

Table 3.7: Results on welded beam design with $g_{ref} = 1$ and $g_{ref} = 10$. $\tau_{rank} = 3$ is kept fixed.

Method	**Training Accuracy**	**Testing Accuracy**	**p-value**	**#Rules**	**F_U/Rule**	**Rule Length**
$g_{ref} = 1$						
Bilevel	99.98 ± 0.06	*99.38 ± 0.49*	*0.158*	**1.0 ± 0.0**	2.6 ± 0.7	**2.6 ± 0.7**
CART	99.97 ± 0.07	**99.50 ± 0.59**	–	2.94 ± 0.71	**1**	2.94 ± 0.71
SVM	99.37 ± 0.17	*99.40 ± 0.40*	*0.331*	**1**	39.5 ± 2.5	39.5 ± 2.5
$g_{ref} = 10$						
Bilevel	99.39 ± 0.38	98.58 ± 1.13	3.42e-02	**1.0 ± 0.0**	3.9 ± 1.0	**3.9 ± 1.0**
CART	99.46 ± 0.27	97.72 ± 1.04	4.63e-08	8.42 ± 1.42	**1**	8.42 ± 1.42
SVM	99.46 ± 0.19	**98.97 ± 0.54**	–	**1**	126.0 ± 6.8	126.0 ± 6.8

3.7.5.2 Modified ZDT (m-ZDT) and DLTZ (m-DTLZ) Problems

Problem Definition and Dataset Creation:

These are the modified versions of ZDT and DLTZ problems [45, 47]. For m-ZDT problems, the *g-function* is modified to the following:

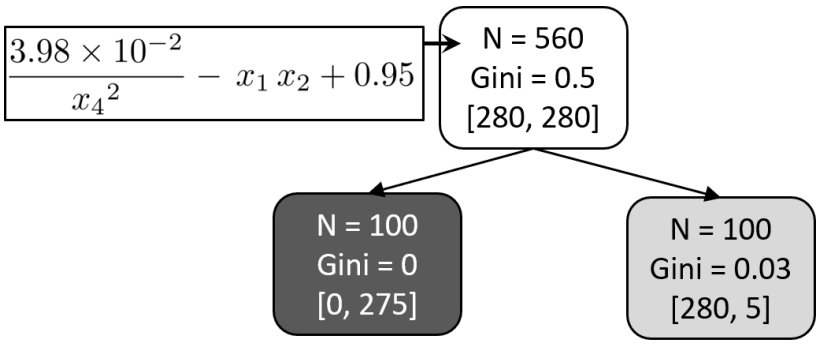


Figure 3.23: Welded beam classifier with one rule for $g_{ref} = 10$.

$$g_{zdt}(x_2, \ldots, x_n) = 1 + \frac{18}{n-1} \sum_{i\,=\,2 \text{ and even}}^{n} (x_i + x_{i+1} - 1)^2. \tag{3.15}$$

Many two-variable relationships ($x_i + x_{i+1} = 1$) must be set to be on the Pareto set.

For m-DTLZ problems, the *g-function* for $\mathbf{x_m}$ variables is

$$g_{dtlz}(\mathbf{x_m}) = 100 \times \sum_{x_i \,\in\, \mathbf{x_m} \text{ and } i \text{ is even}}^{n} (x_i + x_{i+1} - 1)^2. \tag{3.16}$$

Pareto points for m-ZDT and m-DTLZ problems are generated by using the exact analytical expression of Pareto-set (locations where $g = 0$). The non-Pareto set is generated by using two parameters σ_{spread} and σ_{offset}. To compute the location of a point $\mathbf{x_{np}}$ belonging to non-Pareto set from the location of the point $\mathbf{x_p}$ on the Pareto set, following equation is used:

$$x_{np}^{(i)} = x_p^{(i)} + r_1(\sigma_{offset} + r_2\sigma_{spread}) \tag{3.17}$$

$$\text{where } \; r_1 \in -1, 1, \quad r_2 \in [0, 1]. \tag{3.18}$$

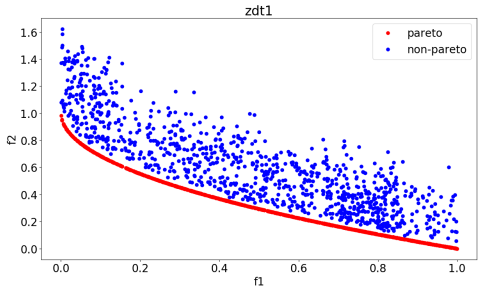
Here, r_1 and r_2 are randomly generated for each $x_{np}^{(i)}$. For m-ZDT problems, $i = 1, 2, \ldots n$,

Table 3.8: Parameter setting to generate datasets for m-ZDT and m-DTLZ problems. We generate 1000 datapoints for each class.

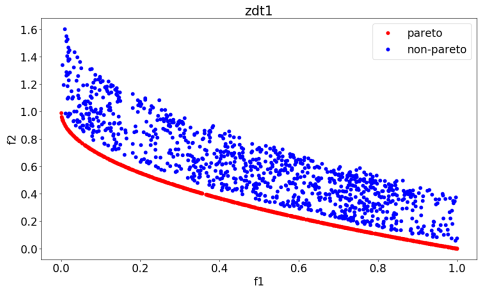
Prob.	$\mathbf{n_{vars}}$	$\sigma_{\mathbf{spead}}$	$\sigma_{\mathbf{offset}}$
m-ZDT1	30	0.3	0.1
m-ZDT1	500	0.3	0.1
m-ZDT2	30	0.3	0.1
m-ZDT2	500	0.3	0.1
m-DTLZ1	30	0.05	0
m-DTLZ1	500	0.05	0
m-DTLZ2	30	0.05	0
m-DTLZ2	500	0.05	0

while for m-DTLZ problems $x_{np}^{(i)}, x_{p}^{(i)} \in \mathbf{x_m}$. Parameter setting for generating m-ZDT and m-DTLZ datasets is provided in Table 3.8. For 30 variable problems, all $x_i \in \mathbf{x_m}$ are changed according to Eq. 3.18 for generate non-pareto points. However, for 500 vars problems, only first 28 of variables in $\mathbf{x_m}$ are modified according to Eq. 3.18 to generate non-pareto data-points.

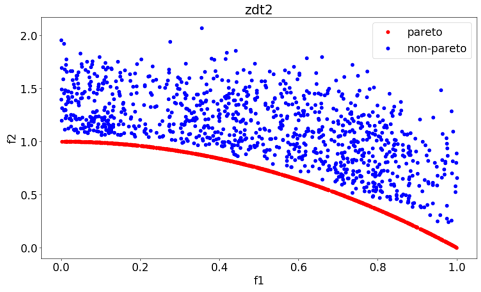
Visualization of Datasets for m-ZDT and m-DTLZ problems in provided in Figure 3.24 and Figure 3.25.



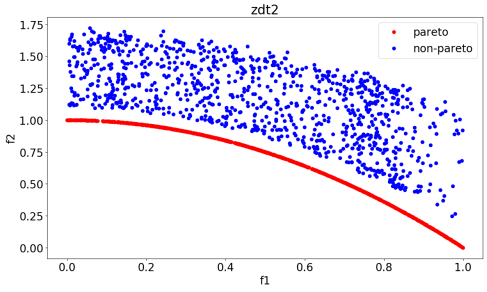
(a) m-ZDT1, 30 vars



(b) m-ZDT1, 500 vars

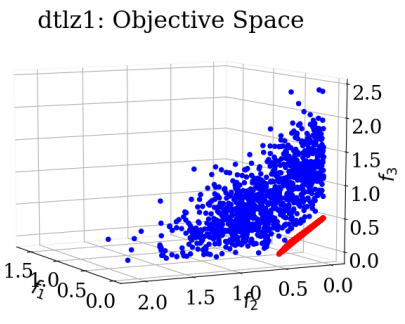


(c) m-ZDT2, 30 vars

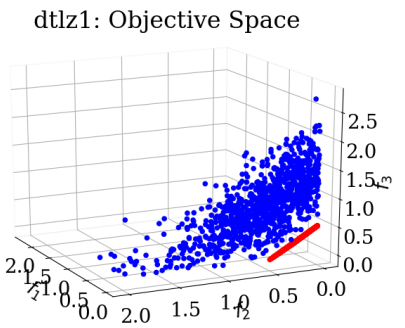


(d) m-ZDT2, 500 vars

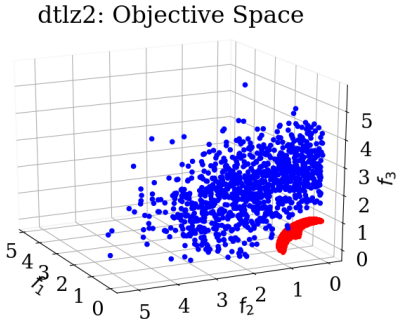
Figure 3.24: m-ZDT Datasets.



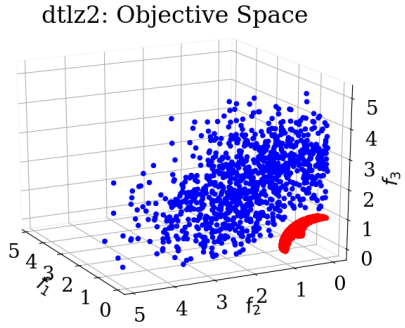
(a) m-DTLZ1, 30 vars



(b) m-DTLZ1, 500 vars



(c) m-DTLZ2, 30 vars



(d) m-DTLZ2, 500 vars

Figure 3.25: m-DTLZ Datasets.

3.7.5.3 m-ZDT and m-DTLZ Results:

Experiments conducted on datasets involving 500 features (see Table 3.9) and for two and three-objective optimization problems confirm the scalability aspect of the proposed approach. In all these problems, traditional methods like CART and SVM find it difficult to conduct a proper classification task. The provision of allowing controlled non-linearity at each conditional-node provides the proposed NLDT approach with necessary flexibility to make an appropriate overall classification.

Table 3.9: Results on multi-objective problems for classifying dominated and non-dominated solutions.

Method	Training Acc.	Testing Acc.	# Rules	F_U/Rule	Rule Length
m-ZDT1-2-30					
NLDT	99.18 ± 0.40	**98.96 ± 0.59**	1.80 ± 0.60	4.47 ± 2.21	7.64 ± 3.50
CART	97.48 ± 0.37	94.78 ± 1.02	26.42 ± 1.89	**1.00 ± 0.00**	26.42 ± 1.89
SVM	84.14 ± 2.42	82.84 ± 2.77	**1.00 ± 0.00**	1192.38 ± 11.34	1192.38 ± 11.34
m-ZDT2-2-30					
NLDT	99.23 ± 0.38	**98.96 ± 0.57**	1.92 ± 0.56	4.37 ± 1.82	8.14 ± 3.36
CART	97.41 ± 0.36	94.72 ± 0.89	27.80 ± 2.19	**1.00 ± 0.00**	27.80 ± 2.19
SVM	99.28 ± 0.18	98.44 ± 0.57	**1.00 ± 0.00**	315.54 ± 6.12	315.54 ± 6.12
m-DTLZ1-3-30					
NLDT	97.21 ± 2.52	**96.65 ± 2.86**	3.12 ± 0.59	6.14 ± 2.18	19.10 ± 7.03
CART	89.72 ± 0.88	71.74 ± 2.49	93.88 ± 3.23	**1.00 ± 0.00**	93.88 ± 3.23
SVM	52.44 ± 0.63	45.86 ± 1.28	**1.00 ± 0.00**	1381.56 ± 8.25	1381.56 ± 8.25
m-DTLZ2-3-30					
NLDT	97.76 ± 1.88	**97.22 ± 2.25**	3.02 ± 0.62	5.81 ± 1.95	17.50 ± 6.73
CART	85.64 ± 3.33	63.41 ± 7.59	100.82 ± 6.11	**1.00 ± 0.00**	100.82 ± 6.11
SVM	54.82 ± 0.98	49.61 ± 1.62	**1.00 ± 0.00**	1367.42 ± 8.44	1367.42 ± 8.44
m-ZDT1-2-500					
NLDT	99.20 ± 0.29	98.93 ± 0.60	1.78 ± 0.54	5.66 ± 3.23	9.36 ± 4.15
CART	98.76 ± 0.27	93.48 ± 0.94	20.58 ± 1.39	**1.00 ± 0.00**	20.58 ± 1.39
SVM	**100.00 ± 0.00**	**100.00 ± 0.00**	**1.00 ± 0.00**	240.88 ± 4.62	240.88 ± 4.62
m-ZDT2-2-500					
NLDT	99.18 ± 0.38	98.88 ± 0.71	1.80 ± 0.69	5.30 ± 2.45	8.88 ± 3.98
CART	98.61 ± 0.36	93.95 ± 1.33	20.98 ± 1.98	**1.00 ± 0.00**	20.98 ± 1.98
SVM	**100.00 ± 0.00**	**100.00 ± 0.00**	**1.00 ± 0.00**	248.06 ± 4.37	248.06 ± 4.37
m-DTLZ1-3-500					
NLDT	94.36 ± 4.03	**93.77 ± 4.24**	3.00 ± 0.45	7.61 ± 2.36	22.76 ± 7.57
CART	83.23 ± 3.52	58.26 ± 7.37	104.88 ± 5.49	**1.00 ± 0.00**	104.88 ± 5.49
SVM	51.56 ± 0.52	47.13 ± 1.19	**1.00 ± 0.00**	1382.38 ± 8.84	1382.38 ± 8.84
m-DTLZ2-3-500					
NLDT	95.89 ± 3.83	**95.33 ± 4.46**	3.04 ± 0.56	7.28 ± 3.16	22.14 ± 10.45
CART	89.09 ± 1.75	70.04 ± 3.46	96.08 ± 3.93	**1.00 ± 0.00**	96.08 ± 3.93
SVM	51.30 ± 0.55	47.01 ± 1.20	**1.00 ± 0.00**	1382.62 ± 8.46	1382.62 ± 8.46

3.8 Additional Comparisons and Results

In previous sections, we focused at fundamental procedure of inducing the NLDT and compared its performance against two standard algorithms: SVM and CART. In this section, we extend our experimental study to compare our approach against other methods which are strong poten-

tial candidates to generate an interpretable AI: *generalized additive models* (GAMs) and *genetic programming* (GP). We also re-iterate experiments using the default parameter settings for NLDT and CART, while we fine tune the results corresponding to SVM using the best possible parameter values.

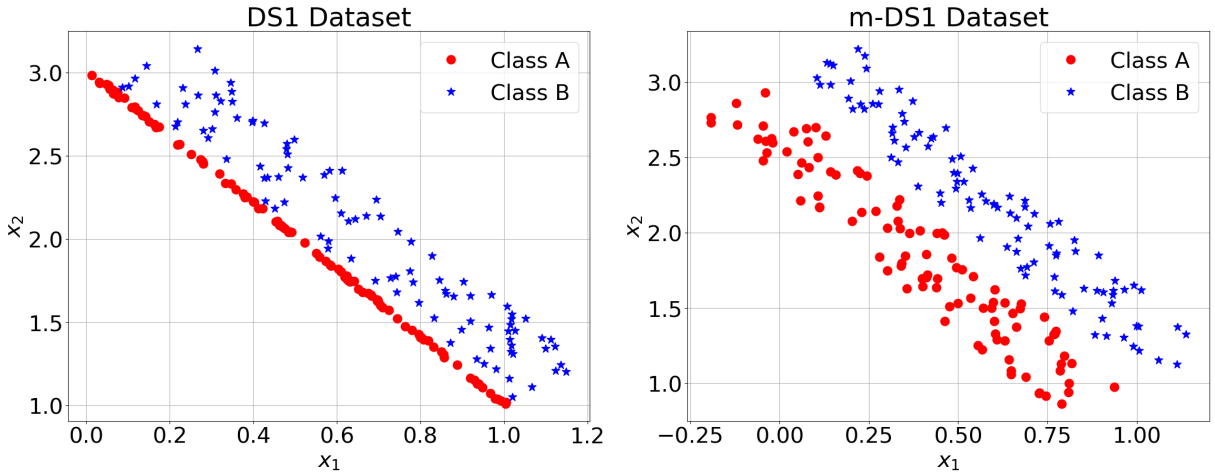
New problems (m-DS1 to m-DS3) are introduced here to obtain further insight into the working principles of above mentioned algorithms. The customized problems DS1 to DS4, m-DS1 to m-DS3, m-ZDTs and m-DTLZs are designed to benchmark the performance of classifiers on various properties of datasets such as:

- **Data Distribution:** For DS1-DS4 datasets, degree of scatter in data varies across classes. For m-DS1, m-DS2 and m-DS3 the scattering of data for each class is more similar than that in original DS datasets. A visualisation of feature spaces for DS1 and m-DS1 dataset is provided in Figure 3.26a and 3.26b, respectively to demonstrate this.

- **Geometry of Decision Boundary:** Here, the effect of the nature of the simplest possible decision boundary is considered. Decision boundary corresponding to DS1-DS2 and modified DS1-DS2 is linear, DS3 and m-DS3 have decision boundary involving nonlinearity of order 2 and DS4 have two disjoint linear decision boundaries.

- **Data Bias:** Here, effect of bias in class representation is considered. All datasets except DS2 and m-DS2 are balanced. For DS2 and m-DS2, minority class has 5 times less number of data points as the majority class.

In next few sections, we briefly discuss SVM, GAM and GP to gain a conceptual insight towards critical parameters which can influence the classification performance and the complexity of the classifier. A more detailed discussion is provided in [48].



(a) DS1 Dataset.

(b) m-DS1 Dataset.

Figure 3.26: Original DS1 and its modified version.

3.8.1 Support Vector Machines (SVMs)

For a separable dataset, support vector machine (SVM) algorithm attempts to derive a decision boundary in the form of a single mathematical equation as shown below:

$$y(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + b, \tag{3.19}$$

where $\boldsymbol{\phi}(\mathbf{x})$ is a set of feature transformation functions which can be either linear or non-linear functions of feature vector $\mathbf{x}$, $\mathbf{w}$ is a weight vector and b is a bias term. A conceptual understanding of SVM is provided in Figure 3.27. For a binary classification task involving class labels $t = -1$ or $t = 1$, an optimal hyper-surface is derived by maximizing the margin between two classes, as shown by $y = 0$ line in the figure. Points with $y \leq -1$ belong to one class and points with $y \geq 1$ belong to another class. The points which fall on $y = 1$ and $y = -1$ are called support vectors, as they alone decide the classifier. However, for non-separable datasets, such as the scenario shown in Figure 3.28, a *soft* margin approach is used to allow some data points within $|y| < 1$ (margin) while training the SVM. These points are also declared as support vectors in addition to the points on the margin.

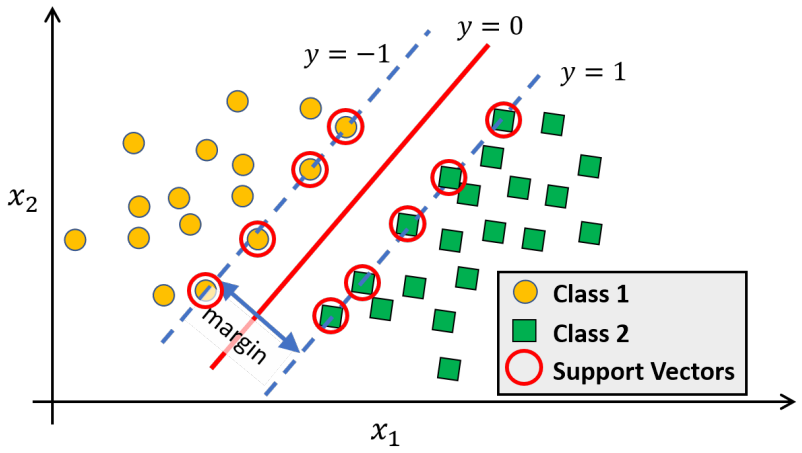


Figure 3.27: SVM on separable datasets with a hard margin.

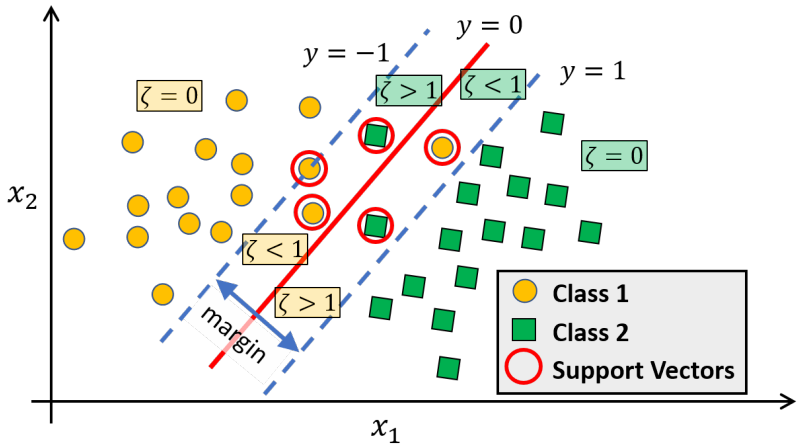


Figure 3.28: SVM with non-separable datasets with a soft margin.

To identify the classifier and the support vectors, the underlying optimization problem is solved:

$$\begin{aligned} \text{Minimize:} \quad & \tfrac{1}{2}||w||^2 + C\sum_{i=1}^{N}\zeta_i, \\ \text{subject to :} \quad & t_i(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i) + b) \geq 1 - \zeta_i, \\ & \zeta_i \geq 0, \quad i = 1, 2, \ldots, N, \end{aligned} \tag{3.20}$$

where t_i is the *true* class label (either 1 or -1) of the datapoint, ζ_i is the distance of i-th data point from its representative margin, thus $\zeta_i = \max\,[0, 1 - t_i y(\mathbf{x}_i)]$ (where value of $y(\mathbf{x}_i)$ is estimated from Eq. 3.19). C is a penalty parameter which is used to enhance *generalizability* by compromising with *training accuracy*. It is also aimed to balance the complexity of the classifier (described

with the number of non-zero terms of $\mathbf{w}$) and soft support vectors within the margin and is an important parameter. With lower values of C, broader margin (with some misclassification of training datapoints) is achieved while for large values of C, misclassification of training datapoints is heavily penalized and so narrower margin is achieved.

Using a kernel trick [49] $k(\mathbf{x}_p, \mathbf{x}_q) = \boldsymbol{\phi}(\mathbf{x}_p)^T \boldsymbol{\phi}(\mathbf{x}_q)$ Eq. 3.19 is transformed into the following:

$$y(\mathbf{x}) = \sum_{i=1}^{N} a_i t_i k(\mathbf{x}, \mathbf{x}_i) + b, \tag{3.21}$$

where a_i is a Lagrange multiplier which is obtained by converting the optimization problem of maximizing the margin (Eq. 3.20) to a dual Lagrangian representation [49]:

$$\begin{aligned} \text{Min:} \quad & L(\mathbf{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j t_i t_j k(\mathbf{x}_i, \mathbf{x}_j), \\ \text{s.t.} \quad & a_i \in [0, C], \quad \sum_{i=1}^{N} a_i t_i = 0. \end{aligned} \tag{3.22}$$

Classical gradient based algorithms can then be employed to find a_i. In Eq. 3.21, data points $\mathbf{x}_i$ for which $a_i = 0$ do not contribute in the equation of the split rule (Eq. 3.21) and data points for which $a_i > 0$ are called support vectors for the SVM classifier and they dictate the overall length of the classifier's equation (Eq. 3.21). The penalty parameter C has to be tuned to efficiently derive the decision boundary. Lower value of C makes the classifier more generalizable. $C = \infty$ (hard margin) attempts to achieve near 100% training accuracy and hence is prone to overfitting. In our case, we use scikit-learn's [50] SVM module and set $C = 1,000$. We use RBF (or Gaussian) kernel function. Table 3.10 shows results for various settings of C on some datasets considered in our study.

3.8.2 Generalized Additive Models (GAMs)

For a binary classification task involving two classes: Class 1 ($y = 0$) and Class 2 ($y = 1$), the GAM based classifier [5, 6] estimates the probability of a data point belonging to class $y = 1$ (i.e.

Table 3.10: SVM Result for different values of penalty parameter C. For each dataset, the first row represents the testing accuracy and the second row represents complexity (number of support vectors). $C = 1000$ gives overall best performance.

Pen. Param.	**DS1**	**DS2**	**DS3**	**DS4**
$C = 1$	94.75 ± 1.97 191.94 ± 4.38	95.24 ± 0.00 16.56 ± 0.80	96.93 ± 1.87 64.68 ± 2.16	45.20 ± 4.24 138.76 ± 1.99
$C = 10$	98.42 ± 1.16 58.70 ± 2.90	95.24 ± 0.00 30.46 ± 0.64	99.32 ± 0.70 26.68 ± 1.88	68.77 ± 4.00 262.60 ± 4.76
$C = 1,000$	**99.88 ± 0.33** **8.36 ± 0.87**	**99.70 ± 0.50** **8.56 ± 0.67**	**99.75 ± 0.58** **10.60 ± 0.92**	**96.63 ± 1.35** **56.70 ± 3.13**

Pen. Param.	**m-DS1**	**m-DS2**	**m-DS3**	**Cancer-10**
$C = 1$	99.77 ± 0.67 70.22 ± 2.23	95.24 ± 0.00 16.18 ± 0.59	99.97 ± 0.23 36.54 ± 1.72	**97.15 ± 1.08** 69.98 ± 6.51
$C = 10$	**100.00 ± 0.00** 26.42 ± 1.46	98.89 ± 0.85 14.40 ± 0.89	**100.00 ± 0.00** 12.60 ± 0.98	95.98 ± 1.13 56.22 ± 6.43
$C = 1,000$	99.93 ± 0.33 **7.38 ± 0.75**	**99.97 ± 0.22** **5.34 ± 0.55**	**100.00 ± 0.00** **8.82 ± 1.01**	95.23 ± 1.09 **52.36 ± 4.91**

Pen. Param.	**Truss**	**Welded Beam**	**Cancer-30**
$C = 1$	77.31 ± 2.16 343.76 ± 9.51	98.83 ± 0.70 47.26 ± 3.00	90.83 ± 1.83 106.88 ± 4.44
$C = 10$	81.29 ± 2.19 258.52 ± 10.85	99.53 ± 0.42 17.86 ± 1.73	91.94 ± 1.36 81.66 ± 4.54
$C = 1,000$	**88.54 ± 1.60** **176.22 ± 7.87**	**99.63 ± 0.38** **7.88 ± 0.86**	**95.08 ± 1.65** **58.74 ± 5.18**

$P(y = 1|\mathbf{x}))$[6] as $\hat{y}(\mathbf{x})$ using the following equation

$$\hat{y}(\mathbf{x}) = \frac{1}{1 + e^{-g(\mathbf{x})}}, \tag{3.23}$$

where $g(\mathbf{x})$ is referred to as *link function* [51]. The link function $g(\mathbf{x})$ in GAM is expressed as a sum of non-linear functions as shown below:

$$g(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x}) + \cdots + f_M(\mathbf{x}) + \beta_0, \tag{3.24}$$

[6] probability of datapoint belonging to other class (i.e. $y = 0$) will be $1 - \hat{y}$.

where β_0 is a constant and $f_i(\mathbf{x})$ are scalar valued nonlinear functions. The functional form of $f_i(\mathbf{x})$ and total number of such nonlinear functions is pre-specified by the user. Modelling of link function $g(\mathbf{x})$ using Eq. 3.24 makes GAMs more generalizable than its precursor: generalized linear models (GLMs) [4], which involves only linear terms.

In our experiments, we use penalized B-splines to model non-linearity of each feature separately (i.e. referring to Eq. 3.24, $f_i(\mathbf{x}) = s_i(x_i)$). Thus, the g-function in our case is given by

$$\begin{aligned} g(\mathbf{x}) &= s_1(x_1) + s_2(x_2) \dots s_d(x_d) + \beta_0, \\ \text{where:} \quad s_i(x_i) &= \sum_{j=1}^{K_i} B_j^{(q_i)}(x_i)\beta_j = \mathbf{B}_i'(x_i)\boldsymbol{\beta}_i. \end{aligned} \tag{3.25}$$

Here, $s_i(x_i)$ denotes a spline function corresponding to i-th feature, $B_j^{(q_i)}(x_i)$ indicates the basis function of order q_i, β_j are scalar coefficients and K_i is the total number of basis functions used to model the spline. The order of spline (i.e. q_i) and the number of basis-functions K_i is user-specified.

Once the structure of link function $g(\mathbf{x})$ is specified, an optimization algorithm is invoked to learn parameters corresponding to basis functions $B_j^{(q_i)}(x_i)$ and coefficients β_j with an objective to minimize the error between the estimated value of probability ($\hat{y}(\mathbf{x})$ Eq. 3.23) and the actual y values across the dataset. To make the resulting model more generalize and simple, a second-order smoothing is employed. Thus, using Eq. 3.23 and 3.25, the overall optimization problem translates to minimizing the following function:

$$\text{Min: } F(\mathbf{B}', \boldsymbol{\beta}) = \sum_{i=1}^{N}(y_i - \hat{y}_i(\mathbf{B}'\boldsymbol{\beta}))^2 + \sum_{j=1}^{d}\lambda_j \int (s_j''(x_j|_{\mathbf{B}_j'\boldsymbol{\beta}_j}))^2 dx_j, \tag{3.26}$$

where y_i is the actual class of the i-th datapoint (which can have value of either 0 or 1) and $\hat{y}_i$ is the probability of i-th point belonging to class $y = 1$ (i.e. $P(y = 1|\mathbf{x_i})$) as predicted by the GAM classifier using Eq. 3.23. λ_j are the penalty parameters which are prespecified. In our case, we use $\lambda_j = 0.6$ for all features. The rule complexity of a GAM classifier can be tuned using λ_j, where higher values of λ_j imposes heavy penalty on non-linearities with more than second order. Additionally, the complexity can also be controlled by regulating the *degree* (q_i) and *number of*

basis-functions K_i (Eq. 3.25). In our experimental setup, we conduct series of experiments using different combinations of (K_i, q_i) to model splines for each feature. Values of K and q are picked from the one listed in Table 3.11.

Table 3.11: Details regarding parametric study for GAMs.

# Basis Functions (K)	**Degree (q)**
2, 3, 5, 8, 13, 21	2, 3, 5

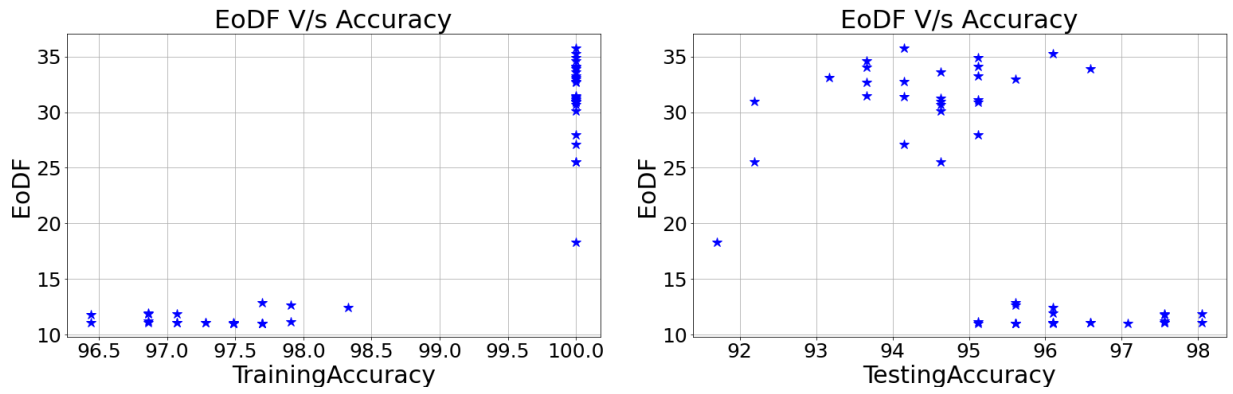
Total number of terms arising from the expression of rule $g(\mathbf{x})$ (Eq. 3.25) is

$$\textit{Total Terms} = \sum_{j=1}^{d} (q_j + 1) \times K_j + K_j + 1. \tag{3.27}$$

However, due to second-order smoothening effect (Eq. 3.26), non-linearities greater than 2nd order which are not contributing in minimizing the error $\sum_{i=1}^{N}(y_i - \hat{y}_i(\mathbf{B}'\boldsymbol{\beta}))^2$ will get removed from the rule and thus, the *effective degree of freedom* (EoDF) will be far less than the total length of the rule. Effective degrees of freedom versus accuracy plot for GAM classifiers obtained using various combinations of (K_i, q_i) on Cancer-10 dataset is shown in Figure 3.29. It is clear that a high training accuracy is achieved with a large EoDF, but makes an over-fitting and produces less testing accuracy. About 500 such experiments are performed and the best combinations of (K_i, q_i) are used to generate results (Table 3.13) for a given dataset. Note here that generating classifiers using GAM is computationally expensive for high-dimensional datasets and so, we do not run experiments on datasets involving 500 features.

3.8.3 Genetic Programming (GP)

Genetic Programming has been extensively used to derive non-linear and interpretable classifiers [52, 53, 54, 55, 56, 57]. A GP algorithm evolves *programs* (or equations of classifier's decision boundary in our case) using genetic operators like crossover and mutation. Programs in GP are usually represented with tree architecture as shown in Figure 3.30. Internal nodes of this tree can involve mathematical operations, like $+, \times, -, \div, \log, \sin$. Allowable set of mathematical operations



(a) Training Accuracy (b) Testing Accuracy

Figure 3.29: Effective degree of freedom (EoDF) V/s Accuracy for Cancer-10 dataset. The best (K_i, q_i) parameter setting for this dataset is found to be $K^* = [8, 3, 8, 13, 8, 8, 13, 3, 8, 21]$ and $q^* = [2, 2, 5, 5, 2, 3, 3, 2, 2, 2]$.

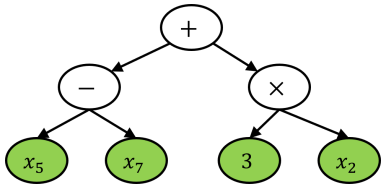


Figure 3.30: A sample genetic program (GP) tree. The above GP translates to this equation: $f(\mathbf{x}) = (x_5 - x_7) + 3x_2$.

are pre-specified by the user. In our case, we use $\{+, \times, -, \div\}$ only. Terminal leaf nodes of a GP program either have one of the input feature x_i or a constant term c. It is to note here that a GP tree ($\mathbf{T}$) represents one non-linear equation and is fundamentally different from the decision tree which involves assembly of split-rule equations which are organized in a hierarchical format. The optimal structure of tree, operators used, features x_i involved and value of constants c are all unknown and are determined through an evolutionary algorithm. The evolution is conducted with an objective to minimize the cross-entropy loss. However, if unchecked, the size of GP trees grows as the evolution progress and the GP algorithm suffers from *bloating* [58]. To counter this effect of bloating and encourage evolution of simpler trees (trees with less number of nodes), a parsimony coefficient P_c is used to penalize the fitness of a GP tree ($\mathbf{T}$) as shown below:

$$\textit{Min:} \quad f_{GP}(\mathbf{T}) = C_{loss} + P_c \times T_{size}, \tag{3.28}$$

where $C_{loss} = -\frac{1}{N}\sum_{i=1}^{N}[y(\mathbf{x}_i)\log(\hat{y}(\mathbf{x}_i)) + (1-y(\mathbf{x}_i))\log(1-\hat{y}(\mathbf{x}_i))]$, and $\hat{y}(\mathbf{x}) = \text{Sigmoid}(f(\mathbf{x}))$. In Eq. 3.28, T_{size} represents size of the tree and is computed by counting total number of nodes in the tree. $f(\mathbf{x})$ is the value the GP tree outputs for a given feature vector $\mathbf{x}$ (see Figure 3.30).

It is important to choose a suitable parsimony coefficient P_c for a problem. Smaller value of P_c will encourage bloating and will evolve complex equations while the higher value of P_c will evolve simpler equations at an expense of reduced classification accuracy. In our case, we perform experiments using three values P_c: 0.01, 0.005 and 0.001, and conduct 50 runs on each dataset shown in Table 3.12 after randomly splitting the dataset into 70% training and 30% testing for each run. Statistics regarding testing accuracy and *complexity* (measured as the total number of internal nodes) is reported in the table. It is clear from the table that while a small P_c produces a better

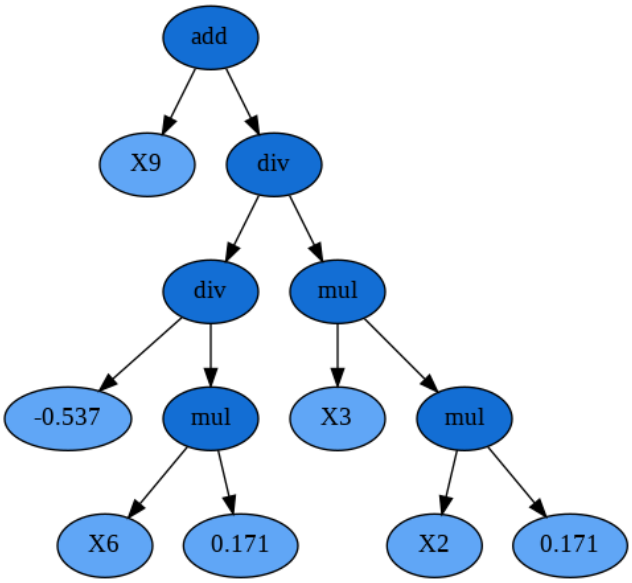
Table 3.12: GP Result for different values of parsimony coefficient P_c. For each dataset, the first row represents the testing accuracy and the second row represents complexity (number of internal nodes). $P_c = 0.001$ produces better results.

Pars. coeff.	**DS1**	**DS2**	**DS3**	**DS4**
$P_C = 0.01$	61.07 ± 9.91 **3.40 ± 3.70**	95.24 ± 0.00 **1.98 ± 0.14**	65.37 ± 11.57 **4.44 ± 2.37**	49.93 ± 1.43 **1.12 ± 3.40**
$P_C = 0.005$	77.3 ± 11.29 16.18 ± 9.99	95.24 ± 0.00 3.86 ± 1.23	86.27 ± 11.41 19.86 ± 11.45	50.37 ± 2.96 2.06 ± 3.88
$P_C = 0.001$	**91.70 ± 6.91** 67.72 ± 26.72	**95.37 ± 0.63** 15.14 ± 13.55	**96.50 ± 3.3** 76.74 ± 33.36	**58.00 ± 11.22** 18.76 ± 23.94

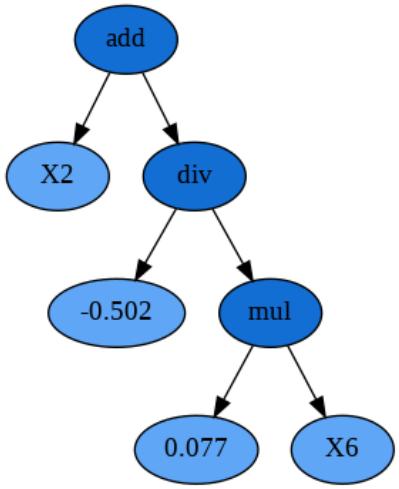
Pars. coeff.	**m-DS1**	**m-DS2**	**m-DS3**	**Cancer-10**
$P_C = 0.01$	89.53 ± 3.27 **8.34 ± 1.98**	95.65 ± 0.70 **3.58 ± 1.07**	96.33 ± 4.68 **15.04 ± 6.36**	94.03 ± 4.59 **5.56 ± 2.06**
$P_C = 0.005$	93.37 ± 4.57 16.32 ± 9.55	95.65 ± 0.70 3.76 ± 1.22	98.4 ± 1.99 19.88 ± 9.94	95.04 ± 1.76 7.88 ± 3.07
$P_C = 0.001$	**98.83 ± 1.88** 55.38 ± 22.39	**96.67 ± 1.93** 14.08 ± 9.11	**99.27 ± 1.22** 49.80 ± 21.69	**96.13 ± 1.29** 15.80 ± 5.66

Pars. coeff.	**Truss**	**WeldedBeam**	**Cancer-30**
$P_C = 0.01$	82.78 ± 11.28 **5.20 ± 3.30**	84.88 ± 13.08 **9.32 ± 5.15**	90.47 ± 4.54 **4.78 ± 2.30**
$P_C = 0.005$	90.03 ± 8.50 11.98 ± 7.12	92.35 ± 6.06 14.08 ± 5.35	90.96 ± 6.29 5.74 ± 1.84
$P_C = 0.001$	**97.36 ± 3.81** 36.02 ± 16.99	**96.46 ± 4.14** 35.90 ± 18.28	**92.40 ± 4.98** 14.58 ± 7.14

accuracy, a large P_c produces smaller sized GPs. To demonstrate, we present two GP classifiers for $P_c = 0.005$ and 0.01 obtained for the breast cancer Wisconsin dataset (involving total 10 features) in Figure 3.31. Training (T_r) and testing (T_e) accuracy are better for $P_c = 0.005$.



(a) $P_c = 0.005$, $T_r = 96.44$, $T_e = 99.02$, Complexity = 6.



(b) $P_c = 0.01$, $T_r = 95.60$, $T_e = 98.05$, Complexity = 3.

Figure 3.31: Classifiers for Cancer data: $P_c = 0.005$: $f(\mathbf{x}) = x_9 + \frac{-0.537}{(0.171x_6)(0.171x_3x_2)}$ and $P_c = 0.01$: $f(\mathbf{x}) = x_2 + \frac{-0.502}{(0.077x_6)}$.

Table 3.12 indicates that GP does not perform well on certain problems even in small-sized problems, such as DS1 and DS4. In a mathematical classifier search, there are two hierarchical aspects which must be learnt: (i) structure of the classifier, and (ii) coefficient of each term in the structure. GP attempts to learn both aspects in a single optimization task. We argue that while a "good" structure may have evolved at a generation, if its associated coefficients are not proper, the whole classifier will be judged as "bad". We attempt to alleviate this aspect in the next procedure by using a bilevel optimization framework.

3.8.4 Results

A comprehensive compilation of results obtained on all 19 datasets is shown in Table 3.13. The results are segmented into four parts: Sr. 1-7 for synthetic DS datasets, Sr. 8-9 for traditional cancer datasets, Sr. 10-11 for real-world multi-objective datasets, Sr. 12-19 for high-dimensional modified multi-objective ZDT and DTLZ benchmarks. For each method, a parametric study is performed on each problem and the setting which obtained the best testing accuracy is used to generate the final results. Statistics of 50 runs (with random data-split of 70% training and 30% testing in each) on each dataset for two performance metrics is presented in Table 3.13.

For CART, the complexity metric is defined as total number of nodes; for SVM, it is defined as total number of support vectors; for GAM, it is defined as the effective degrees of freedom (EoDF); for GP, it is defined as the total number of internal nodes; and for NLDT, it is defined as total number of variable occurrences in the entire tree. It is clear that a method with high testing accuracy and low complexity is better.

The table clearly indicates that NLDT performs well in terms of both metrics. Also, the performance of NLDT scales well with an increase in feature size. CART produces a good compromise on accuracy and complexity, but performs worse than NLDT on both metrics. While SVM achieves a high accuracy, in general, the complexity of its classifiers is large, thereby making them not easy to interpret for any explainability purposes. The performance of GP is poor for achieving a high accuracy. GAM is clearly not suitable for problems with a large number of features and cannot be run due to impractical computational time requirement for some problems (marked with a dash). GP cannot match both accuracy and complexity obtained by NLDT. GP's performance is also depended on the way the data is scattered which is evident from results on DS1 and m-DS1 problem. This is mostly because of the fact that in GP, the rule-structure and coefficients are evolved simultaneously, thereby making it difficult to efficiently navigate through the search-space of equations. A future study could be launched to incorporate bilevel search strategy with customized initialization to evolve rules within GP framework. In most problems, NLDT classifiers require fewer conditional rules (albeit with restricted nonlinearities) and still

achieve near 100% correct testing accuracy.

Table 3.13: Summary of results obtained using various methods. For each dataset, the first row indicates testing accuracy and the second row indicates complexity. Italicized entries are statistically insignificant (according to 95% confidence in Wilcoxon rank-sum test) compared to the best entry in the same row.

Sr.	Problem	NLDT	CART	SVM	GAM	GP
1	DS1	99.55 ± 1.08 **2.3 ± 0.6**	90.32 ± 4.06 14.5 ± 1.7	99.87 ± 0.45 8.16 ± 0.88	**100.0 ± 0.00** 2.89 ± 0.00	91.70 ± 6.91 67.72 ± 26.72
2	DS2	99.44 ± 0.87 **2.3 ± 0.7**	95.43 ± 1.50 11.0 ± 1.4	99.33 ± 1.10 7.64 ± 0.87	**100.0 ± 0.00** 2.89 ± 0.00	95.37 ± 0.63 15.14 ± 13.55
3	DS3	**99.77 ± 0.67** **2.2 ± 0.5**	95.00 ± 2.35 11.5 ± 1.3	*99.63 ± 0.69* 10.22 ± 1.42	*99.47 ± 1.03* 4.98 ± 0.14	96.50 ± 3.30 76.74 ± 33.36
4	DS4	**98.88 ± 1.65** **3.1 ± 1.4**	88.68 ± 3.60 31.3 ± 4.2	93.97 ± 2.35 43.70 ± 2.69	48.63 ± 6.50 3.80 ± 0.99	59.63 ± 10.81 24.70 ± 26.40
5	m-DS1	99.10 ± 1.54 **2.00 ± 0.00**	89.73 ± 4.53 7.90 ± 1.22	99.90 ± 0.40 7.50 ± 0.75	**100.0 ± 0.00** 2.90 ± 0.00	98.83 ± 1.88 55.38 ± 22.39
6	m-DS2	99.46 ± 1.08 **2.10 ± 0.30**	96.25 ± 1.92 5.96 ± 0.81	99.94 ± 0.44 5.44 ± 0.67	**99.94 ± 0.31** 2.90 ± 0.00	96.67 ± 1.93 14.08 ± 9.11
7	m-DS3	99.20 ± 1.30 **2.02 ± 0.14**	92.87 ± 4.35 5.78 ± 1.11	**100.00 ± 0.00** 8.82 ± 0.89	99.17 ± 1.48 3.24 ± 0.22	99.27 ± 1.22 49.8 ± 21.69
8	Cancer-10	**96.50 ± 1.16** **6.4 ± 1.7**	94.34 ± 1.92 11.6 ± 2.4	95.07 ± 1.23 51.26 ± 5.02	95.32 ± 1.49 22.14 ± 10.36	96.13 ± 1.29 15.80 ± 5.66
9	Cancer-30	**96.20 ± 1.49** **9.2 ± 4.1**	92.11 ± 2.07 10.8 ± 2.1	95.24 ± 1.29 58.88 ± 4.46	93.74 ± 5.83 32.47 ± 12.41	92.40 ± 4.98 14.58 ± 7.14
10	Welded Beam	98.58 ± 1.13 **3.9 ± 1.0**	97.72 ± 1.04 8.42 ± 1.42	**99.58 ± 0.45** 7.86 ± 1.27	*99.53 ± 0.48* 11.06 ± 0.81	96.46 ± 4.14 35.90 ± 18.28
11	Truss	**99.54 ± 0.75** **3.30 ± 0.90**	98.33 ± 1.10 11.06 ± 3.15	88.21 ± 1.62 174.28 ± 8.49	96.18 ± 1.20 19.19 ± 1.06	97.36 ± 3.81 36.02 ± 16.99
12	m-ZDT1-30	98.97 ± 0.57 **7.60 ± 3.50**	97.77 ± 0.58 30.26 ± 4.65	**99.39 ± 0.35** 82.08 ± 4.19	85.31 ± 1.35 220.20 ± 11.73	93.58 ± 10.21 45.34 ± 26.09
13	m-ZDT1-500	98.93 ± 0.60 **9.34 ± 4.15**	95.96 ± 0.80 21.02 ± 1.55	**100.00 ± 0.00** 140.58 ± 4.25	— —	83.21 ± 18.42 52.14 ± 24.47
14	m-ZDT2-30	98.96 ± 0.57 **8.10 ± 3.35**	97.88 ± 0.70 28.22 ± 2.35	**99.51 ± 0.33** 80.98 ± 3.50	84.97 ± 1.18 233.69 ± 6.56	91.57 ± 11.92 48.44 ± 23.69
15	m-ZDT2-500	98.87 ± 0.72 **8.84 ± 3.95**	95.96 ± 0.80 21.02 ± 1.55	**100.00 ± 0.00** 140.56 ± 4.00	— —	85.06 ± 16.82 50.64 ± 21.24
16	m-DTLZ1-30	**98.77 ± 0.87** **11.98 ± 5.85**	78.52 ± 7.94 128.40 ± 22.39	94.22 ± 0.95 615.54 ± 9.24	55.96 ± 3.19 33.89 ± 0.01	81.59 ± 17.32 16.08 ± 21.89
17	m-DTLZ1-500	**93.76 ± 4.24** 22.72 ± 7.50	78.31 ± 7.21 126.94 ± 20.07	64.32 ± 1.76 1236.82 ± 13.26	— —	80.49 ± 11.54 **8.66 ± 16.19**
18	m-DTLZ2-30	**97.22 ± 2.25** 17.48 ± 6.75	69.83 ± 6.16 156.00 ± 16.09	94.25 ± 1.04 615.44 ± 10.67	54.52 ± 2.96 35.84 ± 0.02	79.81 ± 19.46 **12.32 ± 14.79**
19	m-DTLZ2-500	**95.32 ± 4.45** 22.02 ± 10.62	76.68 ± 5.44 133.22 ± 15.95	64.22 ± 1.48 1245.92 ± 11.45	— —	78.46 ± 14.08 **8.08 ± 15.21**

3.9 Conclusions and Future work

In this chapter, we have addressed the problem of generating *interpretable* and accurate non-linear decision trees for binary classification problems. Split-rules at the conditional nodes of a decision tree have been represented as a weighted sum of power-laws of feature variables. A provision of integrating *modulus* operation in the split-rule has also been formulated, particularly

to handle sandwiched classes of datapoints. The proposed algorithm of deriving split-rule at a given conditional-node in a decision tree has two levels of hierarchy, wherein the *upper-level* is focused at evolving the power-law structures and operates on a discrete variable space, while the *lower-level* is focused at deriving values of optimal weights and biases by searching a continuous variable space to minimize the net impurity of child nodes after the split. Efficacy of upper-level and lower-level evolutionary algorithm have been tested on customized datasets. The lower-level GA is able to robustly and efficiently determine weights and biases to minimize the net-impurity. A test conducted on imbalanced dataset has also revealed the superiority of the proposed lower-level GA to achieve a high classification accuracy without relying on any synthetic data. Ablation studies conducted on the upper-level GA also demonstrate the efficacy of the algorithm to obtain simpler split-rules without using any prior knowledge. Results obtained on standard classification benchmark problems and a real-world single-objective problem has amply demonstrated the efficacy and usability of the proposed algorithm.

The classification problem has been extended to discriminate Pareto-data from non-Pareto data in a multi-objective problem. Experiments conducted on multi-objective engineering problems have resulted in determining simple polynomial rules which can assist in determining if the given solution is Pareto-optimal or not. These rules can then be used as design-principles for generating future optimal designs. As further future studies, we plan to integrate non-polynomial and generic terms in the expression of the split-rules. The proposed bilevel framework can also be tested for *regression* and *reinforcement-learning* related tasks which we shall discuss in upcoming chapters. The upper-level problem can be converted to a bi-objective problem for optimizing both F_U and F_L simultaneously, so that a set of trade-off solutions can be obtained in a single run. Nevertheless, this proof-of-principle study on the use of a customized bilevel optimization method for classification tasks is encouraging for us to launch such future studies.

3.10 Parameter Settings

3.10.1 Termination Criteria and other Parameter Settings for Inducing a Non-linear Decision Tree (NLDT)

- Number of power-laws per split-rule: $p = 3$,
- Allowable set of exponents: $\mathbf{E} = \{-3, -2, \ldots, 3\}$,
- Impurity Metric: Gini Score.
- Minimum Impurity to conduct a split: $\tau_{min} = 0.05$
- Minimum number of points required in a node conduct a split: $N_{min} = 10$
- Maximum allowable Depth = 5
- Pruning threshold: $\tau_{prune} = 3\%$

3.10.2 Parameter Setting for NSGA-II for multi-objective data creation

We used the implementation of NSGA-II Algorithm as provided in *pymoo* [59]. *pymoo* is an official python package for multiobjective optimization algorithms and is developed under supervision of Prof. Deb who is the original developer of NSGA-II and NSGA-III algorithms. Following parameter setting was adopted to create Pareto and non-Pareto datasets for two objective Truss and Welded beam problems:

- Population Size = 500
- Maximum Generations = 1000
- Cross Over = SBX
- SBX $\eta_c = 15$
- SBX probabililty = 0.9

- Mutation type = Polynomial Mutation
- Mutation η_m = 20
- Mutation probability $p_m = 1/n_{var}$ (where n_{var} is total number of variables)

3.10.3 Parameter Setting for Upper Level GA

This section provides the general parameter setting used for the upper level of our bilevel GA.

- Population size = $10 \times d$, where d is the dimension of the feature space of the original problem.
- Selection = Binary tournament selection.
- Crossover probability (p^U_{xover}) = 0.9
- Mutation parameters:
  - Mutation probability (p^U_{mut}) = $min(0.33, 1/d)$.
  - Distribution parameter β = 3 (any value > 1 will be good).
- p_{zero} = 0.75
- Maximum number of generations = 100

3.10.4 Parameter Setting for Lower Level GA

Following parameter setting is used for lower level GA:

- Population size = 50,
- Maximum generations = 50,
- Variable range = $[-1, 1]$,
- Crossover type = Simulated Binary Crossover (SBX)

- Mutation type = Polynomial Mutation
- Selection type = Binary Tournament Selection
- Crossover probability = 0.9,
- Mutation probability = $1/n_{var}$,
- Number of variables $n_{var} = p + 1$, if $m = 0$ or $p + 2$ if $m = 1$, where p is the total number of power-laws allowed and m is the modulus flag,
- SBX and polynomial mutation parameters: $(\eta_c, \eta_m) = (2, 15)$.

3.10.5 Creation of Customized 2D Datasets: DS1- DS4

As mentioned before, the customized datasets DS1-DS4 (Figure 3.8) were synthetically generated. This section explains the procedure which was adopted to generate these customized datasets. Datapoints in $2D$ space were created using a reference curve $\xi(\mathbf{x})$ and were assigned to either the *Class-1* or *Class-2*. Two additional parameters δ and σ controlled the location of a datapoint relative to the reference curve $\xi(\mathbf{x})$. Datapoints ($\mathbf{x^{(i)}}$) for a given dataset were generated using following steps:

- First n points falling on curve $\xi(\mathbf{x}) = 0$ were initialized for reference. Lets denote these reference points on the curve with $\mathbf{x_r^{(i)}}$ (where $i = 1, 2, \ldots, n$). Thus,

$$\xi(\mathbf{x_r^{(i)}}) = 0, \qquad i = 1, 2, \ldots, n.$$

- n points ($\mathbf{x^{(i)}}$) for a dataset were then generated using the following equation

$$\mathbf{x^{(i)}} = \mathbf{x_r^{(i)}} + \delta + \sigma r, \qquad i = 1, 2, \ldots, n, \tag{3.29}$$

  where r is a random number between 0 and 1, δ represents the *offset* and σ indicates the amount of *spread*.

Four datasets generated for conducting ablation studies are graphically represented in Fig. 3.8.

Parameter setting which was used to generate these datasets is provided Table 3.14. N_A and N_B indicate the number of datapoints belonging to *Class-1* and *Class-2* respectively.

Table 3.14: Parameter setting to create customized datasets *D1-D4*. For Class-1 data-points $(\delta, \sigma) = (0, 0.01)$.

Dataset	$\xi(\mathbf{x})$	**Class-1**	**Class-2**	
		$\mathbf{N_A}$	$\mathbf{N_B}$	(δ, σ)
DS1	$2x_1 + x_2 - 3 = 0$	100	100	$(0.025, 0.2)$
DS2	$2x_1 + x_2 - 3 = 0$	10	200	$(0.025, 0.2)$
DS3	$x_1^2 + x_2 - 1 = 0$	100	100	$(0.025, 0.2)$
DS4	$2x_1 + x_2 - 3 = 0$	100	50	$(0.025, 0.2)$
			50	$(-0.015, -0.2)$

CHAPTER 4

CONTROL: INTERPRETABLE POLICY FOR DISCRETE ACTION SPACES

4.1 Introduction

Control system problems are increasingly being solved by using modern reinforcement learning (RL) and other machine learning (ML) methods to find an autonomous agent (or controller) to provide an optimal action A_t for every state variable combination S_t in a given environment at every time-step t. Execution of the output *action* A_t takes the object to the next *state* S_{t+1} in the environment and the process is repeated until a termination criteria is met. This is conceptually demonstrated in Figure 4.1

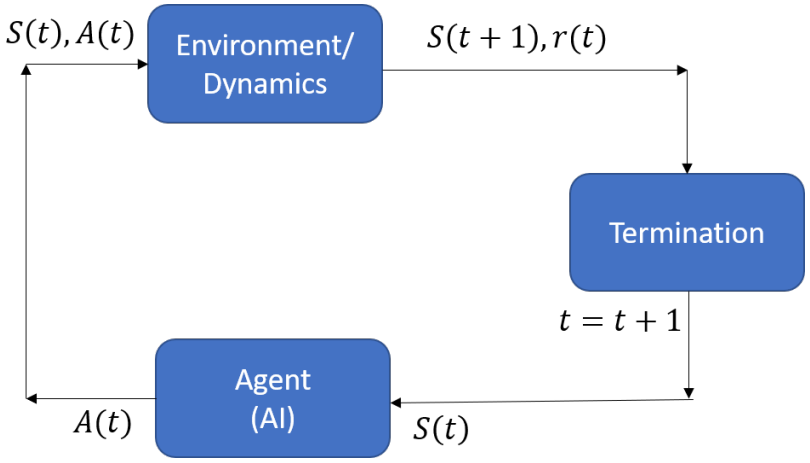


Figure 4.1: Control Loop

The mapping between input *state* S_t and output *action* A_t is usually captured through an artificial intelligence (AI) method. In the RL literature, this mapping is referred to as *policy* ($\pi(S) : \mathbb{S} \rightarrow \mathbb{A}$), where $\mathbb{S}$ is the *state space* and $\mathbb{A}$ is the *action space*. Sufficient literature exists in efficient training of these RL *policies* [60, 61, 62, 63]. While these methods are efficient at training the AI policies for a given control system task, the developed AI policies, captured through complicated networks, are complex and non-interpretable.

Interpretability of AI policies is important to a human mind due to several reasons: (i) they help

provide a better insight and knowledge to the working principles of the derived policies, (ii) they can be easily deployed with a low fidelity hardware, (iii) they may also allow an easier way to extend the control policies for more complex versions of the problem. While defining interpretability is a subjective matter, a number of past efforts have attempted to find interpretable AI policies with limited success.

In the remainder of this chapter, we first present the main motivation behind finding interpretable policies in Section 4.2. A few past studies in arriving at interpretable AI policies is presented in Section 4.3. In Section 4.5, we briefly discuss an extension to our nonlinear decision tree (NLDT) approach in the context of arriving at interpretable AI policies. The overall open-loop and closed-loop NLDT policy generation methods are described in Section 4.6. Results on some control system problems are presented in Section 4.7. Finally, conclusions and future studies are presented in Section 4.10.

4.2 Motivation for the Study

As mentioned in the earlier chapters, various data analysis tasks, such as classification, controller design, regression, image processing, etc., are increasingly being solved using artificial intelligence (AI) methods.These are done, not because they are new and interesting, but because they have been demonstrated to solve complex data analysis tasks without much change in their usual frameworks. With more such studies over the past few decades, they are faced with a huge challenge. Achieving a high-accuracy solution does not necessarily satisfy a curious domain expert, particularly if the solution is not interpretable or explainable. A technique (whether AI-based or otherwise) to handle data well is no more enough, researchers now demand an explanation of why and how they work.

Consider the MountainCar control system problem (see Figure 4.2, which has been extensively studied using various AI methods [64, 65, 66]. The problem has two state variables (position x_t along x-axis and velocity v_t along positive x-axis) at every time instant t which would describe the state of the car at t. Based on the state vector $S_t = (x_t, v_t)$, a policy $\pi(S)$ must decide on one of the three actions A_t: decelerate ($A_t = 0$) along positive x-axis with a pre-defined value $-a$, do nothing

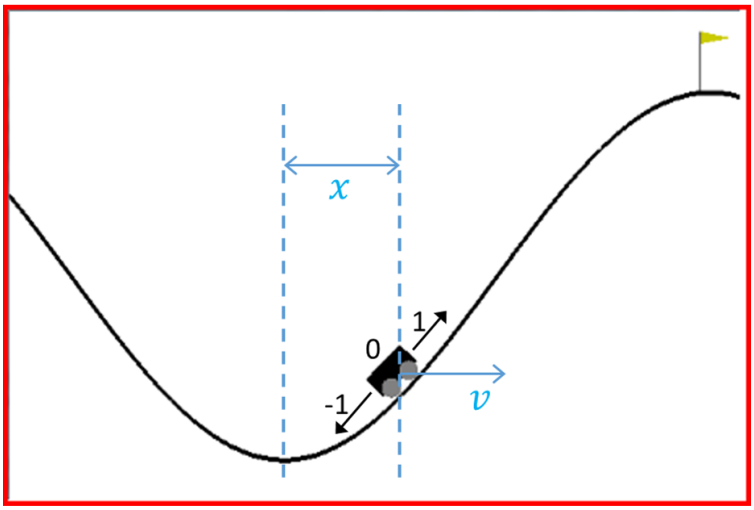
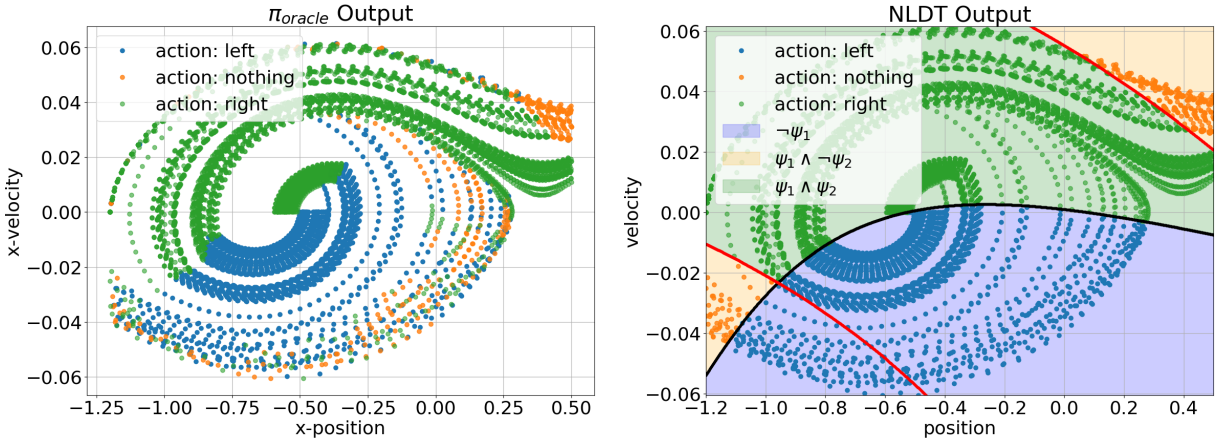


Figure 4.2: Mountain Car problem. It comprises of two state variables x, v and is controlled using three actions: -1 for deceleration, 0 for nothing and +1 for acceleration.

($A_t = 1$), or accelerate ($A_t = 2$) with a in positive x-axis direction. The goal of the control policy $\pi(S)$ is to take the under-powered car (it does not have enough fuel to directly climb the mountain and reach the destination) over the right hump in a maximum of 200 time-steps starting anywhere at the trough of the landscape. Physical laws of motion are applied and a policy $\pi(S)$ has been trained to solve the problem. The RL produces a black-box policy $\pi_{oracle}(S)$ for which an action $A_t \in [0, 1, 2]$ will be produced for a given input $S_t = (x_t, v_t) \in \mathbb{R}^2$. Figure 4.3a shows the state-



(a) Using π_{oracle}.

(b) Using NLDT.

Figure 4.3: State-action combinations for MountainCar prob.

action combinations obtained from 92 independent successful trajectories (amounting to total of

10,000 time-steps) leading to achieving the goal using a pre-trained deterministic black-box policy π_{oracle}. The x-location of the car and its velocity can be obtained from a *point* on the 2D plot. The color of the point $S_t = (x_t, v_t)$ indicates the action A_t suggested by the oracle policy π_{oracle} ($A_t = 0$: blue, $A_t = 1$: orange, and $A_t = 2$: green). If a user is now interested in understanding how the policy π_{oracle} chooses a correct A_t for a given S_t, one way to achieve this would be to *represent* and analyze the policy function $\pi_{int}(S_t)$ as shown below:

$$\pi_{int}(S_t) = \begin{cases} action\ 0, & \text{if } \phi_0(S_t) \text{ is } true, \\ action\ 1, & \text{if } \phi_1(S_t) \text{ is } true, \\ action\ 2, & \text{if } \phi_2(S_t) \text{ is } true, \end{cases} \tag{4.1}$$

where $\phi_i(S_t) : R^2 \rightarrow \{0, 1\}$ is a Boolean function which partitions the state space $\mathbb{S}$ into two sub-domains based on its output value and for a given state S_t, exactly one of $\phi_i(S_t)$ is *true*, thereby making the policy π_{int} deterministic. If we re-look at Figure 4.3a we notice that the three actions are quite mixed at the bottom part of the x-v plot (state space). Thus, the partitioning Boolean functions ϕ_i of Eq. 4.1 need to be quite complex in order to have $\phi_0(S_t) = true$ for all *blue* points, $\phi_1(S_t) = true$ for all *orange* points and $\phi_2(S_t) = true$ for all *green* points in a mutually exclusive manner.

What we address in this study is an attempt to find an *approximated* policy function $\pi_{int}(S_t)$ which may not explain all 100% time instance data corresponding to the oracle black-box policy $\pi_{oracle}(S_t)$ (Figure 4.3a), but it is fairly interpretable to capture and explain most of the behavior of π_{oracle}. Consider the state-action plot in Figure 4.3b, which is generated with a simpler and a relatively more interpretable policy $\pi_{int}(S_t) = \{i | \phi_i(S_t) \text{ is } true, i = 0, 1, 2\}$ obtained by our proposed procedure as shown below

$$\begin{aligned} \phi_0(S_t) &= \neg\psi_1(S_t), \\ \phi_1(S_t) &= (\psi_1(S_t) \wedge \neg\psi_2(S_t)), \\ \phi_2(S_t) &= (\psi_1(S_t) \wedge \psi_2(S_t)), \end{aligned} \tag{4.2}$$

where $\psi_1(S_t) = |0.96 - 0.63/\widehat{x_t}^2 + 0.28/\widehat{v_t} - 0.22\widehat{x_t}\widehat{v_t}| \leq 0.36$, and $\psi_2(S_t) = |1.39 - 0.28\widehat{x_t}^2 - 0.30\widehat{v_t}^2| \leq 0.53$. Here, $\widehat{x_t}$ and $\widehat{v_t}$ are normalized state variables (see Section 4.6.1). The action A_t predicted using the above policy does not match the output of π_{oracle} at some states (about 8.1%), but from our experiments we observe that it is still able to drive the mountain-car to the destination goal located on the right hill in 99.8% episodes.

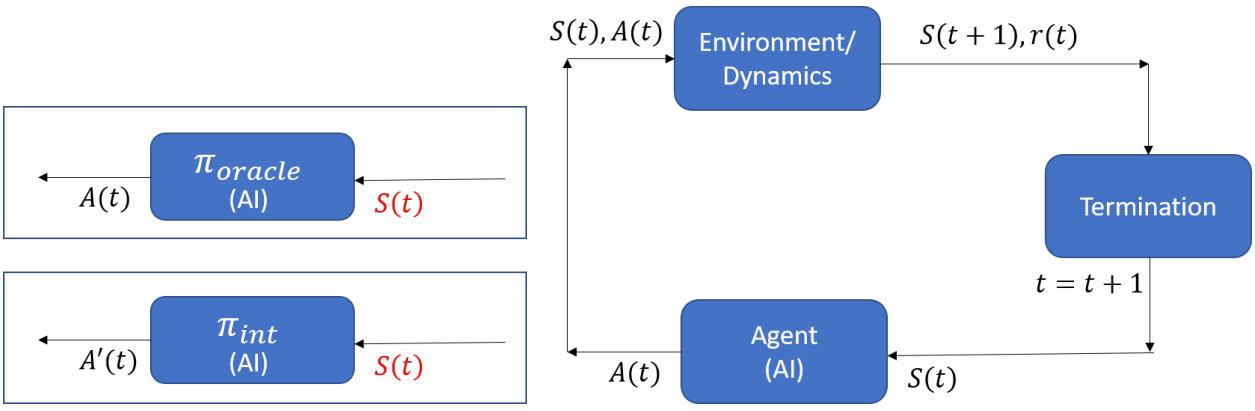
Importantly, the policies are simplistic and amenable to an easier understanding of the relationships between x_t and v_t to make a near perfect control. Since the explanation process used the data from π_{oracle} as the universal truth, the derived relationships can also provide an explanation of the working of the black-box policy π_{oracle}. A more gross approximation to Figure 4.3a by more simplified relationships (ϕ_i) may reduce the overall *open-loop* accuracy (see Section 4.4) of matching the output of π_{oracle}. Hence, a balance between a good interpretability and a high *open-loop* accuracy in searching for Boolean functions $\phi_i(S_t)$ becomes an important matter for such an interpretable AI-policy development study.

In this work, we focus on developing a search procedure for arriving at the ψ-functions (see Eq. 4.2) for discrete action systems. The structure of the policy $\pi_{int}(S_t)$ shown in Eq. 4.1 resembles a decision tree (DT), but unlike a standard DT, it involves a nonlinear function at every non-leaf node, requiring an efficient nonlinear optimization method to arrive at reasonably succinct and accurate functionals. The procedure we propose here is generic and is independent of the AI method used to develop the black-box policy π_{oracle}.

4.3 Related Past Studies

A few studies exist which are focused at generating interpretable policies. In [67], an interpretable orchestrator is developed to choose from two RL-policies: π_C (modelled for reward maximization) and π_R for maximizing an *ethical* consideration. The orchestrator is dependent on only one of the state-variables and despite it being interpretable, the policies: π_C and π_R are still black-box and convoluted. [68] constructs a set of interpretable index based policies and uses multi-arm bandit procedure to select a high performing index based policy. The search space of

interepretable policies is much smaller and the procedure suggested for finding an interpretable policy is computationally heavy, taking about hours to several days of computational time on simple control problems. In [69], genetic programming (GP) is used to obtain interpretable policies on control tasks involving continuous actions space through model-based policy learning. However the *interpretability* was not captured in the design of the fitness function and a large archive was created passively to store every policy for each complexity encountered during the evolutionary search. A linear decision tree (DT) based model is used in [70] to approximate the Q-values of trained neural network. In that work, the split in DT occurs based on only one feature, and at each terminal node the Q-function is fitted using a linear model on all features. [71] uses a *program sketch* S to define the domain of interpretable policies e. Interpretable policies are found using a trained black-box oracle e_N as a reference by first conducting a local search in the sketch space S to mimic the behaviour of the oracle e_N and then fine-tuning the policy parameters through online Bayesian optimization. The bias towards generating interpretable programs is done through controlled initialization and local search rather than explicitly capturing *interpretability* as one of the fitness measure. Particle swarm optimization [72] is used to generate interpretable fuzzy rule set in [73] and is demonstrated on classic control problems involving continuous actions. Works on DT [12] based policies through imitation learning has been carried out in [74]. [75] extends this to utilize Q-values and eventually render DT policies involving $< 1,000$ nodes on some toy games and CartPole environment with an ultimate aim to have the induced policies verifiable. [76] used axis-aligned DTs to develop interpretable models for black-box classifiers and RL-policies. They first derive a distribution function $\mathcal{P}$ by fitting the training data through axis-aligned Gaussian distributions. $\mathcal{P}$ is then used to compute the loss function for splitting the data in the DT. [77] attempts to generate interpretable DTs from an ensemble using a genetic algorithm. In [78], regression trees are derived using classical methods such as CART [12] and Kd-tree [79] to model Q-function through supervised training on batch of experiences and comparative study is made with ensemble techniques. In [80], a gradient based approach is developed to train the DT of pre-fixed topology involving linear split-rules. These rules are later simplified to allow only one feature per split node



(a) Open-loop (b) Closed-loop

Figure 4.4: Performance measures.

and resulting DTs are pruned to generate simplified rule-set.

While the above methods attempt to generate an interpretable policy, the search process does not use *complexity* of policy in the objective function, instead, they rely on initializing the search with certain interpretable policies. In our approach described below, we use to concept of NLDT induction discussed in Chapter 3 to build an efficient search algorithm to directly find relatively simple and interpretable policies than black-box DNN or tabular-tile based policies using recent advances in nonlinear optimization.

4.4 Performance Measures

Before we proceed into formally discussing the NLDT based policies, we will quickly discuss two performance metrics we used to measure and compare performance of policies.

4.4.1 Open-Loop Accuracy

Open-loop accuracy quantifies how accurately does a given policy π_{int} mimics the reference black-box oracle policy π_{oracle}. This is conceptually shown in Figure 4.4a. Here, under ideal scenario (100% open-loop accuracy), for every state $S(t)$, the output $A'(t)$ of policy π_{int} will match the output $A(t)$ of the black-box policy π_{oracle}. In our case, since the action-space is discrete, the open-loop accuracy is identical to the classification accuracy on the labelled state-action dataset

generated using the black-box policy π_{oracle}.

4.4.2 Closed-loop Performance

Under the close-loop setting, the AI is directly used as a controller to control the object by interacting with the environment or dynamics of the control system as shown in Figure 4.4b. For each transition from state $S(t)$ to state $S(t+1)$ using action $A(t)$, a reward $r(t)$ is collected. The loop is repeated until a termination criteria is reached. The entire sequence of state-action-state transitions from start to termination is referred to as *episode*. The cumulative reward value collected during entire episode is given by

$$R_e = \sum_{t=0}^{t_f} r(t), \tag{4.3}$$

where t_f is the time-step at which an episode terminated. The episode is called a *success* if the desired goal is reached or a target is achieved upon the termination. Else, the episode is declared as a *failure*. In the mountain-car example (Figure 4.2), an episode is considered as a success if by 200 time-steps, the AI is able to drive the car to the flag-post located on the right up-hill, else it's a failure.

We measure closed-loop performance of an AI using two metrics

- **Cumulative Reward** which is measured using Eq. 4.3, and
- **Task Completion Rate**, which quantifies number of successful episodes across 100 closed-loop simulations.

4.5 Nonlinear Decision Trees (NLDTs) as Policies

Similar to classification problems where NLDT is used as a classifier to predict the class of a given datapoint, in control systems involving discrete actions, NLDT can represent a policy wherein for a given input state $\mathbf{x}$, the corresponding action a can be determined by traversing the tree and reaching the leaf node.

We implement two frameworks to represent non-linear decision trees: binary-split and multi-split. The binary-split NLDT is identical to the one discussed in Chapter 3. The multi-split NLDT can be obtained by allowing more than two splits for a given conditional node. We will briefly discuss binary-split and multi-split NLDT frameworks.

4.5.1 Binary-split NLDT

In binary-split NLDT, a conditional node undergoes exactly two splits. Unlike in Chapter 3 where we mainly discussed problems involving two classes, in case of discrete-action control problems, more than two actions can appear. This is conceptually illustrated in Figure 4.5 for control problem involving three actions.

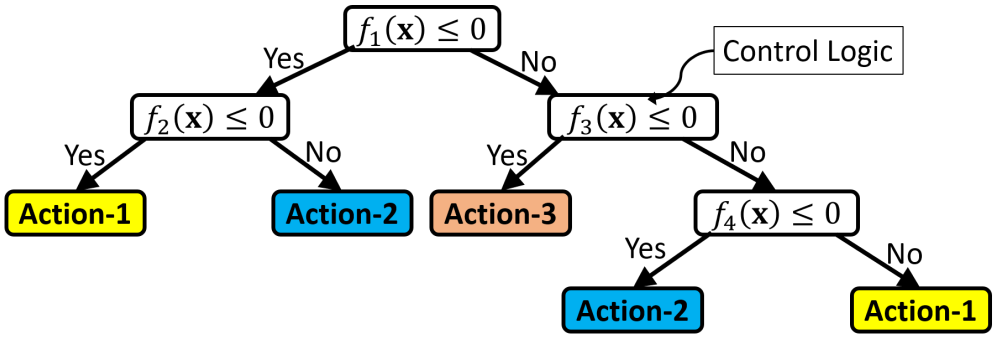


Figure 4.5: Binary-split NLDT for Discrete action control systems.

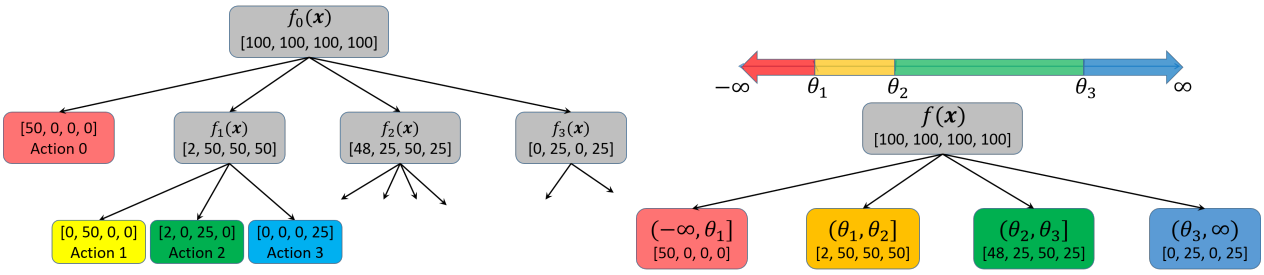
Each conditional node represents a non-linear control logic $f(\mathbf{x}) \leq 0$, where the non-linear function $f(\mathbf{x})$ assumes the similar form as the one discussed in Chapter 3 and is shown below for quick reference

$$f(\mathbf{x}) = \begin{cases} \sum_{i=1}^{p} w_i B_i + \theta_1, & \text{if } m = 0, \\ \left| \sum_{i=1}^{p} w_i B_i + \theta_1 \right| - |\theta_2|, & \text{if } m = 1, \end{cases} \tag{4.4}$$

The B_i are power-laws of type $B_i = \prod_{j=1}^{d} x_j^{b_{ij}}$ and m indicates *presence* or *absence* of absolute operator.

4.5.2 Multi-split NLDT

Unlike the binary-split NLDT, in multi-split NLDT, a node is allowed to have more than two partitions or splits. If the number of classes (or discrete action values in our case) in the dataset is c, then a node in the multi-split NLDT can have upto c-splits as shown in Figure 4.6a.



(a) Schematic of Multi-split NLDT.

(b) Biases in $f_i(\mathbf{x})$ for Multi-split NLDT.

Figure 4.6: Multi-split NLDT configuration. Numbers in square brackets indicate class distribution of datapoints.

The number within square brackets in a node in multi-split NLDT of Figure 4.6a indicates class distribution. The split function $f(\mathbf{x})$ for a node in multi-split NLDT is given by the following equation

$$f(\mathbf{x}) = \sum_{i=1}^{p} w_i B_i. \tag{4.5}$$

Notice the difference between split-rule for binary-split NLDT (Eq. 4.4) and split-rule for multi-split NLDT (Eq. 4.5). The former involves bias terms θ_i and a modulus parameter m while the latter is expressed as the weighted sum of power-laws B_i (Eq. 3.3) only. In multi-split NLDT, for a given feature vector $\mathbf{x}$, the value of its split function $f(\mathbf{x})$ is compared against biases θ_i to determine child node where the datapoint will belong. This is schematically shown in Figure 4.6b for a node undergoing 4 splits. Thus, for a given feature vector $\mathbf{x}$,

$$\begin{aligned} &\text{If } -\infty < f(\mathbf{x}) \leq \theta_1, && \text{Move to Split 1 Node,} \\ &\text{If } \theta_1 < f(\mathbf{x}) \leq \theta_2, && \text{Move to Split 2 Node,} \\ &\ldots && \ldots \end{aligned}$$

4.6 Overall Approach

The overall approach is illustrated in Figure 4.7. First, a dedicated black-box policy π_{oracle} is

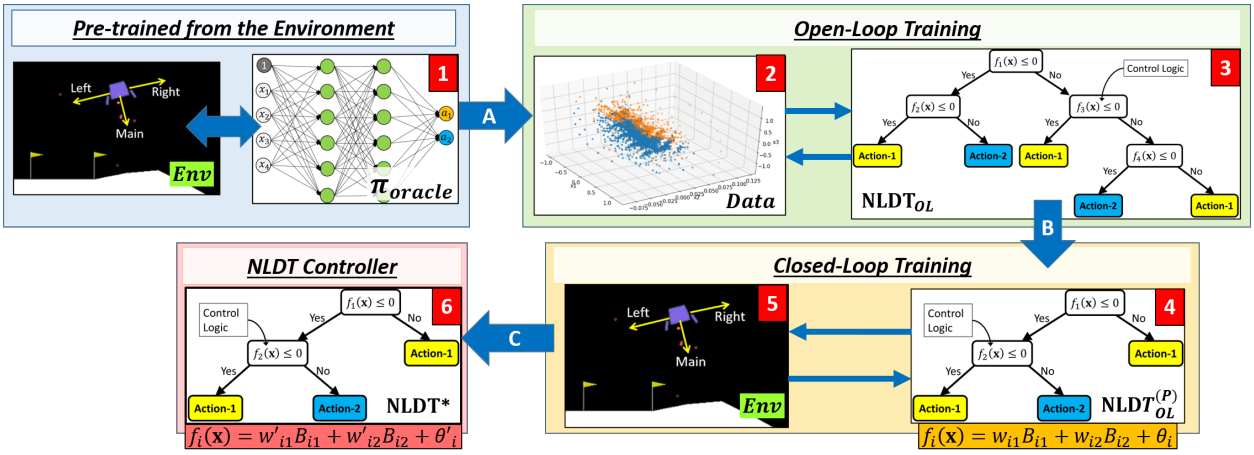


Figure 4.7: A schematic of the proposed overall approach.

trained from the actual environment/physics of the problem. Training of black-box policy π_{oracle} is not the focus of our current work and we use standard approaches from literature to obtain π_{oracle}. Next, the trained policy π_{oracle} (Block 1 in the figure) is used to generate labelled training and testing datasets of *state-action* pairs from different time-steps. We generate two types of training datasets: *Regular* – as they are recorded from multiple episodes, and *Balanced – selected* from multiple episodes to have almost equal number of *states* for each action, where an *episode* is a complete simulation of controlling an object with a policy over multiple time-steps. Third, the labelled training dataset (Block 2) is used to find the NLDT (Block 3) using the recursive bilevel evolutionary algorithm described in Section 4.6.2. We call this an open-loop NLDT (or, NLDT$_{OL}$), since it is derived from a labelled state-action dataset generated from π_{oracle}, without using any overall reward or any final goal objective in its search process, which is typically a case while doing reinforcement learning. Use of labelled state-action data in supervised manner allows a faster search of NLDT even with a large dataset as compared to constructing the NLDT from scratch through reinforcement learning by interacting with the environment to maximize the cumulative rewards [71]. Next, in an effort to make the overall NLDT relatively more interpretable while simultaneously ensuring better closed-loop performance, we prune the NLDT by taking only the

top part of NLDT_{OL} (we call $\text{NLDT}_{OL}^{(P)}$ in Block 4) and *re-optimize* all non-linear rules within it for the weights and biases using an efficient evolutionary optimization procedure to obtain final NLDT* (Block 5). The re-optimization is done here with closed-loop objectives, such as the cumulative reward function or closed-loop completion rate. We briefly discuss the open-loop training procedure of inducing NLDT_{OL} and the closed-loop training procedure to generate NLDT* in next sections.

4.6.1 Data Normalization

First, we provide the exact normalization of state variables performed before the open-loop learning task is executed. Before training and inducing the non-linear decision tree (NLDT), features in the dataset are normalized using the following equation:

$$\widehat{x_i} = 1 + (x_i - x_i^{\min})/(x_i^{\max} - x_i^{\min}), \tag{4.6}$$

where x_i is the original value of the i-th feature, $\widehat{x_i}$ is the normalized value of the i-th feature, $x_i^{\min}$ and $x_i^{\max}$ are minimum and maximum value of i-th feature as observed in the training dataset. This normalization will make every feature x_i to lie within $[1, 2]$. This is done to ensure that $x_i = 0$ is avoided to not cause a division by zero.

4.6.2 Open-loop Training

As a first step for open-loop training, the pre-trained black-box oracle π_{oracle} is used to generate labelled *state-action* dataset (step *A* in Figure 4.7). Once the labeled dataset is generated, the problem of inducing NLDT to fit the data translates to the supervised learning problem of developing a classifier.

4.6.2.1 Open-loop training for Binary-split NLDT

The open-loop training for binary-split is identical to the procedure discussed in Chapter 3 wherein an evolutionary bilevel algorithm is employed to derive non-linear split-rule at each conditional

node. The NLDT is grown using recursive splitting of training data.

4.6.2.2 Open-loop training for Multi-split NLDT

Similar to binary-split NLDT, a dedicated bilevel optimization algorithm is applied to derive the split-rule $f(\mathbf{x})$ of Eq. 4.5 and its associated biases θ_i at a given conditional node. The optimization formulation to obtain the split-rule for a conditional node in multi-split NLDT and values of corresponding θ_i is as given below

$$\begin{aligned}
\text{Min.} \quad & F_U(\mathbf{B}, \mathbf{w}^*, \mathbf{\Theta}^*), \\
\text{s.t.} \quad & (\mathbf{w}^*, \mathbf{\Theta}^*) \in \text{argmin} \left\{ F_L(\mathbf{w}, \mathbf{\Theta})|_{(\mathbf{B},m)} \right| \\
& \quad F_L(\mathbf{w},\ -1 \leq w_i \leq 1,\ \forall i,\ \mathbf{\Theta})|_{(\mathbf{B},m)} \leq \tau_I, \\
& \quad \mathbf{\Theta} \in [-1, 1]^{m+1} \Big\}, \\
& \theta_i < \theta_j,\ i < j,\ \forall i, j,\ b_{ij} \in Z.
\end{aligned} \tag{4.7}$$

Here, the upper level objective function F_U is identical to the one given in Eq. 3.7. It quantifies the complexity of the rule by counting the number of non-zero exponents in power-laws B_i. The lower level objective function F_L quantifies the quality of split by computing weighted sum of impurity scores of resulting child nodes as shown in the equation below

$$F_L = \sum_{i=1}^{n_{childs}} \frac{N_i}{N} \text{Gini}(i), \tag{4.8}$$

where n_{childs} indicates total number of child nodes created from the split. The number of splits (and hence the number of child nodes) depends on the distribution of datapoints in the given conditional node. If the number of datapoints belonging to i-th class in the given conditional node is n_i and there are total c classes in the original dataset, then the number of splits n_{splits} (or equivalently n_{childs}) a given conditional node undergoes is

$$n_{splits} = \sum_{i=1}^{c} \frac{\max(0, n_i - \tau_c)}{n_i - \tau_c}, \tag{4.9}$$

where τ_c is a user specified parameter. The optimization formulation for lower-level optimization in multi-split NLDT is given as under

$$\begin{aligned} \text{Min.} \quad & F_L(\mathbf{w}, \mathbf{\Theta})\big|_{(\mathbf{B},m)}, \\ \text{s.t.} \quad & \theta_i < \theta_j,\ i \in \{1, \ldots, n_c - 2\} \text{ and } j = i + 1, \\ & w_i \in [-1, 1]^p,\ \mathbf{\Theta} \in [-1, 1]^{n_c - 1}, \end{aligned} \tag{4.10}$$

where p is the total number of power-laws B_i (Eq. 4.5).

4.6.3 Closed-loop Training

The intention behind the closed-loop training is to enhance the closed-loop performance of NLDT. It will be discussed in Section 4.7 that while closed-loop performance of NLDT_{OL} is at par with π_{oracle} on control tasks involving two to three discrete actions, like CartPole and MountainCar, the NLDT_{OL} struggles to autonomously control the agent for control problems such as LunarLander having more states and actions. In closed-loop training, we fine-tune and re-optimize the weights $\mathbf{W}$ and biases $\mathbf{\Theta}$ of an entire NLDT_{OL} (or pruned NLDT_{OL}, i.e. $\text{NLDT}_{OL}^{(P)}$ – block 4 in Figure 4.7) to maximize its closed-loop fitness (F_{CL}), which is expressed as the average of the cumulative reward collected on M episodes:

$$\begin{aligned} \text{Maximize} \quad & F_{CL}(\mathbf{W}, \mathbf{\Theta}) = \frac{1}{M} \sum_{i=1}^{M} R_e(\mathbf{W}, \mathbf{\Theta}), \\ \text{Subject to} \quad & \mathbf{W} \in [-1, 1]^{n_w}, \mathbf{\Theta} \in [-1, 1]^{n_\theta}, \end{aligned} \tag{4.11}$$

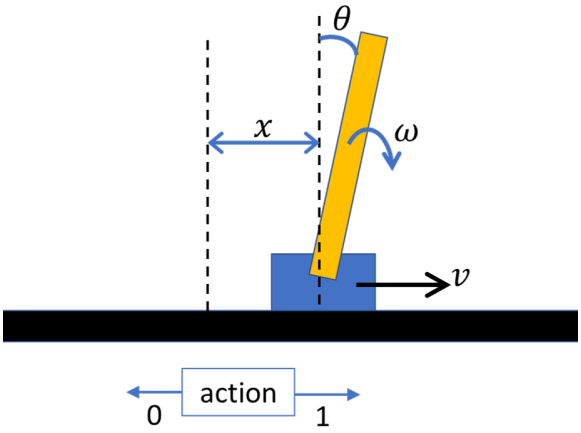
where n_w and n_θ are total number of weights and biases appearing in entire NLDT and $M = 20$ in our case.

4.7 Experiments: AIM and Procedure

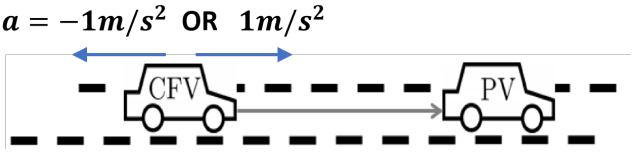
In this work, we conduct experiments to demonstrate the performance of NLDT on control problems involving discrete actions. Efficacy of the proposed approach of inducing NLDT is

shown on two *types* of control tasks as listed below:

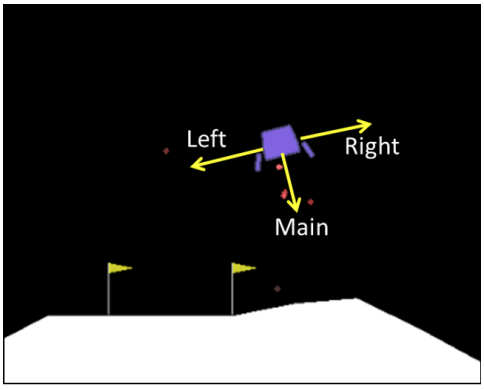
- Binary Discrete Action Space (B-DAS): The agent in these control tasks is allowed to have exactly two discrete actions. Environments considered for this task are CartPole and CarFollowing as shown in Figures 4.8a and 4.8b, respectively.
- Multiple Discrete Action Space (M-DAS): Action space of agent in these control tasks involves three or more discrete actions. Environments considered for this task are MountainCar and LunarLander as shown in Figures 4.2 and 4.8c, respectively.



(a) CartPole environment.



(b) CarFollowing environment.



(c) LunarLander environment.

Figure 4.8: Three control problems.

Through our experiments, we try to empirically demonstrate effects of training data size, data distribution (regular or balanced) and NLDT framework (binary-split or multi-split) on the performance of NLDT.

4.7.1 Experimental Setup

At first, a dedicated black-box AI is trained for each control tasks listed above. The method used to derive the black-box AI for each control task is provided in the corresponding sections. It is

to note here that training of the black-box AI is not the focus of our research. Our aim here is to decipher complicated and incomprehensible (but efficiently functioning) black-box AI controller by representing it in a relatively more interpretable format through Non-linear Decision Tree (NLDT). Once the black-box AI is trained, it is used to generate labelled training and testing datasets of *state-action* pairs (block 2 in Figure 4.7). The labelled training dataset is used to induce the NLDT using the recursive bilevel evolutionary algorithm [81] (Section 4.6.2).

The training dataset is generated by storing state-action values from the sequential interaction of black-box AI (such as DNN) with the corresponding environment. We generate two types of training datasets: *Regular* and *Balanced*. Discussion regarding the procedure used to generate *regular* and *balanced* dataset is explained next.

4.7.1.1 Creation of Regular Dataset

For creating a regular dataset of n_{total} datapoints, the sequential state-action data of different episodes is stored from the interaction of trained black-box AI with the corresponding environment. Since the datapoint (state-action pair) is stored for each time-step in sequential manner (if the episode terminates and the collected number of datapoints is less than n_{total}, then a new episode is invoked and the process of data collection repeats), there is no explicit control on the number of datapoints which will fall into a particular action category. Hence, the resulting data distribution might have an inherent bias. This bias in data distribution is more evident for environments involving more than two actions as shown in Table 4.1.

Table 4.1: Class Distribution of Regular Training Datasets for different problems. For each row, the i-th number in second column represents the number of datapoints belonging to i-th action.

Problem	**#Classes**	**Class Distribution**
CartPole	2	4997; 5003
CarFollowing	2	5237; 4763
MountainCar	3	3452; 495; 6053
LunarLander	4	3661; 1650; 3779; 910

Since some of the *actions* have lesser representation in the training dataset, it might negatively affect the closed-loop performance of the NLDT. To investigate this effect of bias, we create

balanced datasets with near-uniform distribution across all actions/classes. The procedure to create a balanced (or almost balanced) dataset is discussed next.

4.7.1.2 Creation of Balanced Dataset

In order to create a uniformly (or near-uniformly) distributed data across all actions in the dataset of n_{total} datapoints, data creation procedure similar to the one discussed above is implemented, i.e. data is stored from sequential time-steps. However, if the number of datapoints for a particular action reaches a threshold limit of $\lceil \frac{n_{total}}{n_{actions}} \rceil$ (where $n_{actions}$ is the total number of discrete actions and $\lceil x \rceil$ indicates the *round-up* function), no extra datapoints are collected for that action. It is to note that data for other actions (for which number of collected datapoints is still less than the prescribed threshold value) is still collected from sequential time-steps until total n_{total} datapoints are collected.

The *regular* and *balanced* data creation method is used to create *training* datasets only. Testing datasets are created with the *regular* dataset creation approach. Also, since for binary action spaces, the distribution is fairly uniform (Table 4.1), we don't create balanced training datasets for these problems (i.e. CartPole and CarFollowing).

4.8 Experiments and Analysis on Control Tasks with Binary Action Spaces

In this section, we conduct experiments and discuss results obtained by using our approach for control tasks involving two discrete actions, namely: 1) CartPole, and 2) CarFollowing.

4.8.1 CartPole Problem

As shown in Figure 4.8a, the CartPole problem comprises of four state variables: 1) x-position ($x \rightarrow x_0$), velocity in +ve x direction ($v \rightarrow x_1$), angular position from vertical ($\theta \rightarrow x_2$) and angular velocity ($\omega \rightarrow x_3$) and is controlled by applying force towards *left* (action 0) or *right* (action 1) to the cart. The objective is to balance the inverted pendulum (i.e. $-24\,\text{deg} \leq \theta \leq 24\,\text{deg}$) while also ensuring that the cart doesn't fall off from the platform (i.e. $-4.8 \leq x \leq 4.8$). For every time-step,

a reward value of 1 is received while θ is within ±24 deg. The maximum episode length is set to 200 time-steps.

A deep neural network (DNN) controller is trained on the *CartPole* environment using the PPO algorithm [61]. Table 4.2 shows the performance of NLDT on the training data sets of different sizes. It is observed that NLDT trained with 5,000 and 10,000 data points shows a robust open-loop performance and also produces 100% closed-loop performance. Keeping this in mind, we keep the training data size of 10,000 fixed across all control problems discussed in this dissertation. It is observed that NLDT$_{OL}$ trained with at least 5,000 data points shows a robust open-loop performance. The obtained NLDT$_{OL}$ has a about two rules with on an average three terms in the derived policy function.

Table 4.2: Effect of training data size on performance of NLDT$_{OL}$ on CartPole problem.

Training Data Size	**Training Accuracy**	**Testing Accuracy**	**# Rules**	**Rule Length**	**Cumulative Reward**	**Compl. Rate (Closed-loop)**
100	**97.00**	82.79	1.50	**3.30**	199.73	95.0
500	95.5	79.66	1.90	3.88	175.38	51.00
1,000	91.90	90.59	1.80	4.05	**200.00**	**100**
5,000	92.07	92.02	1.70	4.25	**200.00**	**100**
10,000	91.86	**92.05**	**1.30**	4.45	**200.00**	**100**

Interestingly, the same NLDT (without closed-loop training) also produces 100% closed-loop performance by achieving the maximum cumulative reward value of 200.

4.8.1.1 NLDT for CartPole Problem

One of the NLDT$_{OL}$ obtained for the CartPole environment is shown in Figure 4.9 in terms of normalized state variable vector $\widehat{\mathbf{x}}$.

The respective policy can be alternatively stated using the programmable `if-then-else` rule-structure as shown in Algorithm 5:

A little manipulation will reveal that for a correct control startegy, Action 0 must be invoked if following condition is true:

$$2.39 \leq \left(\frac{\widehat{x_0}}{\widehat{x_2}^2} + \frac{3.50}{\widehat{x_3}^2} \right) \leq 5.06,$$

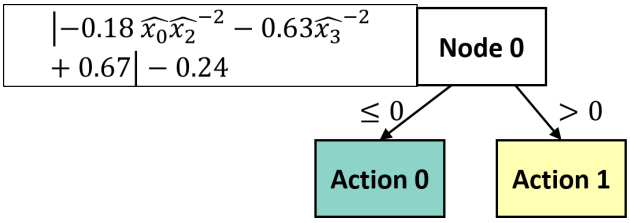


Figure 4.9: CartPole NLDT$_{OL}$ induced using 10,000 training samples. It is 91.45% accurate on the testing dataset but has 100% closed loop performance. Normalization constants are: $\mathbf{x}^{\min}$ = [-0.91, -0.43, -0.05, -0.40], $\mathbf{x}^{\max}$ = [1.37, 0.88, 0.10, 0.45].

Algorithm 5: CartPole Rules. Normalization constants are: x^{min} = [-0.91, -0.43, -0.05, -0.40], x^{max} = [1.37, 0.88, 0.10, 0.45].

if $\left|-0.18\widehat{x_0}\widehat{x_2}^{-2} - 0.63\widehat{x_3}^{-2} + 0.67\right| - 0.24 \leq 0$ **then**
  Action = 0
else
  Action = 1
end

otherwise, Action 1 must be invoked. First, notice that the above policy does not require the current velocity ($\widehat{x_1}$) to determine the left or right action movement. Second, for small values of angular position ($\widehat{x_2} \approx 1$) and angular velocity ($\widehat{x_3} \approx 1$), i.e. the pole is falling towards left, the above condition is always true. That is, the cart should be pushed towards left, thereby trying to stabilize the pole to vertical position. On the other hand, if the pole is falling towards right (large values of $\widehat{x_2} \approx 2$ and $\widehat{x_3} \approx 2$), the term in bracket will be smaller than 2.39 for all $\widehat{x_0} \in [1, 2]$, and the above policy suggests that Action 1 (push the cart towards right) must be invoked. When the pole is falling right, a push of the cart towards right helps to stabilize the pole towards its vertical position. These extreme case analyses are intuitive and our policy can be explained for its proper working, but what our NLDT approach is able to find is a precise rule for all situations of the state variables to control the Cart-Pole to a stable configuration, mainly using the blackbox-AI data.

4.8.2 CarFollowing Problem

This problem simulates a one dimensional car chasing scenario. We have developed a discretized version of the car following problem discussed in [82] (illustrated in Figure 4.8b), wherein the task

is to follow the car in the front which moves with a random acceleration profile (between $-1m/s^2$ and $+1m/s^2$) and maintain a safe distance of $d_{safe} = 30m$ from it. The rear car is controlled using two discrete acceleration values of $-1m/s^2$ (Action 0) and $+1m/s^2$ (Action 1). The car-chase episode terminates when the relative distance $d_{rel} = x_{front} - x_{rel}$ is either zero (i.e. collision case) or is greater than 150 m. At the start of the simulation, both cars start with the initial velocity of zero. A DNN policy for CarFollowing problem was obtained using a double Q-learning algorithm [83]. The reward function for the CarFollowing problem is shown in Figure 4.10, indicating that a relative distance close to 30 m produces the highest reward.

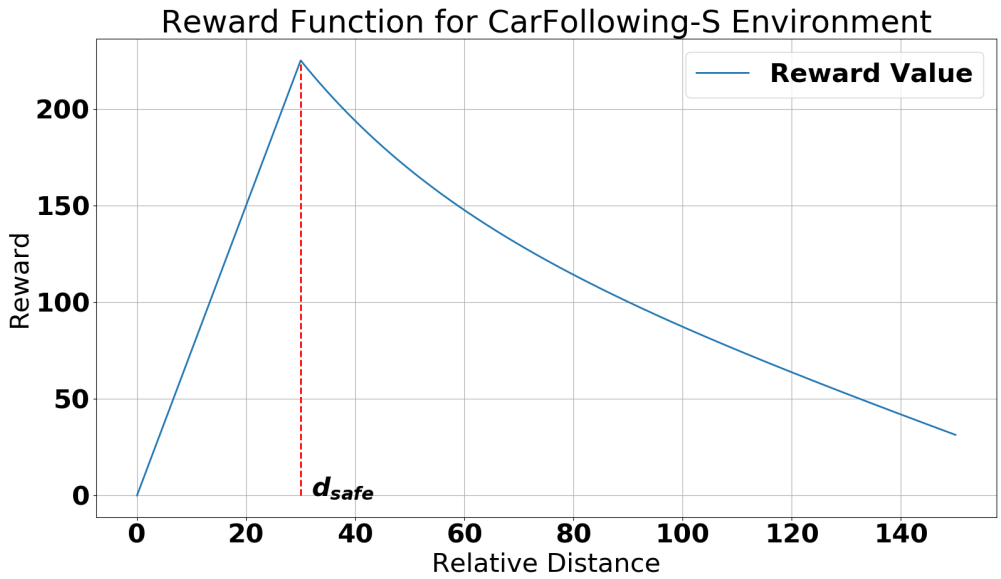


Figure 4.10: Reward function for CarFollowing environment.

It is to note here that unlike the CartPole control problem, where the dynamics of the system was deterministic, the dynamics of the CarFollowing problem is not deterministic due to the random acceleration profile with which the car in the front moves. This randomness introduced by the unpredictable behaviour of the front car makes this problem more challenging.

Results for the CarFollowing problem are shown in Table 4.3. An average open-loop accuracy of 96.53% is achieved with at most three rules, each having 3.28 terms on an average.

For this problem, we apply the closed-loop re-optimization (Blocks 4 and 5 to produce Block 6 in Figure 4.7) on the entire NLDT_{OL}. As shown Table 4.4, NLDT* is able to achieve better closed-

Table 4.3: Results on CarFollowing problem correspoing to open-loop training (NLDT$_{OL}$).

Train. Acc.	**Test. Acc.**	**Depth**	**# Rules**	**Rule Length**	**Compl. Rate**
96.41 ± 1.97	96.53 ± 1.90	1.90 ± 0.30	2.40 ± 0.66	3.28 ± 0.65	100 ± 0.00

loop performances (100% completion rate and better average cumulative reward). Figure 4.11 shows that NLDT* adheres the 30m gap between the cars more closely than original DNN or NLDT$_{OL}$.

Table 4.4: Closed-loop performance analysis after re-optimizing NLDT for CarFollowing problem (k = 10^3).

AI	Cumulative Reward		Compl. Rate
	Best	Avg ± Std	
DNN	174.16k	173.75k ±20.95	**100 ± 0.00**
NLDT$_{OL}$	174.15k	173.87k ±16.48	**100 ± 0.00**
NLDT*	**179.76k**	**179.71k ±0.95**	**100 ± 0.00**

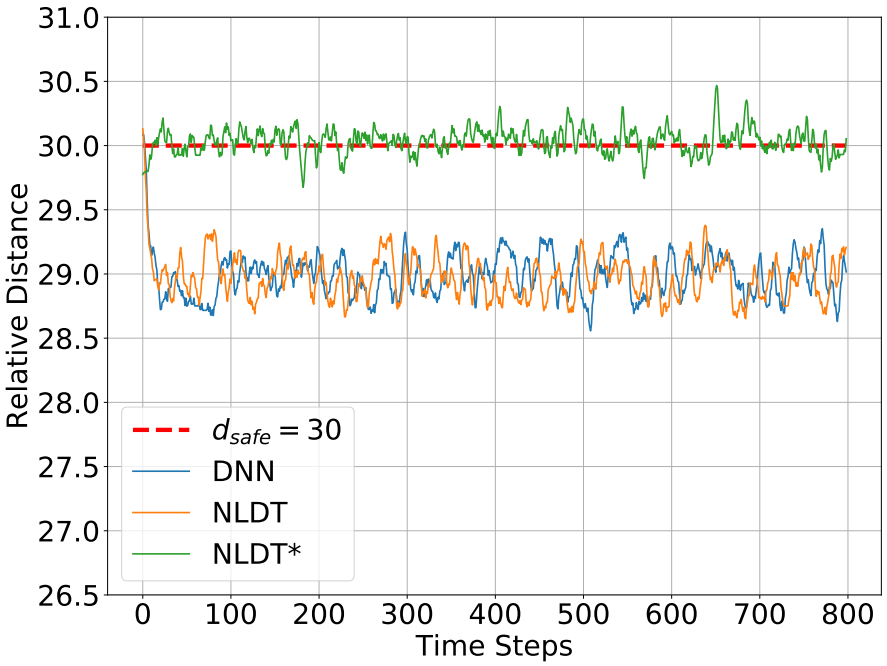


Figure 4.11: Relative distance plot for CarFollowing problem.

4.8.2.1 NLDT for CarFollowing Problem

The NLDT$_{OL}$ obtained for the CarFollowing problem is shown in Figure 4.12. The rule-set is

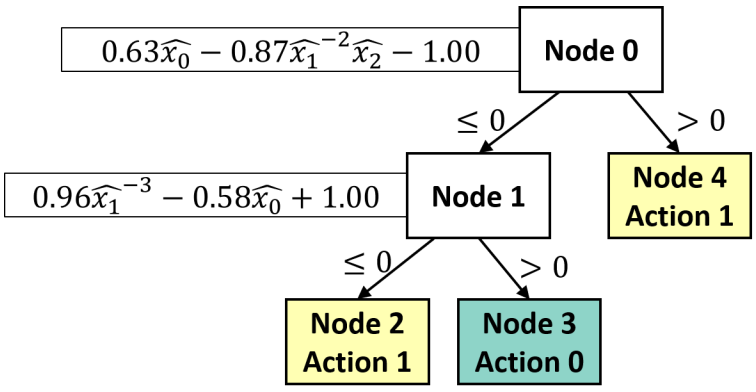


Figure 4.12: NLDT$_{OL}$ for the CarFollowing problem. Normalization constants are: $x^{\min}$ = [0.25, -7.93, -1.00], $x^{\max}$ = [30.30, 0.70, 1.00].

provided in its natural if-then-else form in Algorithm 6.

Algorithm 6: Ruleset corresponding to NLDT$_{OL}$ (Figure 4.12) of the CarFollowing problem.

```
if 0.63x̂0 − 0.87x̂1^−2 x̂2 − 1.00 ≤ 0 then
    if 0.96x̂1^−3 − 0.58x̂0 + 1.00 ≤ 0 then
        Action = 1
    else
        Action = 0
    end
else
    Action = 1
end
```

Recall that the physical meaning of state variables is: $x_0 \rightarrow d_{rel}$ (relative distance between front car and rear car), $x_1 \rightarrow v_{rel}$ (relative velocity between front car and rear car) and $x_2 \rightarrow a$ (acceleration value (-1 or $+1$ m/s^2) at the previous time step). Action = 1 stands for acceleration and Action = 0 denotes deceleration of the rear car in the next time step.

From the first rule (Node 0), it is clear that if the rear car is close to the front car ($\widehat{x_0} \approx 1$), the root function $f_0(\mathbf{x})$ is never going to be positive for any value of relative velocity or previous acceleration of the rear car (both $\widehat{x_1}$ and $\widehat{x_2}$ lying in [1,2]). Thus, Node 4 (Action = 1, indicating

acceleration of the rear car in the next time step) will never be invoked when the rear car is too close to the front car. Thus for $\widehat{x_0} \approx 1$, the control always passes to Node 1. A little analysis will also reveal that for $\widehat{x_0} \approx 1$, the rule $f_1(\mathbf{x}) > 0$ for any relative velocity $\widehat{x_1} \in [1, 2]$. This means that when the two cars are relatively close, only Node 3 gets fired to decelerate (Action = 0) the rear car. This policy is intuitively correct, as the only way to increase the gap between the cars is for the controlled rear car to be decelerating.

However, when the rear car is far away for which $\widehat{x_0} \approx 2$, Action 1 (Node 4) gets fired if $\widehat{x_1} > 1.829\sqrt{\widehat{x_2}}$. If the rear car was decelerating in the previous time step (meaning $\widehat{x_2} = 1$), the obtained NLDT recommends that the rear car should accelerate if $\widehat{x_1} \in [1.829, 2]$, or when the magnitude of the relative velocity is small, or when $x_1 \in [-0.776, 0.700]m/s$. This will help maintain the requisite distance between the cars. On the other hand, if the rear car was already accelerating in the previous time step ($\widehat{x_2} = 2$), Node 4 does not fire, as $\widehat{x_1}$ can never be more than $1.829\sqrt{2}$ and the control goes to Node 1 for another check. Thus, the rule in Node 0 makes a fine balance of the rear car's movement to keep it a safe distance away from the front car, based on the relative velocity, position, and previous acceleration status. When the control comes to Node 1, Action 1 (acceleration) is invoked if $\widehat{x_1} \geq 0.96/(0.58\widehat{x_0} - 1)$. For $\widehat{x_0} \approx 2$, this happens when $\widehat{x_1} > 1.817$ (meaning that when the magnitude of the relative velocity is small, or $x_1 \in [-0.879, 0.700]m/s$), the rear car should accelerate in the next time step. For all other negative but large relative velocities $x_1 \in [-7.930, 0.879]m/s$), meaning the rear car is rushing to catch up the front car, the rear car should decelerate in the next time step. From the black-box AI data, our proposed methodology is able to obtain a simple decision tree with two nonlinear rules to make a precise balance of movement of the rear car and also allowing us to understand the behavior of a balanced control strategy.

Results of NLDT's performance on problems with two discrete actions (Tables 4.2, 4.3 and 4.4) indicate that despite having a noticeable mismatch with the open-loop output of the oracle black-box policy π_{oracle}, the closed-loop performance of NLDT is at par or at times better than π_{oracle}. This observation suggests that certain state-action pairs are not of crucial importance

when it comes to executing the closed-loop control and, therefore, errors made in predicting these state-action events do not affect or deteriorate the closed-loop performance.

4.9 Experiments and Analysis on Multiple Discrete Action Space

In this section, we investigate the performance of proposed NLDT method to approximate the behavior of the parent ANN/black-box AI for control tasks involving more than two discrete actions. Here, as mentioned before, we compare the open-loop and closed-loop performances across different representations of NLDT, i.e. *binary-split NLDT* and *multi-split NLDT*, and two different training dataset distributions, namely, *regular* and *balanced*.

4.9.1 MountainCar Problem

A schematic of this environment is shown in Figure 4.2. The car starts somewhere near the bottom of the valley and the goal of the task is to reach the flag-post located on the right up-hill with non-negative velocity. The fuel is not enough to directly climb the hill and hence a control strategy needs to be devised to move car back (left up-hill), leverage the potential energy and then accelerate it to eventually reach the flag-post within 200 time-steps. The car receives the reward value of -1 for each time-step, until it reaches the flag-post where the reward value is zero. The car is controlled using three actions: *accelerate left* (action 0), *do nothing* (action 1) and *accelerate right* (action 2) by observing its state which is given by two state-variables: *x position* $\rightarrow x_0$ and *velocity* $v \rightarrow x_1$. We use the SARSA algorithm [84] with tile encoding to derive the black-box AI controller, which is represented in form of a tensor and has total $151,941$ elements.

Compilation of results of the NLDT controller induced using different NLDT-representations and training dataset distributions is presented in Table 4.5. A state-action plot of the black-box AI and the NLDT controller corresponding to the first row of Table 4.5 is provided in Figures 4.3a and 4.3b, respectively. It can be seen from the plots that about 8% mismatch in the open-loop performance (i.e. testing accuracy in Table 4.5) comes from the lower region of state-action plot (Figures 4.3a and 4.3b) due to highly non-linear output of the black-box AI controller. Again,

despite having this mismatch, the NDLT controller is able to achieve close to 100% closed-loop control performance. The regular dataset is able to produce a multi-split NLDT with three control

Table 4.5: Mountain car results. The numbers indicate average scores.

Balanced	Training Accuracy	Testing Accuracy	Depth	# Rules	Avg Rule Length	Completion Rate
Binary-split NLDT						
No	**90.61**	**91.92**	**2**	**2**	**3.00**	99.8
Yes	86.11	88.70	3	4	**3.00**	**100.0**
Multi-split NLDT						
No	**91.66**	**91.50**	**2**	**3**	**1.67**	**100.0**
Yes	88.53	88.44	**2**	4	2.50	**100.0**

rules with an average 1.67 terms in each rule to achieve 100% closed-loop performance. The multi-split NLDT has a comparable performance to that of binary-split NLDT.

4.9.2 NLDT for MountainCar Problem

The NLDT$_{OL}$ obtained for the MountainCar problem is shown below in Figure 4.13. The resulting rule-set is also shown in `if-then-else` statements in Algorithm 7.

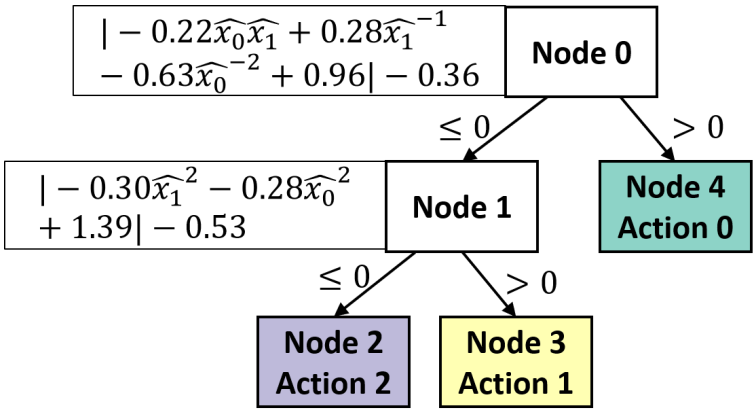


Figure 4.13: NLDT$_{OL}$ for MountainCar problem. Normalization constants: $x^{\min}$ = [-1.20, -0.06], $x^{\max}$ = [0.50, 0.06].

This rule-set corresponds to the plot shown in Figure 4.3b. A detail analysis of the two rules can be made to have a deeper understanding of the control policy.

Algorithm 7: MountainCar NLDT$_{OL}$. Normalization constants are: x^{min} = [-1.20, -0.06], x^{max} = [0.50, 0.06].

if $\left|-0.22\widehat{x_0}\widehat{x_1} + 0.28\widehat{x_1}^{-1} - 0.63\widehat{x_0}^{-2} + 0.96\right| - 0.36 \leq 0$ **then**
  if $\left|-0.30\widehat{x_1}^{2} - 0.28\widehat{x_0}^{2} + 1.39\right| - 0.53 \leq 0$ **then**
    Action = 2
  else
    Action = 1
  end
else
  Action = 0
end

4.9.3 LunarLander Problem

This problem is motivated from a classic problem of design of a rocket-controller. Here, the state of the lunar-lander is expressed with eight state variables, of which six can assume continuous real values, while the rest two are categorical, and can assume a Boolean value. The first six state variables indicate the (x, y) position, and velocity and angular orientation and angular velocity of the lunar-lander. The two Boolean state variables provide the indication regarding the left-leg and right-leg contact of lunar-lander with the ground terrain. The lunar-lander is controlled using four actions: action 0 $\rightarrow$ *do nothing*, action 1 $\rightarrow$ *fire left engine*, action 2 $\rightarrow$ *fire main engine* and action 3 $\rightarrow$ *fire right engine* as shown schematically in Figure 4.8c. The black-box DNN based controller for this problem is trained using the PPO algorithm [61] and involves two hidden layers of 64 nodes.

Table 4.6 provides the compilation of results obtained using open-loop training.

It is evident from the table that the binary-split NLDT$_{OL}$ representation is superior to the multi-split NLDT$_{OL}$ representation in terms of both open-loop and closed-loop performances. In this problem, a better closed-loop performance of NLDT$_{OL}$ is observed when the open-loop training is done on the balanced dataset. This indicates that under-represented actions in regular training dataset (see Table 4.1) limits the exposure to some crucial states during the training phase of NLDT. The crucial state-action pairs (or experience) necessary for closed-loop control gets captured while creating the balanced dataset, and so, despite having the lower open-loop performance on the testing

Table 4.6: LunarLander open-loop training (NLDT_{OL}) results. The numbers indicate average scores.

Balanced	Training Accuracy	Testing Accuracy	Depth	# Rules	Avg Rule Length	Completion Rate
Binary-split NLDT						
No	79.17	76.36	**3**	5.60	5.59	14.00
Yes	69.83	66.58	**3**	**4.40**	5.79	42.00
No	**87.43**	**81.74**	6	34.70	**4.94**	48.00
Yes	81.74	71.52	6	25.70	5.17	**93.00**
Multi-split NLDT						
No	75.84	69.68	**2**	**4**	**3.50**	0.11
Yes	68.31	70.79	**2**	5	5.80	21.60
No	**86.95**	**79.40**	6	37	3.84	17.70
Yes	82.17	71.08	6	41	3.95	**89.50**

dataset (which is *regular*), the overall closed-loop performance improves. The best performance is observed with a binary-split NLDT_{OL} trained on the balanced dataset.

4.9.3.1 NLDT for LunarLander Problem

The topology of one of the binary-split NLDT_{OL} obtained after open-loop training using balanced dataset is shown in Figure 4.14.

This NLDT_{OL} successfully lands the lander in 93.4% episodes and has in total 26 rules each having about 4.15 terms involving state variables.

It is understandable that a complex control task involving many state variables cannot be simplified or made interpretable with just one or two control rules. A separate study focusing at identifying crucial state-action pairs would help us understand this phenomenon better and we plan to do this in our future work. Next, we use a part of the NLDT_{OL} from the root node to obtain the pruned $\text{NLDT}_{OL}^{(P)}$ (step 'B' in Figure 4.7) and re-optimize all weights ($\mathbf{W}$) and biases ($\mathbf{\Theta}$) using the procedure discussed in Section 4.6.3 (shown by orange box in Figure 4.7) to find closed-loop NLDT*.

Table 4.7 shows that for the pruned NLDT-3 which comprises of the top three layers and involves only four rules of original 26-rule NLDT_{OL} (i.e. NLDT-6), the closed-loop performance increases

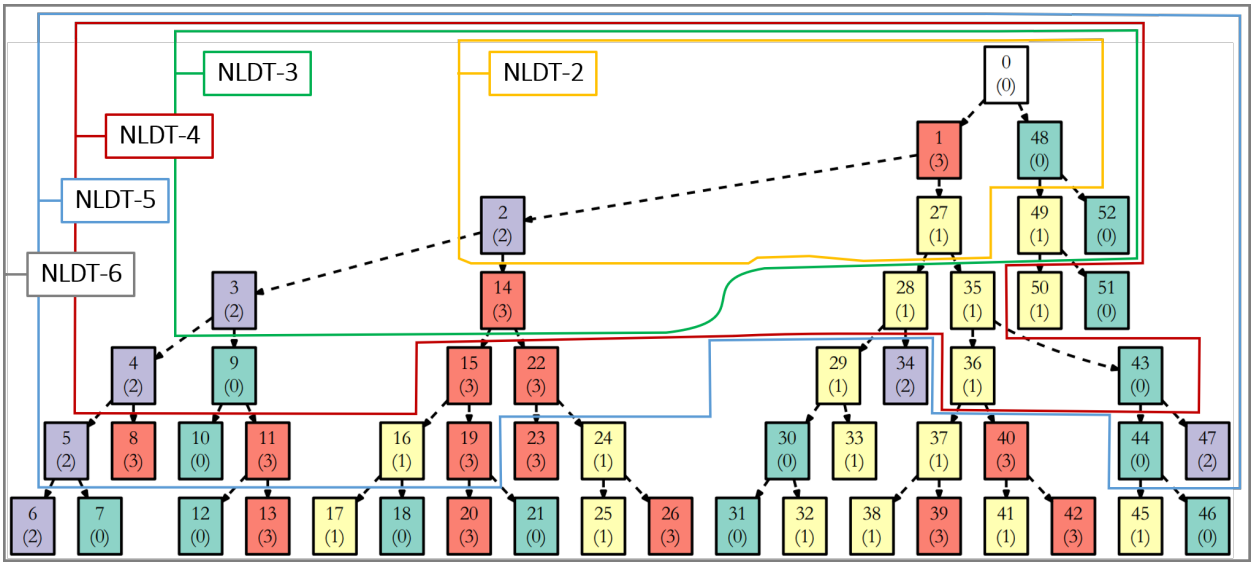


Figure 4.14: NLDT-6 (with 26 rules) and other lower depth NLDTs for the LunarLander problem. Lower depth NLDTs are extracted from the depth-6 NLDT. Each node has an associated node-id (on top) and a node-class (mentioned in bottom within parenthesis). Table 4.7 in provides results on closed-loop performance obtained using these trees *before* and *after* applying re-optimization on rule-sets using the closed-loop training procedure.

from 51% to 96% (NLDT*-3 results in Table 4.7) after re-optimizing its weights and biases with closed-loop training.

The resulting NLDT with its associated four rules are shown in Figure 4.15.

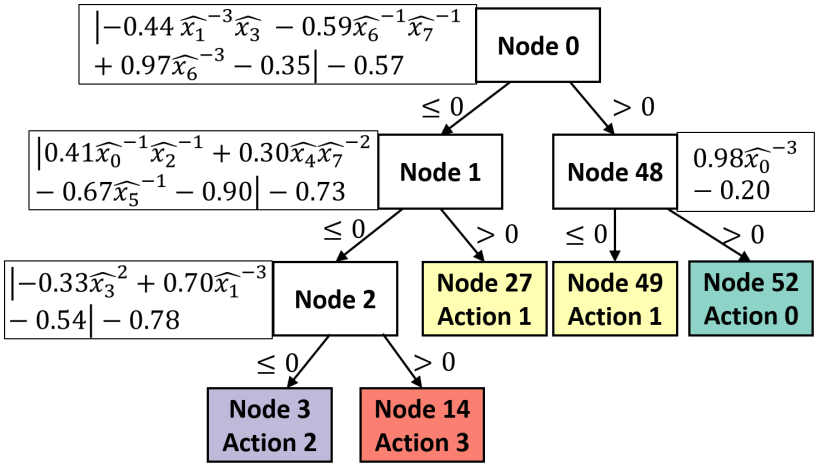


Figure 4.15: Final NLDT*-3 for LunarLander prob. $\widehat{x_i}$ is a normalized state variable (see Section 4.6.1).

As shown in Table 4.7, the NLDT* with just two rules (NLDT-2) is too simplistic and does

not recover well after re-optimization. However, the NLDT*s with four and seven rules achieve a near 100% closed-loop performance. Clearly, an NLDT* with more rules (NLDT-5 and NLDT-6) are not worth considering since both closed-loop performances and the size of rule-sets are worse than NLDT*-4. Note that DNN produces a better reward, but not enough completion rate, and the policy is more complex with 4,996 parameters.

Table 4.7: Closed-loop performance on LunarLander problem with and without re-optimization on 26-rule NLDT$_{OL}$. Number of rules are specified in brackets for each NLDT and total parameters for the DNN is marked.

Re-Opt.	**NLDT-2 (2)**	**NLDT-3 (4)**	**NLDT-4 (7)**	**NLDT-5 (13)**	**NLDT-6 (26)**	**DNN (4,996)**
Cumulative Reward						
Before	−1675.77 ± 164.29	42.96 ± 13.83	54.24 ± 27.44	56.16 ± 23.50	**169.43 ±23.96**	247.27 ± 3.90
After	−133.95 ± 2.51	231.42 ± 17.95	**234.98 ±22.25**	182.87 ± 21.92	214.94 ± 17.31	
Completion Rate						
Before	0.00 ± 0.00	51.00 ± 3.26	82.00 ± 9.80	79.00 ± 7.66	**93.00 ±3.30**	94.00 ±1.96
After	48.00 ± 7.38	96.00 ± 2.77	**99.00 ±1.71**	93.00 ± 7.59	94.00 ± 4.45	

To demonstrate the efficacy and repeatability of our proposed approach, we perform another run of the open-loop and closed-loop training and obtain a slightly different NLDT*-3, which is shown in Figure 4.16. This NLDT also has four rules, which are shown in Table 4.8. Four rules corresponding to the pruned NLDT$^{(P)}_{OL}$ (Depth 3) are also shown in the table for comparison. It can be noticed that the re-optimization of NLDT through closed-loop training (Section 4.6.3) modifies the values of coefficients and biases, however the basic structure of all four rules remains intact.

Figure 4.17 shows the closed-loop training curve for generating NLDT* from Depth-3 NLDT$^{(P)}_{OL}$. The objective is to maximize the closed-loop fitness (reward) F_{CL} (Eq. 4.11) which is expressed as the average of the cumulative reward R_e collected over M episodes. It is evident that the cumulative reward for the best-population member climbs to the target reward of 200 at around 25-th generation and the average cumulative reward of the population also catches up the best cumulative reward value with generations.

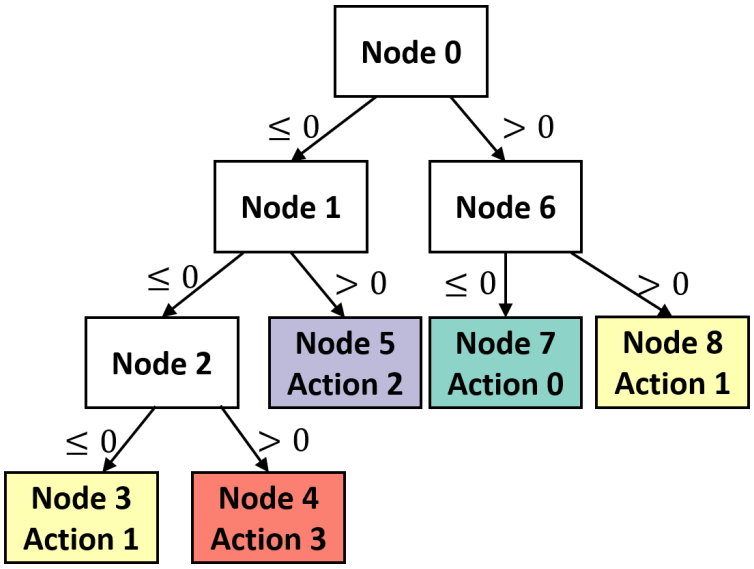


Figure 4.16: Topology of Depth-3 NLDT$^{(P)}_{OL}$ obtained from a different run on the LunarLander problem. The equations corresponding the conditional-nodes before and after re-optimization are provided in Table 4.8.

A visualization of the real-time closed-loop performance obtained using this new NLDT (Figure 4.16) for two different rule-sets (i.e. before applying re-optimization and after applying the re-optimization through closed-loop training) is shown in `https://youtu.be/DByYWTQ6X3E`. It can be observed in the video that the closed-loop control executed using the Depth-3 NLDT$^{(P)}_{OL}$ comprising of rules obtained directly from the open-loop training (i.e. without any re-optimization) is able to bring the LunarLander close to the target. However the LunarLander hovers above the landing pad and the Depth-3 NLDT$^{(P)}_{OL}$ is unable to land it in most occasions. Episodes in these cases are terminated after the flight-time runs out. On the other hand, the Depth-3 NLDT* comprising of rule-sets obtained after re-optimization through closed-loop training is able to successfully land the LunarLander.

4.10 Conclusions

In this work, we have proposed a two-step strategy to arrive at hierarchical and *relatively* interpretable rulesets using a nonlinear decision tree (NLDT) concept to facilitate an explanation of the working principles of AI-based policies. The NLDT training phases use recent advances in

Table 4.8: NLDT rules before and after the closed-loop training for LunarLander problem, for which NLDT* is shown in Figure 4.16. Video showing the simulation output of the performance of NLDTs with rule-sets mentioned in this table can be found at `https://youtu.be/DByYWTQ6X3E`. Respective minimum and maximum state variables are $x^{\min}$ = [-0.38, -0.08, -0.80, -0.88, -0.42, -0.85, 0.00, 0.00], $x^{\max}$ = [0.46, 1.52, 0.80, 0.50, 0.43, 0.95, 1.00, 1.00], respectively.

Node	Rules before Re-optimization (Depth-3 $\text{NLDT}_{OL}^{(P)}$)
0	$\left\lvert -0.23\widehat{x_0}\widehat{x_2}^{-1}\widehat{x_6}^{-1}\widehat{x_7}^{-1} - 1.00\widehat{x_1}^{-1}\widehat{x_6} - 0.79\widehat{x_0}^{-1}\widehat{x_1}^{-1}\widehat{x_6}^{2} + 0.83\right\rvert - 0.85$
1	$0.17\widehat{x_2}^{-1} - 0.64\widehat{x_3}\widehat{x_7}^{-1} + 0.90\widehat{x_1}^{-2}\widehat{x_6}^{-2}\widehat{x_7}^{-3} + 0.29$
2	$0.82\widehat{x_7}^{-1} + 0.52\widehat{x_0}^{-1}\widehat{x_4}\widehat{x_6}^{-1} - 0.59\widehat{x_4}^{-1} - 0.95$
6	$\left\lvert -0.16\widehat{x_4}^{-3}\widehat{x_6}^{-3}\widehat{x_7} - 0.86\widehat{x_0}\widehat{x_5}^{-1}\widehat{x_6}^{-3} + 1.00\widehat{x_4}\widehat{x_6}^{-1} - 0.70\right\rvert - 0.26$

Node	Rules after Re-optimization (Depth-3 NLDT*)
0	$\left\lvert -0.39\widehat{x_0}\widehat{x_2}^{-1}\widehat{x_6}^{-1}\widehat{x_7}^{-1} - 0.96\widehat{x_1}^{-1}\widehat{x_6} - 0.12\widehat{x_0}^{-1}\widehat{x_1}^{-1}\widehat{x_6}^{2} + 0.89\right\rvert - 0.80$
1	$0.17\widehat{x_2}^{-1} - 0.78\widehat{x_3}\widehat{x_7}^{-1} + 0.90\widehat{x_1}^{-2}\widehat{x_6}^{-2}\widehat{x_7}^{-3} + 0.35$
2	$0.82\widehat{x_7}^{-1} + 0.52\widehat{x_0}^{-1}\widehat{x_4}\widehat{x_6}^{-1} - 0.59\widehat{x_4}^{-1} - 0.96$
6	$\left\lvert -\left(1.3\times 10^{-3}\right)\widehat{x_4}^{-3}\widehat{x_6}^{-3}\widehat{x_7} - 0.86\widehat{x_0}\widehat{x_5}^{-1}\widehat{x_6}^{-3} + 0.65\widehat{x_4}\widehat{x_6}^{-1} - 0.42\right\rvert - 0.26$

nonlinear optimization to focus its search on rule structure and details describing weights and biases of the rules by using a bilevel optimization algorithm. Starting with an open-loop training, which is relatively fast (due computationally fast fitness evaluation) but uses only time-instant state-action data, we have proposed a final closed-loop training phase in which the complete or a part of the open-loop NLDT is re-optimized for weights and biases using complete episode data. Results on popular discrete action problems have amply demonstrated the usefulness of the proposed overall approach.

This proof-of-principle study encourages us to pursue a number of further studies. First, the scalability of the NLDT approach to large-dimensional state-action space problems must now be explored. A study conducted in Chapter 3 on binary classification of dominated versus non-dominated data in multi-objective problems was successfully extended to 500-variable problems. While it is encouraging, the use of customization methods for initialization and genetic operators using problem heuristics and/or recently proposed *innovization* methods [46] in the upper level problem can be tried. Second, this study has used a computationally fast open-loop accuracy measure as the fitness for evolution of the NLDT_{OL}. This is because, in general, an NLDT_{OL} with

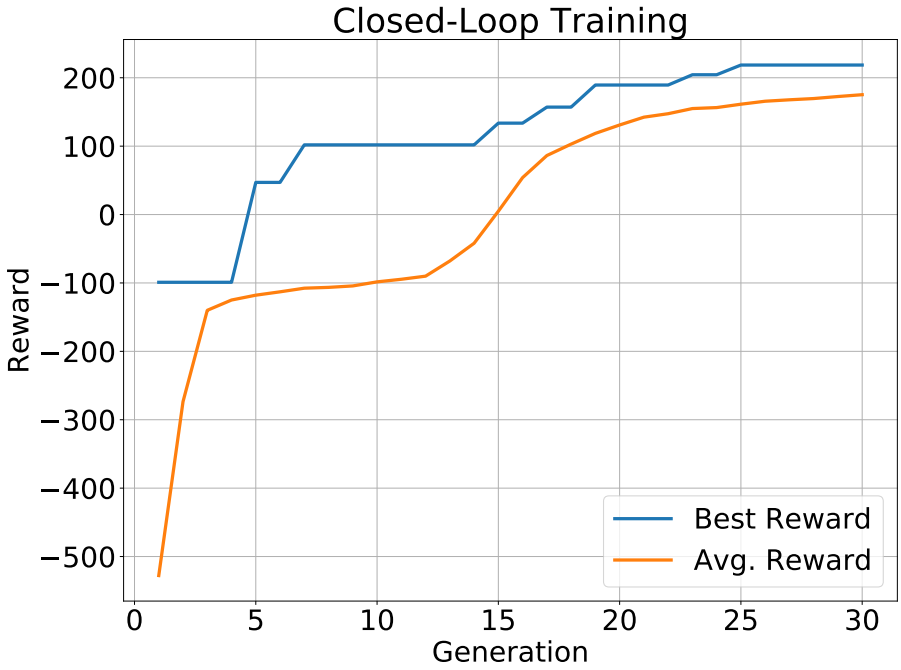


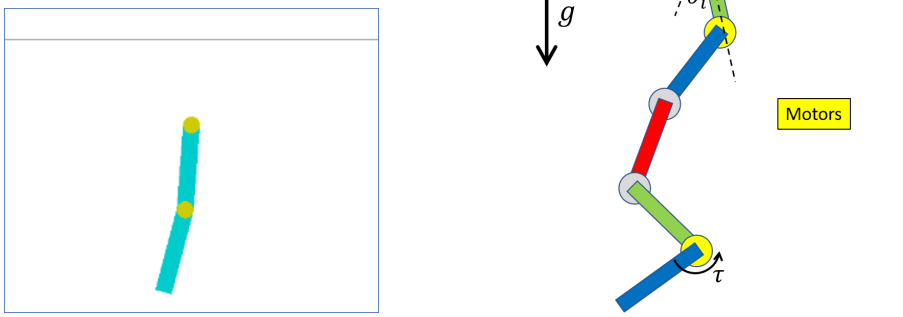
Figure 4.17: Closed-loop training plot for finetuning the rule-set corresponding to depth-3 NLDT$^{(P)}_{OL}$ (Table 4.8) to obtain NLDT* for LunarLander problem.

a high open-loop accuracy is likely to achieve a high closed-loop performance. However, we have observed here that a high closed-loop performance is achievable with a NLDT$_{OL}$ having somewhat degraded open-loop performance, but re-optimized using closed-loop performance metrics. Thus, a method to identify the crucial (open-loop) states from the AI-based simulation dataset that improves the closed-loop performance would be another interesting step for deriving NLDT$_{OL}$. This may eliminate the need for re-optimization through closed-loop training. Third, a more comprehensive study using closed-loop performance and respective complexity as two conflicting objectives for a bi-objective NLDT search would produce multiple trade-off control rule-sets. Such a study can, not only make the whole search process faster due to the expected similarities among multiple policies, they will also enable users to choose a single policy solution from a set of accuracy-complexity trade-off solutions.

CHAPTER 5

SCALE-UP STUDY AND IMPROVISATION

In this chapter, we focus at how the overall algorithm developed in previous chapter can be made more efficient in terms of – *training time* and *scalability*. To this purpose, we introduce a benchmark problem of planar serial robotic manipulator. This problem is inspired from the classical Acrobot control problem [64]. A schematic of the acrobot problem and serial manipulator problem is provided in Figure 5.1a and 5.1b respectively.



(a) Acrobot (b) Planar Serial Manipulator

Figure 5.1: Acrobot and a customized Planar Serial Manipulator benchmark problems.

The state-space for acrobot environment (Figure 5.1a) is six dimensional with following state variables: $sin(\theta_1), cos(\theta_1), sin(\theta_2), cos(\theta_2), \omega_1$ and ω_2, where θ_1 is the angle the first link makes with the vertical axis and θ_2 is the angle the longitudinal axis of the second link makes with that of the first one. The second joint between the first and second link is actuated with a motor. In case of planar manipulator (Figure 5.1b), the sate space comprises of angular position θ_i and angular velocity ω_i of each joint. Thus, for a n-link manipulator involving n revolute joints, the state space would be $2n$ dimensional. The motor is located at the last joint of the manipulator and is actuated

using three torque values: $-\tau, 0$ and τ. Each link is of 1 unit length and has its center-of-mass at its geometric center. The motor is assumed to be massless for the sake of simplicity. The base of the planar-manipulator is located at $(0, 0, 0)$ and the motion of the planar manipulator is limited to the XZ plane. There is a downward gravitational pull (g) of 10 units (i.e. -10 along vertical Z axis). Torque is applied along the Y-axis. The task in this problem is to take the end-effector (i.e. tip of the last link) of the serial-manipulator to a desired height of H units by supplying torque to the motor located at the last joint (joint between $(n-1)$-th and n-th link). The difficulty of this benchmark problem can be adjusted by

1. changing the number of links,
2. changing the value of desired height level H,
3. changing the value of torque τ and
4. placing extra motors at other joints.

We simulate the mechanics of the planar serial manipulator using *PyBullet* [85]: a Python based physics engine.

In our work, we provide two case-scenarios by focusing at the first three points of the above list. As mentioned before, by changing the number of links, the dimension of the state-space changes. The dimension of the action-space depends on the number of motors used. In the present work, we keep the number of motors fixed to one.

The details regarding two environments which are created and studied in this chapter are summarized in Table 5.1.

Table 5.1: Details regarding custom designed Planar Serial Manipulator environments.

Env. Name	**# Links** (n)	**Motor Torque** (τ)	**Desired Height** (H)	**# State Vars.**
5-Link Manipulator	5	1,000	+2	10
10-Link Manipulator	10	2,000	+2	20

The reward function $r(\mathbf{x}, A)$ is given by the following equation

$$r(\mathbf{x}) = \begin{cases} -1 - \left(\frac{H+1-z_E}{n+H+1}\right)^2 \text{ if } z_E < H, \\ 100 \text{ if } z_E \geq \text{H}, \end{cases} \tag{5.1}$$

where z_E is the vertical location of the end-effector. The minimium value for z_E is $-n$ when the entire manipulator is stretched to its full length and all joint angles (i.e. θ_i) are 0.

At the beginning of an episode, joint angles θ_i of the manipulator are randomly initialized between -5 deg and +5 deg, and the angular velocities ω_i are initialized to a value between -0.5 rad/sec and +0.5 rad/sec.

In next sections we discuss results obtained on the above two custom designed environments by using different procedures of inducing NLDE$_{OL}$. The black-box AI (DNN) is trained using the PPO algorithm [61].

5.1 Ablation Study for Open-loop Training

In this section, we launch two separate studies related to open-loop training procedure (see Figure 4.7). Following our results from Chapter 4, we restrict our discussion to binary-split NLDTs. It was seen in the previous chapter that the open-loop training is conducted using the hierarchical bilevel-optimization algorithm, which is discussed at length in Chapter 3. A dedicated bilevel-optimization algorithm is invoked to derive the split-rule $f(\mathbf{x})$ at a given conditional node. The upper-level search is executed using a discrete version of a genetic algorithm and the lower-level search is realized through an efficient real-coded genetic algorithm (RGA). Evolutionary algorithms are in general considered robust and have a potential to conduct more global search. However, being population driven, their search-speed is often less that of classical optimization algorithms. In this section, we study the effect of replacing the real-coded algorithm with a classical *sequential quadratic programming* (SQP) optimization algorithm in the lower-level of the overall bilevel algorithm for obtaining NLDT$_{OL}$. Later, closed-loop training based on real-code genetic algorithm is applied to NLDT$_{OL}$ to obtain NLDT* by re-optimizing the real valued coefficients of

NLDT$_{OL}$ (section 4.6.3). We use the *SciPy* [86] implementation of SQP. The initial point required for this algorithm is obtained using the dipole concept (Figure 3.7, Eq. 3.11).

For analysis, we induce the NLDT$_{OL}$ of depth-3 on the balanced training dataset (section 4.7.1.2) of 10,000 datapoints. The testing dataset is regular (see section 4.7.1.1) and comprises of 10,000 datapoints. Comparison of accuracy scores and average training time of inducing NLDT$_{OL}$ by using SQP and RGA algorithm at lower-level is provided in Table 5.2. For a given procedure (SQP or RGA) the best NLDT$_{OL}$ from 10 independent runs is chosen and is re-optimized using closed-loop training (section 4.6.3). Statistics regarding closed-loop performance of NLDT* is shown in the last two columns of Table 5.2. It is to note here that the closed-loop training is done using the real-coded genetic algorithm (RGA) discussed in section 4.6.3.

Table 5.2: Comparing performance of different lower-level optimization algorithms. For comparison, closed-loop performance of the original DNN policy is also reported.

	Open-Loop NLDT$_{OL}$			**Closed-Loop** NLDT*	
Algo. Name	**Training Accuracy**	**Testing Accuracy**	**Training Time (s)**	**Cumulative Reward**	**Completion Rate**
5-Link Manipulator					
SQP	62.46 ± 2.01	**69.34 ± 5.39**	**15.29 ± 4.95**	**−91.52 ± 14.42**	**97.92 ± 1.93**
RGA	**71.14 ± 1.77**	69.17 ± 4.39	1091.56 ± 319.18	−123.48 ± 25.11	97.00 ± 5.13
DNN	NA	NA	NA	−170.90 ± 42.57	89.00 ± 5.19
10-Link Manipulator					
SQP	56.57 ± 1.00	55.64 ± 3.58	**39.27 ± 11.63**	−318.82 ± 15.52	**96.00 ± 3.92**
RGA	**65.84 ± 0.85**	**62.60 ± 3.09**	2860.96 ± 789.49	**−281.88 ± 9.38**	95.00 ± 4.31
DNN	NA	NA	NA	−325.86 ± 4.63	85.88 ± 1.94

It can be observed from the results that open-loop training done with SQP in lower-level is about 70 times faster than the training done using RGA at lower level. The training done with RGA has a better overall open-loop performance. Thus, if the task is to closely mimic the behavior of black-box AI or if only a high classification accuracy is desired (in case of classification problems), then RGA is the recommended algorithm for lower-level optimization to obtain NLDT$_{OL}$. However, NLDT* obtained after re-optimizing NLDT$_{OL}$ corresponding to SQP and RGA have similar closed-loop completion rate (last column of Table 5.2). This implies that despite low open-loop accuracy scores, the open-loop training done using SQP in lower-level was able to successfully determine

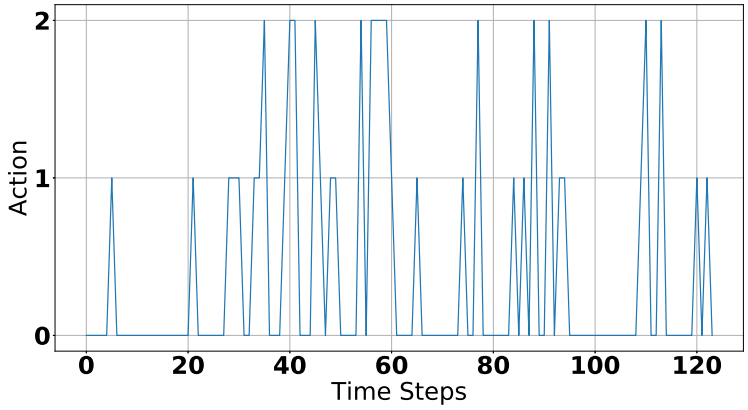
the template of split-rules $f(\mathbf{x})$ and the topology of NLDT, which upon re-optimization via closed-loop training algorithm could fetch a decent performing NLDT*. During open-loop training, the search on weights and coefficient using SQP was possibly not as perfect as compared to the one obtained through RGA, however, the re-optimization done through closed-loop training could compensate this shortcoming of SQP algorithm and produce NLDT* with a respectable closed-loop performance. Additionally, in either cases, the NLDT* obtained always had a better closed-loop performance than the original black-box DNN policy. This observation suggests that it is preferable to use SQP in lower-level during open-loop training to quickly arrive at a rough structure of NLDT_{OL} and then use closed-loop training to derive a high performing NLDT*. Another important thing to note here is that the open-loop training time for obtaining NLDT_{OL} for 10-link manipulator problem is about 2.5 times less than the corresponding training time for obtaining NLDT_{OL} for the 5-link manipulator problem. This is because, the population size in upper-level GA for 10-link manipulator problem (20 state variables) is twice that of the one for the 5-link manipulator problem (10 state variables). Also, the upper-level search in the high-dimensional space could possibly take more generations than it will take for lower-dimensional search spaces to converge.

5.2 Closed-loop Visualization

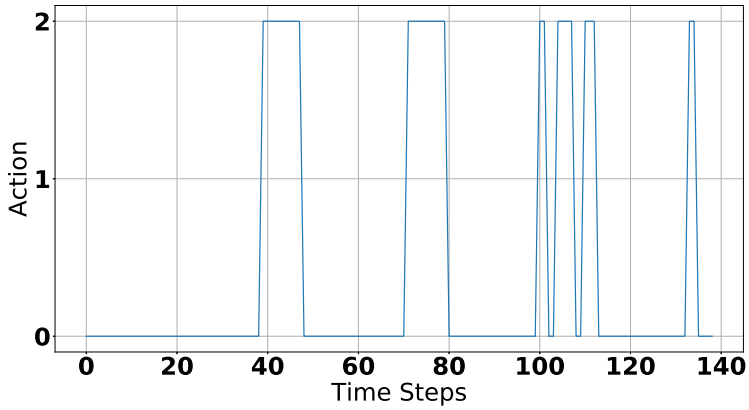
In this section we provide a visual insight into the closed-loop performance of NLDT* and DNN which we derived in the previous section. In our case, the frequency of the simulation is set to 240Hz, meaning that the transition to the next state is calculated using the time-step of 1/240 seconds. Geometrically speaking, this implies that the Euclidean distance between states from neighboring time-steps would be small. The AI (DNN or NLDT*) outputs the action value of 0 ($-\tau$ torque), 1 (0 torque) or 2 ($+\tau$ torque) for a given input state. Action Vs Time plots corresponding to different closed-loop simulation runs obtained by using DNN, NLDT* (SQP)[1] and NLDT* (RGA)[2] as controllers is shown in Figure 5.2 and 5.3 for 5-link and 10-link manipulator problems respectively.

[1] NLDT* (SQP) indicates that the corresponding NLDT_{OL} was derived using the SQP algorithm in the lower-level

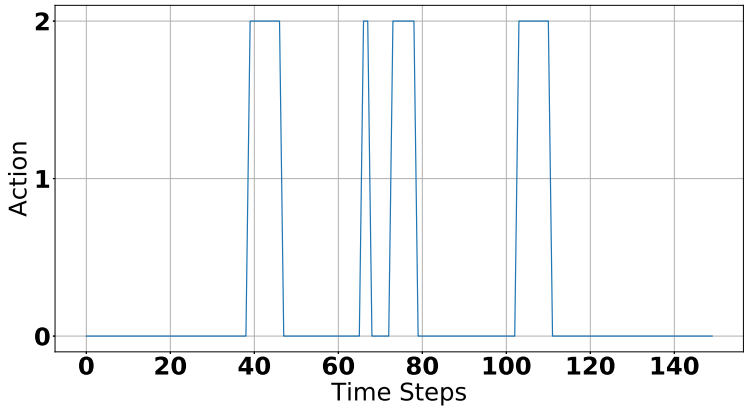
[2] NLDT* (RGA) indicates that the corresponding NLDT_{OL} was derived using the RGA algorithm in the lower-level



(a) DNN



(b) NLDT* (SQP)

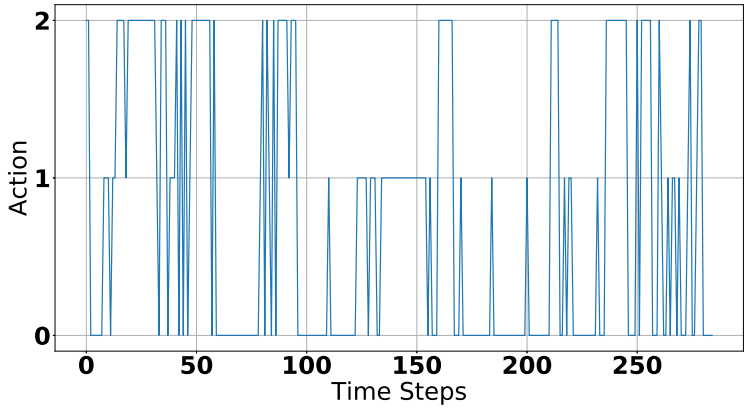


(c) NLDT* (RGA)

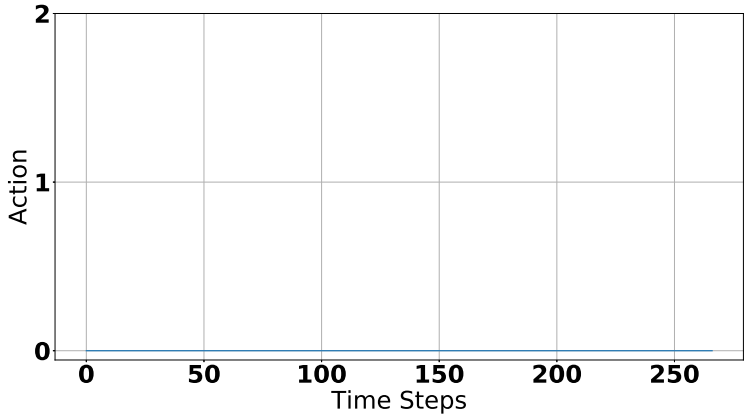
Figure 5.2: Action Vs. Time plot for 5-Link manipulator problem. Figure 5.2b provides the plot for NLDT* which is obtained from the NLDT$_{OL}$ trained using SQP algorithm in lower-level. Similarly, Figure 5.2c provides the plot for NLDT* which is obtained from the NLDT$_{OL}$ trained using RGA algorithm in lower-level.

Certain key observations can be made by looking at the plots in Figure 5.2. The control output for DNN is more erratic, with sudden jerks as compared to the control output of NLDT* (SQP) and NLDT* (RGA). The performance of NLDT* in Figures 5.2b and 5.2c is smooth and regular. This behaviour can be due to the involvement of a relatively simpler non-linear rules (as compared to the complicated non-linear rule represented by DNN) which are captured inside NLDT*. This is equivalent to the observation we made for the mountain car problem in Figures 4.3a and 4.3b, wherein the black-box AI had a very erratic behavior for the region of state-space in the lower-half of the state-action plot, while the output of NLDT was more smooth. Additionally, it was seen in Table 5.2 that the NLDT* (irrespective of how its predecessor NLDT$_{OL}$ was obtained, i.e. either through SQP or RGA in lower-level) showed better closed-loop performance than the parent DNN policy. This observation implies that simpler rules expressed in the form of a nonlinear decision tree have better generalizability, thereby giving more robust performance for randomly initialized control problems. A careful investigation to the plots in Figure 5.2b and 5.2c reveals that only two out of three allowable actions are required to efficiently execute the given control task of lifting the end-effector of a 5-link serial manipulator. This concept will be used to re-engineer the NLDT*, a discussion regarding which is provided in the next section.

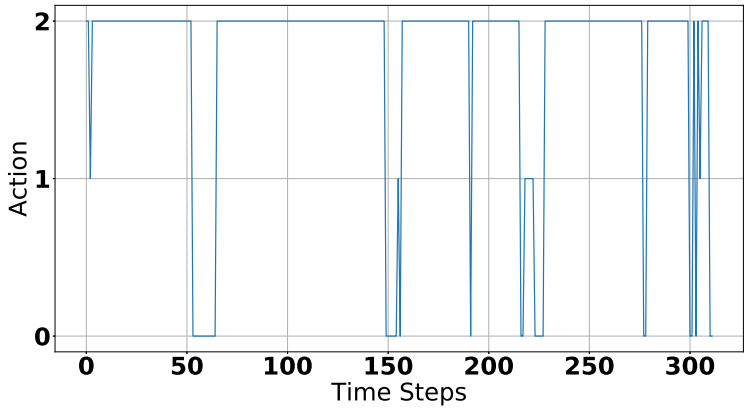
A similar argument can be made from plots in Figure 5.3 for 10-link manipulator problem. This is a slightly difficult problem to solve than the 5-link version since it involves twice the number of state variables. Interestingly, the search for NLDT* provides us with the simplest solution to this problem to lift the end-effector to the desired height of 2 units above the base. The simplest solution here is to give a constant torque in one direction as shown in Figure 5.3b. However, for this problem the best closed-loop performance in terms of cumulative reward (second last column of Table 5.2) is obtained using the control strategy corresponding to NLDT* (RGA) (Figure 5.3c). Here too, for most of the states, only one action is required, and occasionally other actions are invoked.



(a) DNN



(b) NLDT* (SQP)



(c) NLDT* (RGA)

Figure 5.3: Action Vs. Time plot for 10-Link manipulator problem. Figure 5.3b provides the plot for NLDT* which is obtained from the NLDT$_{OL}$ trained using SQP algorithm in lower-level. Similarly, Figure 5.3c provides the plot for NLDT* which is obtained from the NLDT$_{OL}$ trained using RGA algorithm in lower-level.

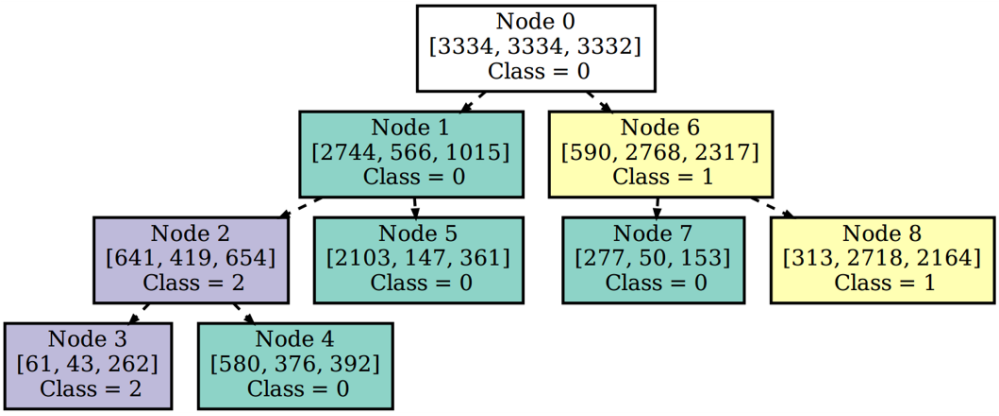
As mentioned before, in the next section we will discuss a post-processing approach to further simplify the NLDT*.

5.3 Reengineering NLDT*

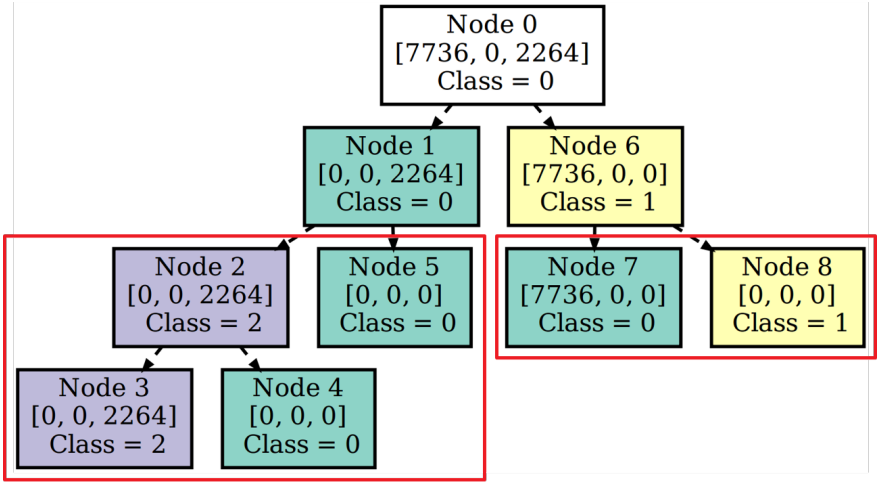
It was seen in action-time plots in Figure 5.2b, 5.2c, 5.3b and 5.3c that not all actions are required to perform a given control task. Also, it might be possible that while performing a closed-loop control using NLDT, not all branches and nodes of NLDT are visited. Thus, the portion of the NLDT which is not being utilized or is getting utilized very rarely can be pruned and the overall NLDT architecture can be made simpler. To illustrate this idea, we consider the NLDT which is derived for the 5-link manipulator problem. The topology of the best performing NLDT$_{OL}$ (SQP) for the 5-link manipulator problem is shown Figure 5.4a.

As mentioned before, this NLDT$_{OL}$ is trained on a balanced training dataset which is generated by collecting state-action pairs using parent DNN controller. In the figure, for each node, the information regarding its node-id, class distribution (given in square parenthesis) and the most dominating class is provided. Other than the root-node (Node 0), all nodes are colored to indicate the dominating class, however, it is to note that only the class associated to leaf-nodes carry the actual meaning while predicting the action for a given input state. This NLDT$_{OL}$ comprises of three split-rules in total. The class-distribution for each node is obtained by counting how many datapoints from the balanced training dataset visited a given node. Thus, the root node comprises of all datapoints (total 10,000), which are then scattered according to the split-rules present at each conditional node.

Figure 5.4b provides topology of NLDT* which is obtained after re-optimizing NLDT$_{OL}$ of Figure 5.4a using closed-loop training. As discussed in section 4.6 and 4.6.3, the topology of the tree and the structure of non-linear rules is identical for both: NLDT$_{OL}$ and NDLT*. However, the weights and biases of NLDT* are updated to enhance the closed-loop control performance. Similar to NLDT$_{OL}$ of Figure 5.4a, the information regarding node id and class-distribution is provided for all the nodes of NLDT* in Figure 5.4b. However, the data distribution in NLDT*



(a) NLDT$_{OL}$



(b) NLDT*

Figure 5.4: NLDTs for 5-Link Manipulator problem.

is obtained by using the actual state-action data from closed-loop simulations, wherein NLDT* is used as a controller. Total 10,000 datapoints are collected in form of sequential states-action pairs from closed-loop simulation runs which are executed using NLDT*. As can be seen in the root node of NLDT* (Figure 5.4b), out of 10,000 states visited during closed-loop control, action 0 ($-\tau$ torque) was chosen by NLDT* for total 7736 states and action 2 ($+\tau$ torque) was chosen for 2264 states. In none of the states visited during closed-loop control was action 1 (no torque) chosen. This is consistent with what we have observed in the Action Vs Time plot in Figure 5.2b, wherein most of the time action 0 was executed, while there was no event where action 1 was executed. The flow of these 10,000 state-action pairs through NLDT* and their corresponding distribution

in each node of NLDT* is provided in Figure 5.4b. It can be observed that Node 5, Node 4 and Node 8 of NLDT* are never visited during closed-loop control. This implies that splits at Node 2, Node 1 and Node 6 are redundant. Thus, the part of NLDT* shown in red-box in Figure 5.4b can be pruned and the overall topology of the tree can be simplified. The pruned NLDT* will involve only one split (occurring at Node 0) and two leaf nodes: Node 1 and Node 6. However, it is to note here that we need to re-assign class-labels to the newly formed leaf nodes (i.e. Node 1 and Node 6) based on the data-distribution from closed-loop simulations. The old class-labelling for the Node 1 and Node 6 was done based on the open-loop data (Figure 5.4a). Using the new class distribution corresponding to NLDT* (Figure 5.4b), Node 1 is re-labelled with Class-2 and Node 6 with Class-0. The pruned version of NLDT* of Figure 5.4b is provided in Figure 5.5 (here nodes are re-numbered, with Node 6 of NLDT* in Figure 5.4b re-numbered to Node 2 in the pruned NLDT* as shown in Figure 5.5). The split-rule corresponding to the root-node is also shown. Interestingly, out of 10 total state-variables, only 2 are used to decide which action to execute for closed-loop control. x_1 corresponds to the angular position of the second link and x_9 variable corresponds to the angular velocity of the last joint (i.e. the joint between link-4 and link-5 where the motor is connected).

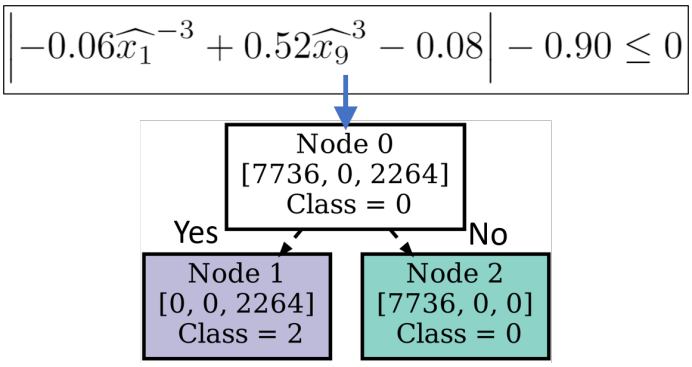


Figure 5.5: Pruned version of NLDT* (Figure 5.4b) for 5-link manipulator problem.

5.4 Conclusion

In this chapter we introduced benchmark problems to conduct scalable study and investigate and compare algorithms to efficiently conduct open-loop training. The lower level optimization

done using SQP during open-loop training reduces the overall open-loop training time by 70 times. The NLDT$_{OL}$ induced using RGA in lower level shows robust and relatively better open-loop performance. However the closed-loop performance observed after re-optimizing NLDT$_{OL}$ to NLDT* is similar in either case. This observation suggests the use of SQP in control tasks to quickly generate NLDT$_{OL}$. An extension of this work can be made for action-spaces involving more than 3 actions. This can be achieved by allowing more joints of the serial-manipulator to get actuated using motors. Also, a hybrid approach combining SQP and RGA for open-loop training can be derived to efficiently induce high performing NLDT$_{OL}$. This aspect will be particularly useful for classification problems.

CHAPTER 6

EXTENSION TO REGRESSION AND CONTINUOUS CONTROL PROBLEMS

6.1 Introduction

In this chapter, we provide a conceptual path on how to extend the idea of nonlinear decision tree to solve problems pertaining to regression and control systems involving continuous action as output. Regression problems involves several independent features which are represented with feature vector $\mathbf{x}$, and one dependent variable y. The task here is to learn the relationship between dependent and independent variables ($y = f(\mathbf{x})$) Traditionally, linear regression models and neural networks can be trained to predict the value of a dependent variable y from the input features $\mathbf{x}$. However, these regression models are inherently complex since they translate to long equations to represent the mathematical relationship between output y and input features $\mathbf{x}$. Regression trees have been popularly used to find *piece-wise* constant curves to fit the regression surface. The idea behind regression trees is to partition the feature space into sub-regions using one or more splits and then have a constant term for each such sub-region to approximate the value of the dependent variable y. CART and M5 based trees fall in this category. An improvisation to this concept is suggested in [1], where, instead of having the leaf node to represent a constant valued function as in CART, leaf nodes represent a linear or quadratic functions comprising of m features (where m is a user-specified parameter). Partitioning splits however occur on only one feature. An example of interpretable tree generated in this fashion is shown in Figure 6.1.

6.2 Interpretable AI for Regression Problems using NLDT

The interpretable AI (IAI) developed for regression in our case is represented in the form of a nonlinear decision tree (NLDT), with terminal leaf nodes representing *regression equations* and the non-terminal conditional split-nodes representing *conditional* rules. The set of conditions at split-nodes define the domain in the feature space where the regression rule at a follower terminal

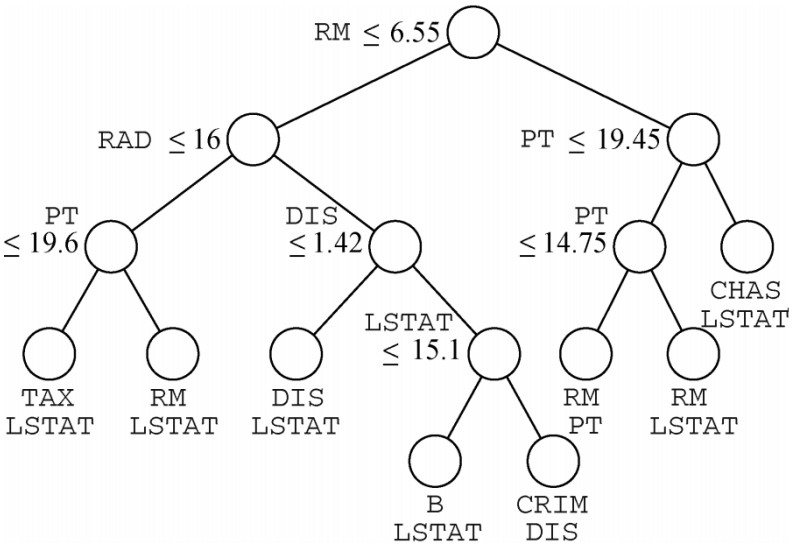


Figure 6.1: Piecewise linear regression tree with two predictors from [1]. At each leaf node, features involved in the expression of two-regressor linear model is shown. Splits use only one feature variable.

node is applicable. A conceptual illustration of the NLDT framework for regression task is provided in Figure 6.2.

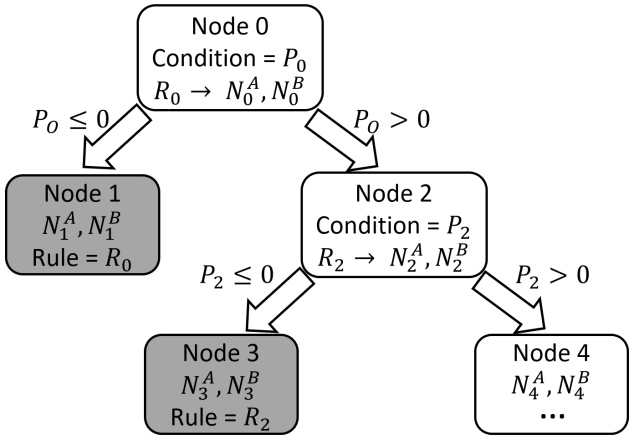


Figure 6.2: Conceptual layout of NLDT for regression task.

The rule R_i at each terminal node is a function of feature vector $\mathbf{x}$ and provides the value of the dependent variable y (i.e. $y = R_i(\mathbf{x})$). Each conditional rule P_i partitions the feature space into

two parts: $P_i(\mathbf{x}) \leq 0$ and $P_i(\mathbf{x}) > 0$. Thus mathematically, for each i-th leaf node:

$$\begin{aligned} \text{If:} \quad & P_j(\mathbf{x}) > 0, \text{ for all } j < i, \text{ and } P_i(\mathbf{x}) \leq 0, \\ \text{Rule:} \quad & y = R_i(\mathbf{x}). \end{aligned} \tag{6.1}$$

For example, for Node 1 ($i = 0$) leaf node, the rule indicates: If $P_0 \leq 0$, then $y = R_0(\mathbf{x})$. Node 3 rule ($i = 2$) indicates: If $P_0 > 0 \wedge P_2 \leq 0$, then $y = R_2(\mathbf{x})$. The nonlinear decision tree is induced by hierarchically applying a *Dual bilevel algorithm* at each conditional node. For an i-th conditional node, Stage 1 of the proposed dual bilevel algorithm focuses at deriving the regression rule $y = R_i(\mathbf{x})$. Stage 2 of the algorithm is then invoked to derive the partition rule $P_i(\mathbf{x})$ to split the data in conditional node C_i into two subsets: C_i^L and C_i^R. Points in C_i^L follows the regression rule $y = R_i(\mathbf{x})$ and have a mean squared error (MSE) value within a user defined threshold value τ_{mse}. Thus, C_i^L is declared as a terminal leaf node. A new dedicated dual bilevel algorithm is reapplied to the datapoints present in node C_i^R and process repeats. This procedure of hierarchically inducing the decision tree continues until one of the termination criteria is met. We now discuss the dual bilevel algorithm which is used to derive regression and partition rule at a given conditional node C_i.

6.3 Dual Bilevel Algorithm (DBA) for Regression

The dual bilevel algorithm is used to derive two rules at a given conditional node i: 1) regression rule $y = R_i(\mathbf{x})$ and 2) partition rule $P_i(\mathbf{x})$. The objective of deriving regression rule $R_i(\mathbf{x})$ is to fit as many datapoints present in the conditional node C_i as possible. If the regression rule $y = R_i(\mathbf{x})$ is able to fit all datapoints present in node C_i within an acceptable limit on mean squared error, the dual bilevel algorithm is terminated and the node C_i is declared as the leaf node. However, if regression rule $y = R_i(\mathbf{x})$ is not able to fit all datapoints in C_i, the second stage of dual bilevel algorithm is invoked to derive the partition rule $P_i(\mathbf{x})$. As mentioned before, the partition rule $P_i(\mathbf{x})$ partitions the data in node C_i into two subsets

- Left child node (C_i^L) (for which $P_i(\mathbf{x}) \leq 0$), and

- Right child node (C_i^R) (for which $P_i(\mathbf{x}) > 0$),

such that datapoints in subset C_i^L have their MSE value w.r.t to regression curve $y = R_i(\mathbf{x})$ within a pre-specified threshold value τ_{mse}. Thus, for a new test datapoint $\mathbf{x}$, conditions listed in Eq. 6.1 are first checked before applying regression rule $y = R_i(\mathbf{x})$ to estimate the value of the dependent variable y. A high level pseudo-code for the dual bilevel algorithm is provided in Algorithm 8.

Algorithm 8: Pseudo-code of dual bilevel algorithm (DBA)

Input : Data in Conditional node C_i: N datapoints with d independent variables $((x_1, x_2, \ldots, x_d) = \mathbf{x})$ and one dependent variable y.
Output Regression Rule $y = R_i(\mathbf{x})$ and partition rule $P_i(\mathbf{x})$.
:

```
// Bilevel Algorithm to derive R_i(x)
```

Step 1 *derive regression rule:* $y = R_i(\mathbf{x})$;
if $MSE(Data, R_i(\mathbf{x})) \leq \tau_{mse}$ **then**
  terminate;
else
  Derive Partition Rule $P_i(\mathbf{x})$;
end

We will next discuss the bilevel algorithm which is used to derive the regression rule $y = R_i(\mathbf{x})$.

6.3.1 Data Normalization

Before training and inducing the NLDT for regression, we normalize the data. Each feature x_i and dependent variable y is normalized using the following equation:

$$x_{norm} = \frac{x - \mu_x}{2\sigma_x}, \tag{6.2}$$

where x_{norm} is the normalized value of the variable x, μ_x and σ_x are mean and standard deviation of variable x, respectively.

6.3.2 Bilevel Regression Algorithm to Obtain $R(\mathbf{x})$

The regression rule to be evolved is expressed as a weighted sum of power-laws as given in equation below:

$$R_i(\mathbf{x}) = \frac{w_1 B_1 + w_2 B_2 + \cdots + w_p B_p + \theta_0}{\theta_1}, \tag{6.3}$$

where B_i's are power-laws of type $B_i = \prod_{j=1}^{d} x_j^{b_{ij}}$. Like in the standard classification task discussed in Chapter 3, the genome representing the *template* of the regression rule is represented with a block matrix $\mathbf{B}$ as shown below

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1d} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & b_{p3} & \dots & b_{pd} \end{bmatrix}. \tag{6.4}$$

The upper level of the bilevel algorithm conducts the search in the discrete space of exponents b_{ij}. The lower level of the bilevel algorithm searches the coefficients w_i corresponding to power-laws B_i and the bias terms θ_0 and θ_1. The upper level algorithm is similar to the upper level algorithm developed for classification problems (see section 3.3.2), however the objective and constraint evaluation is different as shown in Eq. 6.5. For the lower level, we use the sequential quadratic program (SQP) to estimate the values of w_i's and θ_i's. The optimization formulation corresponding to the problem of estimating the regression function $y = R_i(\mathbf{x})$ is given below:

$$\begin{aligned} \text{Minimize} \quad & F_U(\mathbf{B}, \mathbf{w}^*, \boldsymbol{\Theta}^*) = F_L(\mathbf{w}^*, \boldsymbol{\Theta}^*)|_{\mathbf{B}}, \\ \text{Subject to} \quad & (\mathbf{w}^*, \boldsymbol{\Theta}^*) \in \operatorname{argmin} \{F_L(\mathbf{w}, \boldsymbol{\Theta})|_{\mathbf{B}}\}, \\ & -1 \le w_i \le 1, \ \forall i, \ \theta_0 \in [-1, 1], \ \theta_1 \in (0, 1], \\ & b_{ij} \in Z, \end{aligned} \tag{6.5}$$

where for our study here we choose $Z = \{-3, -2, -1, 0, 1, 2, 3\}$. The allowable powers indicated by set Z controls the maximum complexity achievable by our procedure. The fitness of the upper

level individual corresponds to the fitness of the *optimal* solution of the corresponding lower level problem. We will next discuss how the lower level fitness function is designed to facilitate the regression fit.

6.3.2.1 Lower Level Regression Optimization

Since we are targeting at evolving a simple rule, it is highly likely that one *simple* rule won't fit the entire dataset. Figure 6.3 illustrates one such situation involving two independent variables and one dependent variable. Here, datapoints are scattered across three disjoint islands. Most of

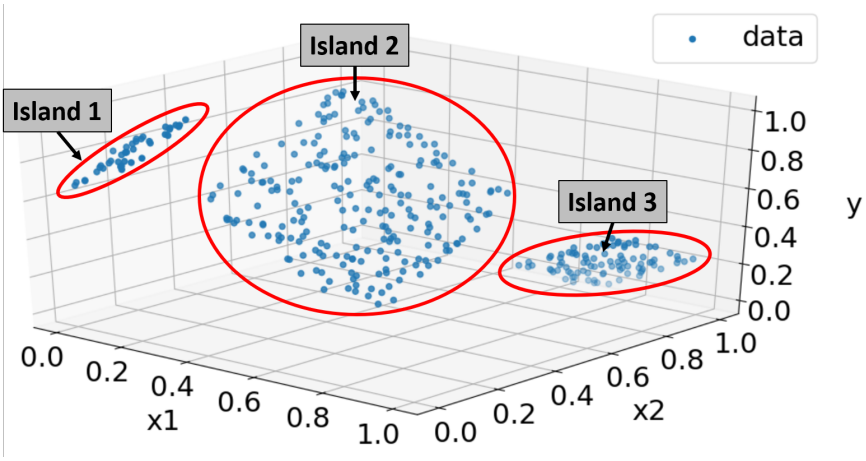


Figure 6.3: Three Island Regression Problem.

them belong to island 2 while island 1 has the least number of datapoints. There might exist one complicated regression surface which passes through all the datapoints. However, since we are interested in having simple rule, one of the simple solution is to have three simple linear rules, one for each island. Furthermore, in our overall algorithm, we would be interested to determine the rule corresponding to island 2 first since it has majority number of points. The fitness function F_L for the lower-level optimization is formulated to capture these aspects. The algorithm to compute F_L is provided in Algorithm 9.

We will discuss the motivation behind different segments of Algorithm 9 here and explain the idea with the help of Figure 6.4.

Algorithm 9: Algorithm to compute the lower level fitness F_L

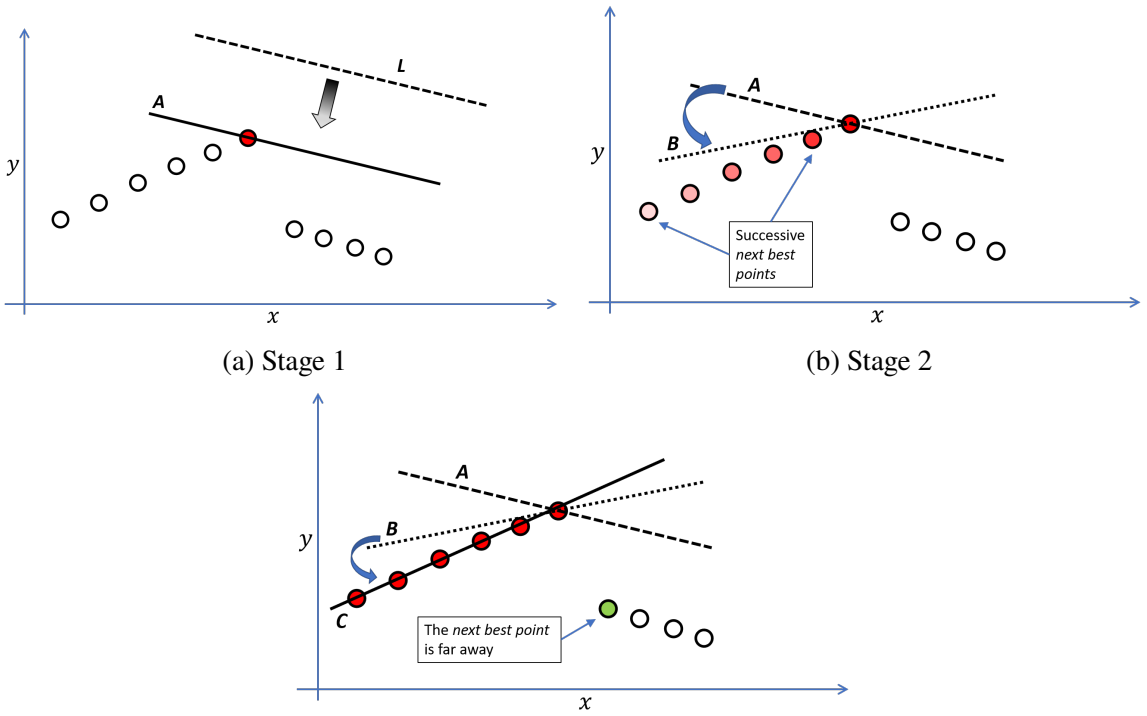
Input : Dataset **D** in B-space of size $N \times p$ and array Y of length N comprising of values of dependent variable y, weight vector **w**, bias valus θ_0 and θ_1.

Output : Lower level fitness value F_L

```
// predict the value of y
Y' = PredictY(D, w, θ0, θ1);
// Compute the absolute difference between predicted and actual Y values
Y_error = |Y − Y'|;
// Get a Boolean array of good points
G = Y_error ≤ τ_error;
// Compute the total number of good points
N_good = Σ_i^N G[i];
if N_good == 0 then
    // No good point found
    F_L = ComputeRMSE(Y', Y); // Root Mean Squared Error
else
    if N_good < N − 2 then
        // Sorted array of error in ascending order
        S_error = Sort(Y_diff);
        /* Get the error value of the best point from set of bad points G'. */
        d' = S_error[N_good + 1];
        if d' < 0.5 then
            r = 1/(N_good+1) Σ_{i=1}^{N_good+1} S_error[i]^2;
        else
            r = ComputeRMSE(Y[G], Y'[G]);
        end
    else
        r = ComputeRMSE(Y[G], Y'[G]);
    end
    F_L = r − N_good
end
```

The value of F_L is computed based on which *stage* the regression surface is relative to the training data.

Stage 1 Here, the regressed surface is far off from the location of training datapoints as shown in Figure 6.4a. This implies that no points are within a τ_{error} distance from the regressed



(a) Stage 1

(b) Stage 2

(c) Stage 3

Figure 6.4: Computation of Lower Level Objective function F_L based on Algorithm 9.

surface, i.e. $\forall \mathbf{x} \in D, |y(\mathbf{x}) - y'(\mathbf{x})| > \tau_{error}$, where y and y' are actual and regressed values of $\mathbf{x}$ respectively. The value of parameter τ_{error} is user specified. Under this scenario, the objective F_L is evaluated by computing *mean squared error* (MSE) between actual and predicted values. Minimization of MSE in *Stage 1* brings the regression surface closer to the location of training dataset. This is shown pictorially in Fig. 6.4a. This segment is represented by lines 5-6 in Algorithm 9. Eventually, the regressed surface/curve will come in the vicinity of one (or more) datapoints (shown by *red* in Fig. 6.4a) and will result into $N_{good} \geq 0$, where N_{good} is the number of datapoints for which the difference between predicted and actual y value is less than τ_{error}.

Stage 2 During Stage 2, we already have atleast one point with $|y - y'| \leq \tau_{error}$ (i.e. $N_{good} \geq 1$). In Stage 2, we design the objective function F_L such that its minimization will ensure maximization of number of *good* points (i.e. points with $|y - y'| \leq \tau_{error}$). One such way to enforce this is to set $F_L = -N_{good}$. However, setting F_L this way resembles a discontinuous

many-to-one mapping. This will create *local flat regions* (similar to the situation discussed in Figure 3.6) thereby making it difficult for an optimization solver to navigate the search space of $\mathbf{w}$ and $\mathbf{\Theta}$. We thus do the following modification to avoid creation of *flat zones* in the fitness landscape of F_L:

Step 1 Determine the *closest* point $\mathbf{x}'$ in dataset $\mathbf{D}$ to the regressed surface such that $|y'(\mathbf{x}') - y(\mathbf{x}')| > \tau_{error}$ and compute its error as $d' = |y'(\mathbf{x}') - y(\mathbf{x}')|$ (line 9-10 in Algorithm 9). If $d' < 0.5$, go to Step 2, else go to Stage 3

Step 2 Since $d' < 0.5$, $\mathbf{x}'$ is potentially a *next good* point. Thus, we append datapoint $\mathbf{x}'$ to the list of good points and compute the MSE on this *modified* set of good points. Let the MSE value of this set (good points + *potentially* good point) be denoted with r. F_L is then computed as

$$F_L = r - N_{good}$$

as shown in line 12 and line 19 in Algorithm 9. Note that at first, the term N_{good} will remain constant and only r will get minimized while minimizing F_L. This will result in *tilting* of the regressed surface as shown in Figure 6.4b (where regressed surface A is tilted to B). Eventually, minimization of F_L this way will successively cover more points (shown by *shaded red* colors in Figure 6.4b) and will bring the regression surface to location C (shown in Figure 6.4c).

Stage 3 Here, the *next closest point* with $d' > \tau_{error}$ is far away from the regression surface (marked with *green* in Figure 6.4c). Hence, F_L will be computed as

$$F_L = r - N_{good},$$

where r is the mean squared error of *good* points (i.e. all those points for which $|y(\mathbf{x}) - y'(\mathbf{x})| \leq \tau_{error}$). In the example shown in Figure 6.4, values assumed by F_L during Stage 3 will be certainly less than those in Stage 2 since N_{good} term of F_L in

Stage 3 will be more by atleast 1 unit than the N_{good} term in Stage 2. Also, N_{good} term in Stage 3 will remain constant since the next possible good point (green colored in Figure 6.4c) is located very far away. Thus, minimization of F_L will imply minimization of r. This will steer to re-position the line B (for which all *red* points of Figure 6.4c were within τ_{error} distance) to line C.

Based on the above explanation and illustrations shown in Figure 6.4, in a particular lower level optimization run using a *point based* classical optimization approach, the values of F_L as observed in Stage 1 will be more than those observed in Stage 2 and values observed in Stage 2 will be more than those observed in Stage 3. This would be true in general, but will also depend on several other factors such as noise in the dataset, value of τ_{error} and relative locations of datapoints in training set.

Eventually, the regression curve will fit certain segment of the training data (represented by C in Figure 6.4c). In our experiments, we set $\tau_{error} = 0.03$. Computing F_L using the procedure discussed above removes the bottleneck of having *flat* fitness landscape. This allows us to use point based classical optimization algorithms to quickly determine optimal values of $\mathbf{w}$ and $\mathbf{\Theta}$. Having a *fast* and *reliable* lower level optimization is really important for the efficient design of the overall bilevel algorithm to estimate the regression function $R(\mathbf{x})$. Next we discuss the upper level optimization which is used to estimate the structure and template of the regression function $R(\mathbf{x})$.

6.3.2.2 Upper Level Regression Optimization

Fitness of the optimal solution of lower level optimization becomes the fitness of the upper level optimization as shown in Eq. 6.5. Since genetic operators in the upper level are designed to incrementally complexify the expression of the power-law rule (Eq. 6.3), starting from a very basic rule involving only one variable appearance, we observe a natural preference and bias for simpler rules over complex rules. Output from one run of bilevel regression algorithm is shown in Figure 6.5. It can be observed that the algorithm is able to successfully fetch us with a linear

regression rule $R_i(\mathbf{x}) = 0.91x_1 - 0.90x_0 + 0.08$ to fit the island comprising of maximum number of datapoints (Figure 6.5).

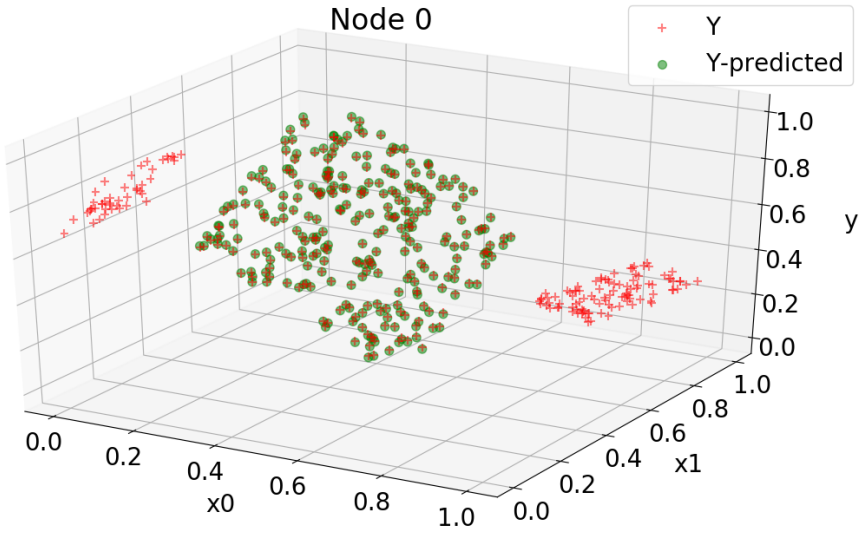


Figure 6.5: Regression algorithm is applied on all datapoints. The obtained regression rule $R_0(\mathbf{x}) = 0.91x_1 - 0.90x_0 + 0.08$ is able to fit the subset of datapoints which are represented by *Y-predicted* (in green circles). Partition rule $P_0(\mathbf{x})$ will be now derived to identify the domain in x-space (i.e. $P_0(\mathbf{x}) \leq 0$)) where this regression rule is applicable.

6.3.3 Bilevel Partition Algorithm (BPA) to Obtain $P(\mathbf{x})$

The regression rule ($y' = R_i(\mathbf{x})$) derived in the previous section fits certain portion of the dataset in node C_i (as an illustration, this is highlighted with green circles in Figure 6.5). The bilevel partition algorithm is now employed to derive a partitioning boundary $P_i(\mathbf{x}) = 0$ which separates the dataset in C_i into two subsets: C_i^L (for which $P_i(\mathbf{x}) \leq 0$) and C_i^R (for which $P_i(\mathbf{x}) > 0$). The fraction of datapoins in C_i^L have their mean squared error w.r.t. the regression curve $R_i(\mathbf{x})$ within a user specified threshold value τ_{mse}. Hence, for these datapoints, the depended variable y can be predicted using the regression function $R_i(\mathbf{x})$. The partition rule $P_i(\mathbf{x})$ is derived to maximize the number of datapoints in C_i^L (for which $P_i(\mathbf{x}) \leq 0$) while keeping their mean squared error value (where the error is measured between the actual y value and the predicted y' value) within τ_{mse}.

The optimization formulation to derive the partition rule $P_i(\mathbf{x})$ is organized in the bilevel structure as shown below

$$
\begin{aligned}
\text{Minimize} \quad & F_U(\mathbf{B}, m, \mathbf{w}^*, \boldsymbol{\Theta}^*) = F_L(\mathbf{w}^*, \boldsymbol{\Theta}^*)|_{(\mathbf{B},m)}, \\
\text{Subject to} \quad & g_U(\mathbf{B}, m, \mathbf{w}^*, \boldsymbol{\Theta}^*) = g_L(\mathbf{w}^*, \boldsymbol{\Theta}^*)|_{(\mathbf{B},m)} \leq 0, \\
& (\mathbf{w}^*, \boldsymbol{\Theta}^*) \in \text{ argmin} \left\{ F_L(\mathbf{w}, \boldsymbol{\Theta}) \middle| g_L(\mathbf{w}, \boldsymbol{\Theta}) \leq 0 \right\} \Big|_{(\mathbf{B},m)}, \\
& -1 \leq w_i \leq 1, \ \forall i, \ \boldsymbol{\Theta} \in [-1, 1]^{m+1}, \\
& m \in \{0, 1\}, b_{ij} \in Z.
\end{aligned}
\tag{6.6}
$$

The overall bilevel algorithm is identical to the one developed for the classification problem to derive split-rules for a classification task (as explained in Section 3.3), with the only change in the objective and constraint functions. The partition rule $P_i(\mathbf{x})$ (where i is the node number (see Figure 6.2)) being derived is of the following type

$$
P_i(\mathbf{x}, \mathbf{w}, \boldsymbol{\Theta}, \mathbf{B}) = \begin{cases} \theta_1 + w_1 B_1 + \ldots + w_p B_p, & \text{if } m = 0, \\ \left| \theta_1 + w_1 B_1 + \ldots + w_p B_p \right| - \left| \theta_2 \right|, & \text{if } m = 1. \end{cases}
\tag{6.7}
$$

where all symbols carry similar meanings to those discussed in the equation of split-rule for classification problem (Eq. 3.2). $\mathbf{x}$ is the input feature vector, B_i is the power-law of type $B_i(\mathbf{x}) = \prod_{j=1}^{d} x_j^{b_{ij}}$, w_i's are the coefficients and θ_i's are the biases.

Since both objective and constraint value for upper level optimization are borrowed from the optimal solution of the lower level optimization, we discuss the optimization formulation for lower level optimization in the next section.

6.3.3.1 Lower Level Partition Optimization

The lower level optimization estimates optimal values of weights $\mathbf{w}$ and biases $\boldsymbol{\Theta}$ which minimize the following objective function

$$
\begin{aligned}
\text{Min.} \quad & F_L(\mathbf{w}, \boldsymbol{\Theta})\Big|_{(\mathbf{B},m)} = -N_{left} \\
\text{s.t.} \quad & g_L(\mathbf{w}, \boldsymbol{\Theta})\Big|_{(\mathbf{B},m)} = MSE_{left} - \tau_{mse} \leq 0 \\
& -1 \leq w_i \leq 1, \ \forall i, \ \boldsymbol{\Theta} \in [-1, 1]^{m+1}
\end{aligned}
\tag{6.8}
$$

Here, N_{left} indicates the number of points in node i for which $P_i(\mathbf{x})|_{(\mathbf{w},\mathbf{\Theta},\mathbf{B},m)} \leq 0$ (Eq. 6.7). Thus, this objective function tends to maximize the number of points going to the left node C_i^L after partition. MSE_{left} is computed by measuring the mean squared error of the actual y value of datapoints in the left node C_i^L with the predicted y' values obtained using the regression function $y' = R_i(\mathbf{x})$ derived in the previous section. The constraint satisfaction $\Big(g_L(\mathbf{w}, \mathbf{\Theta})|_{(\mathbf{B},m)} = MSE_{left} - \tau_{mse} \leq 0\Big)$ ensures that y value of all datapoints coming to the left node C_i^L after the split (i.e. $P_i(\mathbf{x}) \leq 0$) can be reliably predicted by using the regression function $y' = R_i(\mathbf{x})$. In our experiments, we set the value of τ_{mse} parameter to τ_{error}^2 (where $\tau_{error} = 0.03$). It is to note here that the regression rule $y = R_i(\mathbf{x})$ is not applicable to points belonging to subset C_i^R (i.e. $P_i(\mathbf{x}) > 0$). Hence, the right node which contains datapoints from subset C_i^R becomes the new *non-terminal* node and a dedicated *dual bilevel algorithm* is applied to it to derive its corresponding regression and partition rules ($R_2(\mathbf{x})$ and $P_2(\mathbf{x})$ in Figure 6.2). This process is repeated until maximum depth of tree is reached or the regression rule is able to fit all datapoints belonging to a non-terminal node (in which case it is declared as a terminal leaf node).

6.3.4 Results on a Customized Benchmark Problem

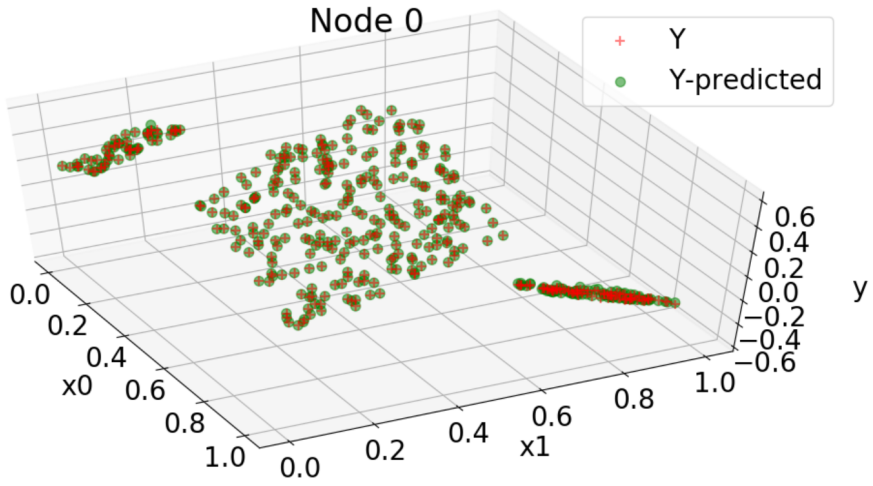
In this section, we show results obtained on different versions of customized benchmark *three island* problem (Figure 6.3). In the first experiment, we generate a pure dataset using the following set of equations for islands 1, 2 and 3:

$$\begin{aligned}
\textbf{Island 1:} \quad & Y_1 = X_0 + X_1 + 0.2, \quad [X_0, X_1] \in [0, 0.2]^2 \\
\textbf{Island 2:} \quad & Y_2 = -X_0 + X_1 + 0.1, \quad [X_0, X_1] \in [0.2, 0.7]^2 \\
\textbf{Island 3:} \quad & Y_3 = X_0 - X_1 - 0.3, \quad [X_0, X_1] \in [0.7, 1]^2 \\
\textbf{Concatenate:} \quad & Y_{pure} = [Y_1; Y_2; Y_3]
\end{aligned} \tag{6.9}$$

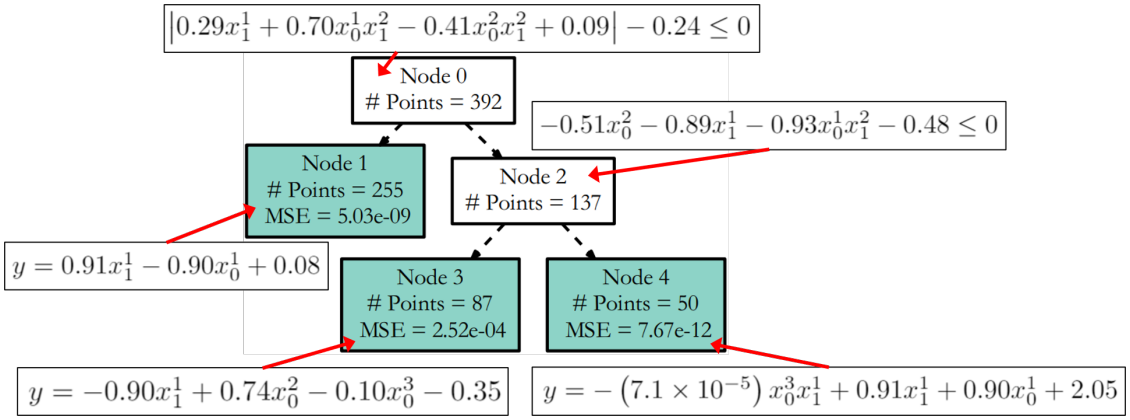
In the second experiment, we add noise to the pure dataset of Eq. 6.9 as shown below

$$Y_{noise}^{(i)} = Y_{pure}^{(i)} + \epsilon, \quad i = 1, 2, \ldots N, \qquad \epsilon \in [-0.1, 0.1] \tag{6.10}$$

Results obtained on pure and noisy three island datasets are provided in Figures 6.6 and 6.7 respectively. Very small MSE values for each of the three leaf nodes indicate the efficacy of our proposed Dual bilevel optimization algorithm to find regression and partition rules.



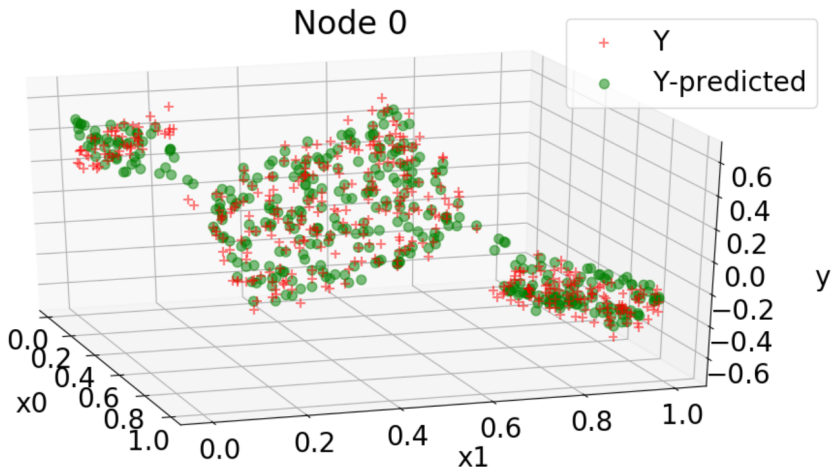
(a)



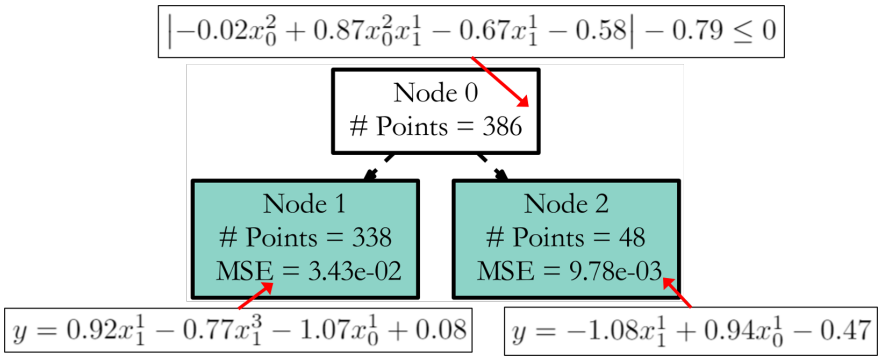
(b)

Figure 6.6: Result on the Pure Three Island Dataset. (a) Visualization of result. (b) Interpretable AI representation using NLDT. Normalization constants are: $\mathbf{X_{mean}} = [0.49, 0.49]$, $\mathbf{X_{std}} = [0.25, 0.25]$, $\mathbf{Y_{mean}} = 0.05$ and $\mathbf{Y_{std}} = 0.28$.

As can be seen from results shown in Figure 6.6 and Figure 6.7, the proposed approach of deriving *piece-wise non-linear* regression equations fetches us with a relatively simple and interpretable regressor, which otherwise would have assumed a complicated form with other regression models like artificial neural networks or regression CART. The rules evolved are simplistic in structure and



(a)



(b)

Figure 6.7: Result on the Noisy Three Island Dataset. (a) Visualization of result. (b) Interpretable AI representation using NLDT. Normalization constants are: $\mathbf{X_{mean}} = [0.52, 0.52]$, $\mathbf{X_{std}} = [0.28, 0.27]$, $\mathbf{Y_{mean}} = 0.03$ and $\mathbf{Y_{std}} = 0.29$.

the induced NLDT is also topologically simple.

Currently, we are required to properly set the parameter τ_{error} ($\tau_{mse} = \tau_{error}^2$) to get optimal performance. One of the possible way to efficiently handle this is to re-formulate the upper-level problem as multi-objective and search for the *knee* in the bi-objective pareto front of MSE Vs N_L (where N_L is the number of points in left node) and use that solution to conduct the partition.

In the next section, we will apply the proposed *dual bilevel* algorithm to arrive at a relatively interpretable policy for a control problem involving continuous action.

6.4 Interpretable AI for Control Problem with Continuous Action Space

In this section, we implement the dual bilevel algorithm we developed in the previous section to explain control logic of an ANN (artificial neural network) agent which is used to control an object using an action value from a continuous real domain. The problem considered here is borrowed from [87] and is pictorially shown in Figure 6.8.

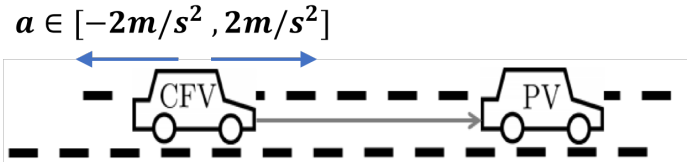


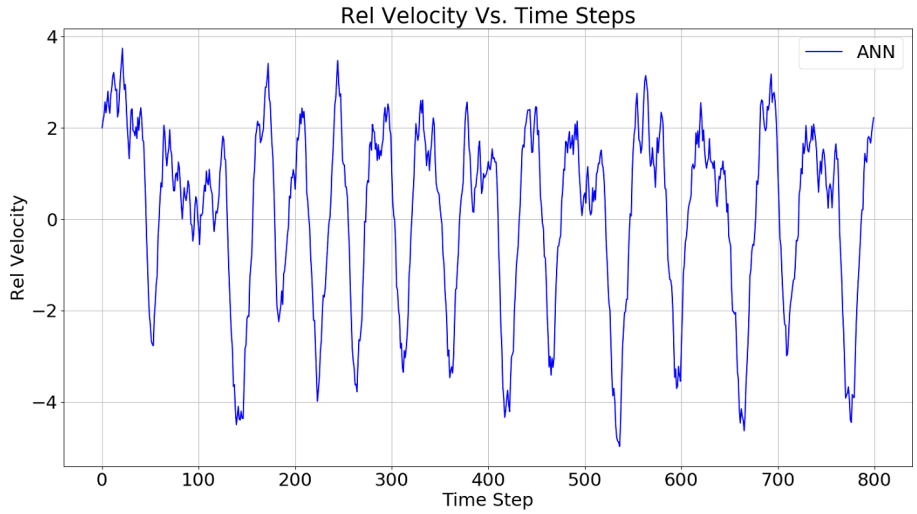
Figure 6.8: A Car-following problem with continuous acceleration. The rear car is controlled by an AI while the car in the front moves with a random acceleration profile. This problem is similar to the problem shown in Figure 4.8b with the only difference being that the cars can now assume a value of acceleration in range $-2m/s^2$ to $2m/s^2$ (unlike in the previous case where the rear car could only have an acceleration from pre-specified discrete values).

Here, the car in the rear is autonomously controlled using an AI. The car in the front moves with a random acceleration profile. The state of the rear car at a given time t is determined by three quantities (the *state* is denoted by a real-parameter vector **x**):

1. Its relative distance from the front car ($x_0 = d_{front} - d_{rear}$),
2. its relative velocity with respect to the front car ($x_1 = v_{front} - v_{rear}$) and
3. its acceleration at time $t - 1$ ($x_2 = a(t-1)$).

The control task requires to have a safe distance d_{safe} between two cars, while simultaneously ensuring a comfortable and smooth ride. The ANN agent is trained using the DDPG algorithm [62]. For a given time-step t, the ANN agent takes the state **x** of the rear car as an input and outputs the acceleration value y (i.e. $a(t)$) in the continuous range from -$2m/s^2$ to $2m/s^2$. Figure 6.9 shows a sample run where the trained ANN agent is used to control the rear car while the car in the front operates with a random acceleration.

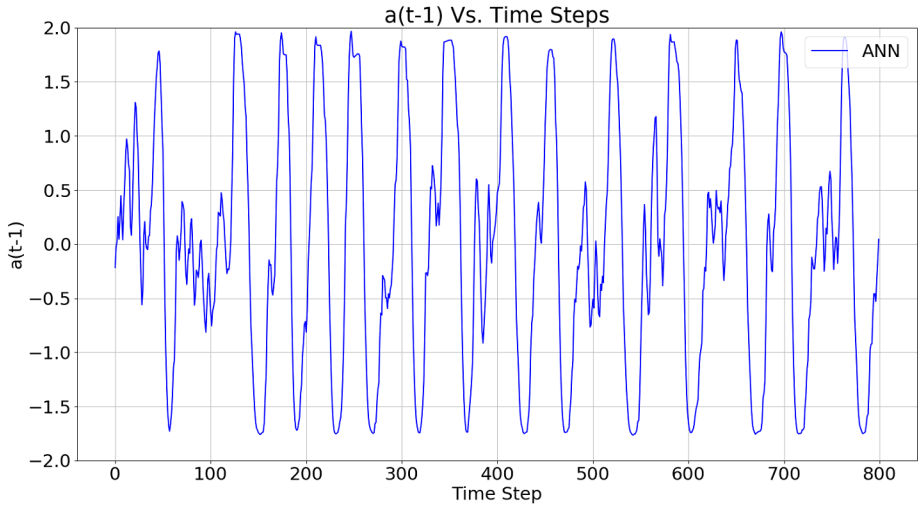
(a)



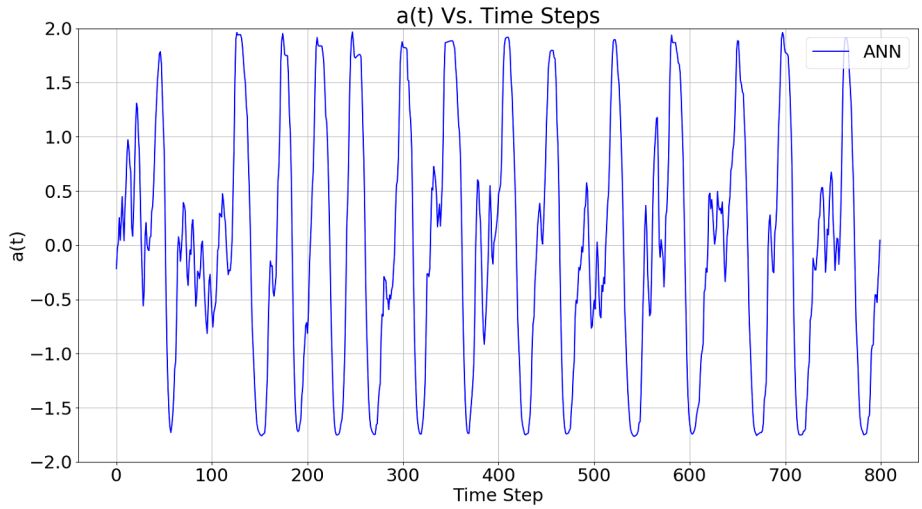
(b)

Figure 6.9: ANN Output for the continuous car-following problem described in Figure 6.8. Figures (a), (b) and (c) are the plots of different state variables w.r.t to the time-step. The output of the ANN is shown in Figure (d).

Figure 6.9 (cont'd)



(c)



(d)

To explain the performance of the ANN, we first collect the data comprising of series of states and corresponding action value as determined by the ANN. The tabulated data is then used to train and induce the NLDT. This problem of training and deriving NLDT from the ANN generated dataset translates to a regression problem of predicting the value of a continuous dependent variable given a set of continuous input features (or state variables in our case). The challenge here is to obtain the NLDT regressor with simple rules so that it can act as a *logic-translator* to explain the inner-logic of the ANN, RL or any more compelex CNN/DNN used earlier to solve the problem.

6.4.1 NLDT Representation and Training

As mentioned in the last section, the problem of developing interpretable AI for a continuous control task involving one action translates to a regression rule finding problem. We thus adopt the dual bilevel approach we discussed in section 6.3 to derive the NLDT for this regression problem. For the control task, the regression rules represent different *control logic* and the partition rules represent different *conditions* (or scenario) under which a given control logic is applicable. An illustrative sketch of the same is provided in Figure 6.10.

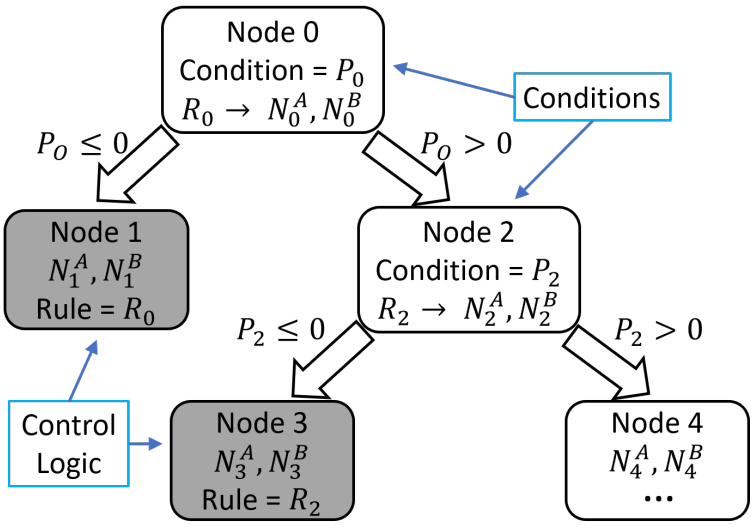


Figure 6.10: NLDT Agent representation for continuous control task involving one action.

The conditional rules $P_i(\mathbf{x})$ and control logic $R_i(\mathbf{x})$ are expressed as weighed sum of power-laws as mentioned in Eq. 6.3 and 6.7 respectively. The induction and training of the non-linear decision tree (NLDT) of Figure 6.10 is done using the methodology discussed in the previous section in Section 6.3.

6.4.2 Data generation

To derive the NLDT, the data is first generated using the trained artificial neural network (ANN) controller. For our task, the ANN controller takes state variables as input and outputs the value of the acceleration. In our experiments, we generate 5,000 datapoints to populate the training

data. Once the training data is generated, the dataset is normalized using the method discussed in Section 6.3.1. The normalized data is then used to train the NLDT. As mentioned before, each regression rule and partition rule is derived using the proposed dual bilevel algorithm (Section 6.3).

6.4.3 Results on Car-Following Problem

In this section we discuss some results obtained on the 1D car following problem involving continuous action space (Figure 6.8). As mentioned before, the car in the front follows a random acceleration profile. The rear car is controlled using a trained ANN controller. The NLDT takes the same state information as the ANN counterpart and predicts the value of acceleration. It is desired to have the predicted acceleration value of NLDT to be as close as possible to the corresponding ANN output for a given input state. Plots from different simulation runs are shown in Figure 6.11 to provide the visualization of how closely the NLDT is able to mimic the parent ANN controller's output.

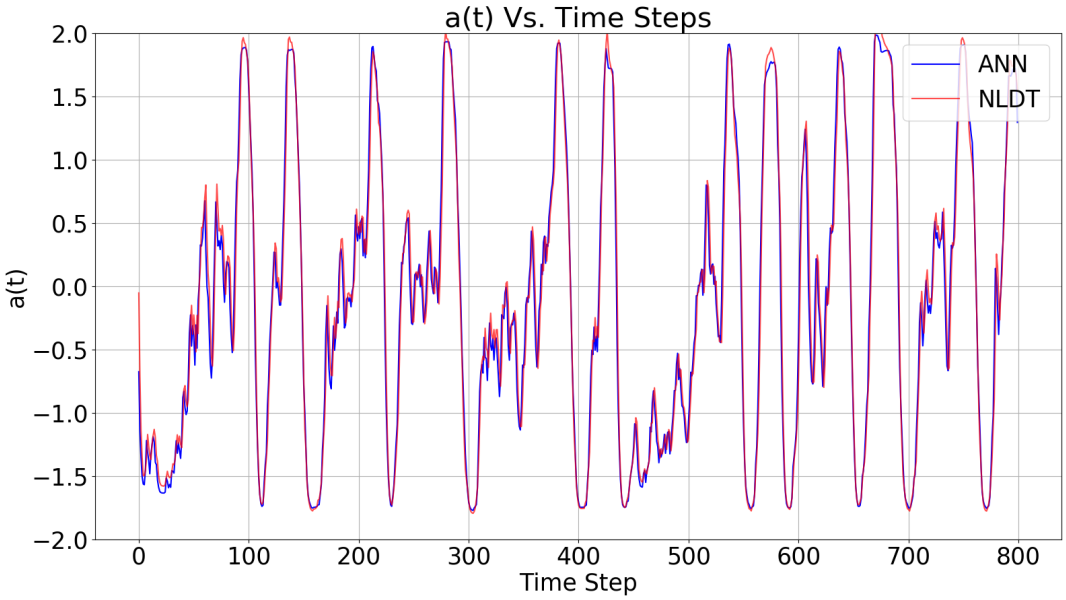
The performance seen in Figure 6.11 is obtained by the NLDT agent shown in Figure 6.12.

From the results shown in Figure 6.11, it can be concluded that our proposed algorithm is able to induce a relatively interpretable NLDT (Figure 6.12) which is able to match the predictions of the complicated ANN controller. It is important to highlight all components of the NLDT (Figure 6.12) which is obtained by our procedure:

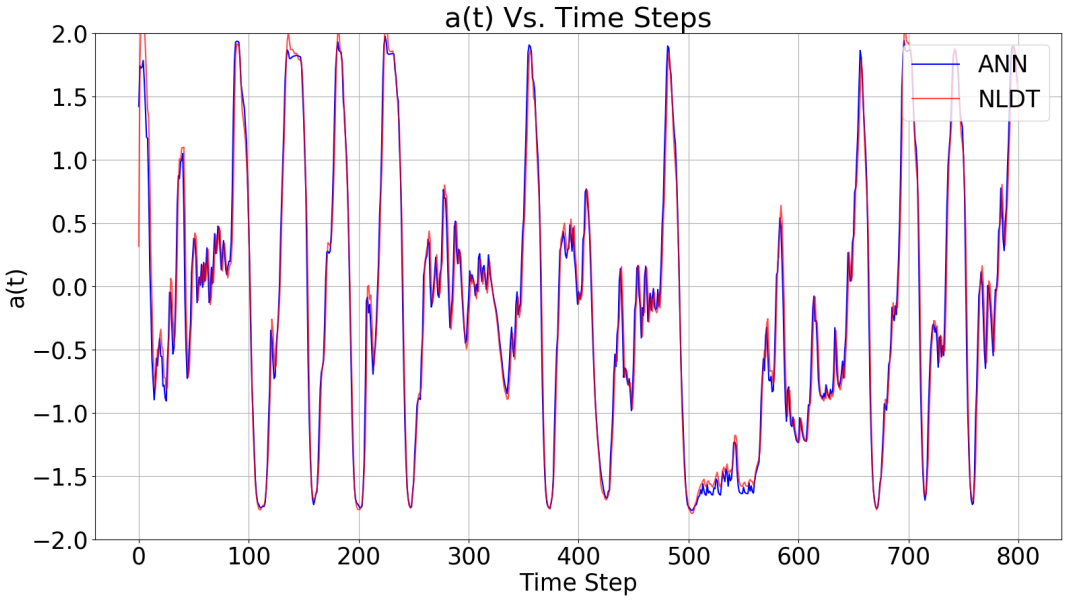
1. A two-depth decision tree is adequate to explain control strategies of the ANN with a small MSE error. Our dual bilevel approach is able to evolve this simple tree.

2. The tree involves one non-linear partition rule ($P_0(\mathbf{x})$) at the root node, and two non-linear regression rules ($y = R_i(\mathbf{x})$) at left leaf node ($i = 1$, applicable for $P_0(\mathbf{x}) \leq 0$) and right leaf node ($i = 2$, applicable for $P_0(\mathbf{x}) > 0$).

3. The structures of the regression and partition rules are evolved by our dual bilevel optimizer involving simple expressions.



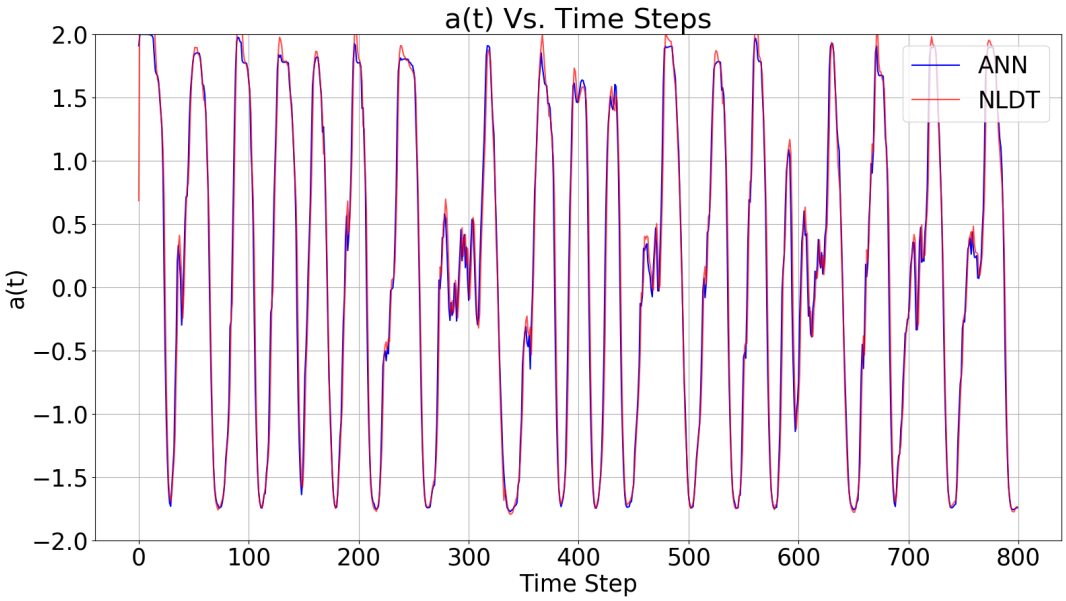
(a) $v_{rel}(t = 0) = 0$



(b) $v_{rel}(t = 0) = 5$

Figure 6.11: Plots from different simulation runs with different initial relative velocity (v_{rel}) between cars. The NLDT's acceleration output (red line) matches with the ANN's acceleration output (blue line). NLDT agent behaviour is almost the same as that of ANN and can thus be used to explain the behaviour of ANN.

Figure 6.11 (cont'd)



(c) $v_{rel}(t = 0) = 10$

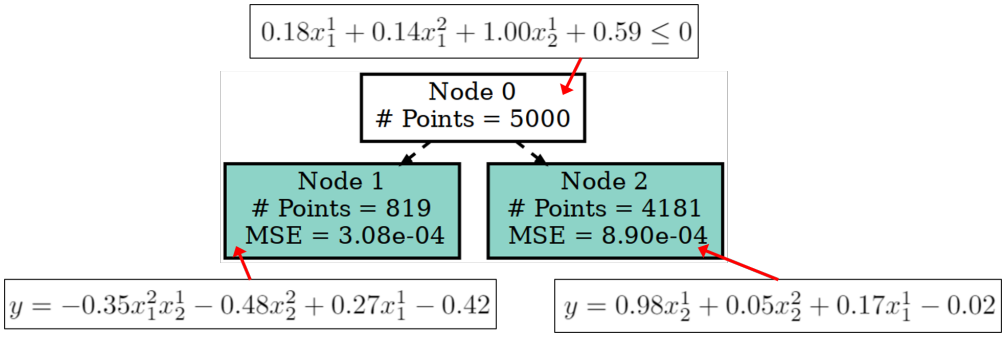


Figure 6.12: NLDT for car following problem with continuous action space.

While the original ANN can control the rear car well, it certainly cannot explain the reasons behind choosing the action for each combination of three input states. Above partition and regression rules paves the way of arriving at a relatively humanly-interpretable explanation to the data derived from the ANN controller. Notice that a regular decision tree (DT) is incapable of coming up with such a nonlinear combinations of variables. This is true even for the generalized linear models (GLMs) and generalized additive models (GAMs). Due to hurdles of optimizing non-linear optimization problems with guaranteed optimality, nonlinear decision trees were not studied in the

past. With the advancements of efficient evolutionary (and bilevel) optimization methods, we are able to open up this avenue to come up with as few as possible and as interpretable as possible nonlinear decision rules. We believe that this strategy can be extended to more complex control problems.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this work, we primarily focused on inducing an intrinsically interpretable AI which is relatively interpretable and simple than ANN, CART and SVM. The AI in our case is modelled using a nonlinear decision tree (NLDT) wherein, unlike traditional decision trees, the conditional nodes involve non-linear rules on input features. The non-linear rules at conditional nodes are expressed as weighted sum of power-laws with exponents assuming values from a discrete set while the coefficients and biases take real values between -1 to 1. The search for the non-linear rules is carried out using a custom-designed bilevel approach. The upper level spans the discrete search space corresponding to exponents while the lower level conducts a search on continuous variables to optimize the split criteria. The NLDT is grown through a recursive splitting procedure, and for each new conditional node, the non-linear rule is efficiently derived using a bilevel algorithm. This implies that there is no need to pre-supply the *topology* of the NLDT since it gets organically determined. The NLDTs produced are simple in topology as compared to traditional CART trees and have much simpler rules than ANN or SVM counterparts at each conditional node, thereby making the overall AI relatively more interpretable. A comparison with classical classification algorithms on some standard and custom-designed benchmarks reveals the superiority of the proposed NLDT algorithm in efficiently deriving simple yet high-performing classifiers.

This idea is extended to develop relatively interpretable controllers for control problems involving discrete actions. The open-loop training determines the structure of the NLDT and non-linear rules, which are then re-optimized using an evolutionary-algorithm-based closed-loop training procedure to enhance the closed-loop control performance of the NLDT. The objective evaluation for closed-loop training is computationally expensive; however, a speed-up to the overall evolutionary closed-loop training algorithm is possible by distributing the evaluation of population individuals to different computer cores. The proposed algorithm is able to induce an NLDT* controller which is relatively simple than DNN controller and has better closed-loop control than the parent black-box

AI controller. One of our key observations in this study was it was not required to have an NLDT$_{OL}$ with 100% open-loop accuracy to achieve a competent closed-loop performance. We extended our study of inducing relatively interpretable controllers in Chapter 5 to conduct a scale-up study and proposed a benchmark-problem. The open-loop training is made considerably fast by replacing the lower-level RGA algorithm with a classical SQP algorithm. This compromises the open-loop accuracy, however the re-optimization through the follow-up closed-loop training successfully readjusts weights and coefficients of NLDT$_{OL}$. The resulting NLDT* has a better closed-loop performance than the original black-box AI. The NLDT* is then pruned and simplified by eliminating nodes which are rarely visited during closed-loop control runs.

A conceptual layout on the possible extension of NLDT idea to solve regression and control problems with continuous action space is demonstrated in Chapter 6. Here, a dual-bilevel algorithm to proposed to systematically derive regression functions $R(\mathbf{x})$ and partition functions $P(\mathbf{x})$. The proposed approach is first tested on a customized three-island regression problem involving three separate disjoint linear regression surfaces. Next, the dual-bilevel algorithm of inducing NLDT is applied to derive an interpretable translator to the DNN for a CarFollowing problem involving continuous action. The derived NLDT involves only one partition rule and two non-linear regression control rules and is able to successfully mimic the behaviour of the parent DNN controller.

The results obtained are very encouraging and open up a number of possible research paths. For classification problems, the parameter τ_I is required (Eq. 3.5). A parameter-free algorithm could be developed to conduct splits. Currently, the rule structure is fixed to be a weighted sum of power-laws. A genetic-programming-based approach can be used with the bilevel idea to derive rules with different non-linearity. Another possible extension to this work would involve developing an algorithm to handle categorical features in addition to ordinal or continuous features. It would also be interesting to see how the dual-bilevel approach can be applied to any generic regression or continuous control task. Currently, this approach is sensitive to the parameters τ_{error} and τ_{mse} (Algo. 9 and Eq. 6.8). Another parallel research can be launched focused on automating the reasoning and explanation process of developed NLDTs. Currently, we could do it for simple

problems manually.

This work on producing a transparent AI system which is relatively interpretable used several concepts from the non-linear optimization literature, such as bilevel-optimization and constraint-handling. The training of inducing NLDT$_{OL}$ and NLDT* can be made better and faster by using concepts from surrogate optimization and metamodelling. A multi-objective-optimization-based approach can also be developed to induce NLDT wherein the *interpretability* and *performance* are modelled as separate objectives. As mentioned before, interpretability is subjective, so it calls for a joint collaborative research to develop transparent AI systems by integrating domain expertise of human experts and requirements of end-users. We hope that our work helps to advance the research in the field of interpretable artificial intelligence and beyond.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] H. Kim, W.-Y. Loh, Y.-S. Shih, and P. Chaudhuri, "Visualizable and interpretable regression models with good prediction power," *IIE Transactions*, vol. 39, no. 6, pp. 565–579, 2007.

[2] B. Letham, C. Rudin, T. H. McCormick, D. Madigan *et al.*, "Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model," *The Annals of Applied Statistics*, vol. 9, no. 3, pp. 1350–1371, 2015.

[3] J. H. Friedman, B. E. Popescu *et al.*, "Predictive learning via rule ensembles," *The Annals of Applied Statistics*, vol. 2, no. 3, pp. 916–954, 2008.

[4] J. A. Nelder and R. W. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society: Series A*, vol. 135, no. 3, pp. 370–384, 1972.

[5] T. J. Hastie and R. J. Tibshirani, *Generalized additive models*. CRC press, 1990, vol. 43.

[6] S. N. Wood, *Generalized additive models: An introduction with R*. CRC press, 2017.

[7] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, "Interpretable machine learning: definitions, methods, and applications," *arXiv preprint arXiv:1901.04592*, 2019.

[8] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[9] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.

[10] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.

[11] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 847–855.

[12] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[13] D. Heath, S. Kasif, and S. Salzberg, "Induction of oblique decision trees," in *IJCAI*, vol. 1993, 1993, pp. 1002–1007.

[14] S. K. Murthy, S. Kasif, and S. Salzberg, "A system for induction of oblique decision trees," *Journal of artificial intelligence research*, vol. 2, pp. 1–32, 1994.

[15] S. K. Murthy, S. Kasif, S. Salzberg, and R. Beigel, "Oc1: A randomized algorithm for building oblique decision trees," in *Proceedings of AAAI*, vol. 93. Citeseer, 1993, pp. 322–327.

[16] E. Cantú-Paz and C. Kamath, "Using evolutionary algorithms to induce oblique decision trees," in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2000, pp. 1053–1060.

[17] M. Kretowski, "An evolutionary algorithm for oblique decision tree induction," in *International Conference on Artificial Intelligence and Soft Computing*. Springer, 2004, pp. 432–437.

[18] A. Ittner and M. Schlosser, "Non-linear decision trees-ndt," in *ICML*. Citeseer, 1996, pp. 252–257.

[19] K. P. Bennett and J. A. Blue, "A support vector machine approach to decision trees," in *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, vol. 3. IEEE, 1998, pp. 2396–2401.

[20] H. Núñez, C. Angulo, and A. Català, "Rule extraction from support vector machines." in *Esann*, 2002, pp. 107–112.

[21] G. Fung, S. Sandilya, and R. B. Rao, "Rule extraction from linear support vector machines," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 32–40.

[22] M. Craven and J. W. Shavlik, "Extracting tree-structured representations of trained networks," in *Advances in neural information processing systems*, 1996, pp. 24–30.

[23] P. M. Murphy and M. J. Pazzani, "Id2-of-3: Constructive induction of m-of-n concepts for discriminators in decision trees," in *Machine Learning Proceedings 1991*. Elsevier, 1991, pp. 183–187.

[24] U. Johansson, R. König, and L. Niklasson, "The truth is in there-rule extraction from opaque models using genetic programming." in *FLAIRS Conference*. Miami Beach, FL, 2004, pp. 658–663.

[25] D. Martens, B. Baesens, T. Van Gestel, and J. Vanthienen, "Comprehensible credit scoring models using rule extraction from support vector machines," *European journal of operational research*, vol. 183, no. 3, pp. 1466–1476, 2007.

[26] H. Ishibuchi, T. Nakashima, and T. Murata, "Three-objective genetics-based machine learning for linguistic rule extraction," *Information Sciences*, vol. 136, no. 1-4, pp. 109–133, 2001.

[27] Y. Jin and B. Sendhoff, "Pareto-based multiobjective machine learning: An overview and case studies," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 3, pp. 397–415, 2008.

[28] H. Guo and A. K. Nandi, "Breast cancer diagnosis using genetic programming generated feature," *Pattern Recognition*, vol. 39, no. 5, pp. 980–987, 2006.

[29] M. Muharram and G. D. Smith, "Evolutionary constructive induction," *IEEE transactions on knowledge and data engineering*, vol. 17, no. 11, pp. 1518–1528, 2005.

[30] M. Shirasaka, Q. Zhao, O. Hammami, K. Kuroda, and K. Saito, "Automatic design of binary decision trees based on genetic programming," in *Proc. The Second Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'98*. Citeseer, 1998.

[31] S. Haruyama and Q. Zhao, "Designing smaller decision trees using multiple objective optimization based gps," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 6. IEEE, 2002, pp. 5–pp.

[32] C.-S. Kuo, T.-P. Hong, and C.-L. Chen, "Applying genetic programming technique in classification trees," *Soft Computing*, vol. 11, no. 12, pp. 1165–1172, 2007.

[33] M. C. Bot, "Improving induction of linear classification trees with genetic programming," in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, 2000, pp. 403–410.

[34] R. E. Marmelstein and G. B. Lamont, "Pattern classification using a hybrid genetic program-decision tree approach," *Genetic Programming*, pp. 223–231, 1998.

[35] E. M. Mugambi, A. Hunter, G. Oatley, and L. Kennedy, "Polynomial-fuzzy decision tree structures for classifying medical data," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2003, pp. 155–167.

[36] A. Sinha, P. Malo, and K. Deb, "A review on bilevel optimization: From classical to evolutionary approaches and applications," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 2, pp. 276–295, 2017.

[37] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex systems*, vol. 9, no. 2, pp. 115–148, 1995.

[38] K. Deb, *Multi-objective optimization using evolutionary algorithms*. Wiley, 2005.

[39] L. Bobrowski and M. Kretowski, "Induction of multivariate decision trees by using dipolar criteria," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2000, pp. 331–336.

[40] M. Kretowski and M. Grześ, "Global induction of oblique decision trees: an evolutionary approach," in *Intelligent Information Processing and Web Mining*. Springer, 2005, pp. 309–318.

[41] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[42] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[43] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[44] J.-P. Vert, K. Tsuda, and B. Schölkopf, "A primer on kernel methods," *Kernel methods in computational biology*, vol. 47, pp. 35–70, 2004.

[45] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[46] K. Deb and A. Srinivasan, "Innovization: Innovating design principles through optimization," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1629–1636.

[47] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable multi-objective optimization test problems," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, vol. 1. IEEE, 2002, pp. 825–830.

[48] Y. Dhebar, S. Gupta, and K. Deb, "Evaluating nonlinear decision trees for binary classification tasks with other existing methods," *ArXiv*, vol. abs/2008.10753, 2020.

[49] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[50] 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*, vol. 12, pp. 2825–2830, 2011.

[51] K. Larsen, "Gam: the predictive modeling silver bullet," *Multithreaded Stitch Fix*, vol. 30, pp. 1–27, 2015.

[52] K. Neshatian, M. Zhang, and P. Andreae, "A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 5, pp. 645–661, 2012.

[53] A. Cano, A. Zafra, and S. Ventura, "An interpretable classification rule mining algorithm," *Information Sciences*, vol. 240, pp. 1–20, 2013.

[54] I. De Falco, A. D. Cioppa, and E. Tarantino, "Discovering interesting classification rules with genetic programming," *Applied Soft Computing*, vol. 1, no. 4, pp. 257–269, 2002.

[55] M. C. J. Bot and W. B. Langdon, "Application of genetic programming to induction of linear classification trees," in *European Conference on Genetic Programming*. Springer, 2000, pp. 247–258.

[56] J. Eggermont, J. N. Kok, and W. A. Kosters, "Genetic programming for data classification: Partitioning the search space," in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 1001–1005.

[57] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng, "Mining multiple comprehensible classification rules using genetic programming," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02*, vol. 2. IEEE, 2002, pp. 1302–1307.

[58] H. Iba, "Bagging, boosting, and bloating in genetic programming," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, 1999, pp. 1053–1060.

[59] J. Blank and K. Deb, "pymoo - Multi-objective Optimization in Python," https://pymoo.org.

[60] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning (ICML)*, 2015, pp. 1889–1897.

[61] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[62] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[63] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1928–1937.

[64] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems 8*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds. MIT Press, 1996, pp. 1038–1044.

[65] J. Peters, K. Mülling, and Y. Altun, "Relative entropy policy search," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, vol. 10. Atlanta, 2010, pp. 1607–1612.

[66] W. D. Smart and L. P. Kaelbling, "Practical reinforcement learning in continuous spaces," in *International Conference on Machine Learning (ICML)*, 2000, pp. 903–910.

[67] R. Noothigattu, D. Bouneffouf, N. Mattei, R. Chandra, P. Madan, K. Varshney, M. Campbell, M. Singh, and F. Rossi, "Interpretable multi-objective reinforcement learning through policy orchestration," *arXiv preprint arXiv:1809.08343*, 2018.

[68] F. Maes, R. Fonteneau, L. Wehenkel, and D. Ernst, "Policy search in a space of simple closed-form formulas: towards interpretability of reinforcement learning," in *International Conference on Discovery Science*. Springer, 2012, pp. 37–51.

[69] D. Hein, S. Udluft, and T. A. Runkler, "Interpretable policies for reinforcement learning by genetic programming," *Engineering Applications of Artificial Intelligence*, vol. 76, pp. 158–169, 2018.

[70] G. Liu, O. Schulte, W. Zhu, and Q. Li, "Toward interpretable deep reinforcement learning with linear model u-trees," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 414–429.

[71] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," *arXiv preprint arXiv:1804.02477*, 2018.

[72] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[73] D. Hein, A. Hentschel, T. Runkler, and S. Udluft, "Particle swarm optimization for generating interpretable fuzzy reinforcement learning policies," *Engineering Applications of Artificial Intelligence*, vol. 65, pp. 87–98, 2017.

[74] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 627–635.

[75] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in neural information processing systems (NIPS)*, 2018, pp. 2494–2504.

[76] O. Bastani, C. Kim, and H. Bastani, "Interpretability via model extraction," *arXiv preprint arXiv:1706.09773*, 2017.

[77] G. Vandewiele, O. Janssens, F. Ongenae, F. De Turck, and S. Van Hoecke, "Genesim: Genetic extraction of a single, interpretable model," *arXiv preprint arXiv:1611.05722*, 2016.

[78] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 503–556, 2005.

[79] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[80] A. Silva, M. Gombolay, T. Killian, I. Jimenez, and S.-H. Son, "Optimization methods for interpretable differentiable decision trees applied to reinforcement learning," in *International Conference on Artificial Intelligence and Statistics*, 2020, pp. 1855–1865.

[81] Y. Dhebar and K. Deb, "Interpretable rule discovery through bilevel optimization of split-rules of nonlinear decision trees for classification problems," *arXiv preprint arXiv:2008.00410*, 2020.

[82] S. Nageshrao, B. Costa, and D. Filev, "Interpretable approximation of a deep reinforcement learning agent as a set of if-then rules," in *18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 2019, pp. 216–221.

[83] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning" *arXiv preprint arXiv:1509.06461*, 2015.

[84] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," University of Cambridge, Department of Engineering Cambridge, UK, Tech. Rep. CUED/F-INFENG/TR166, 1994.

[85] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2020.

[86] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[87] N. Subramanya, C. Bruno, and D. Filev, "Interpretable approximation of a deep reinforcement learning agent as a set of if-then rules."