DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Learning Domain-Specific Predicates in Decision Trees for Explainable Controller Representation

Christoph Weinhuber

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Learning Domain-Specific Predicates in Decision Trees for Explainable Controller Representation

Lernen von domänespezifischen Prädikaten in Entscheidungsbäumen für die Repräsentation erklärbarer Controller

Author:	Christoph Weinhuber
Supervisor:	Univ.-Prof. Dr. Jan Křetínský
Advisor:	M.Sc. Pranav Ashok, M.Sc. Maximilian Weininger
Submission Date:	04.09.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 04.09.2020 Christoph Weinhuber

Acknowledgments

First of all, I would like to thank my supervisor Prof. Dr. Jan Kretínský for making this thesis possible and providing me the opportunity to do research in such fascinating topic.

Secondly, I cannot stress enough how deeply grateful I am for my research advisors M.Sc. Maximilian Weininger and M.Sc. Pranav Ashok. Their continuous support, constructional guidance and infinite patience remains unmatched. I am especially thankful for the positive working environment they have created; they always had time for me and every single question could be asked. It was the most exciting topic I have ever encountered and I enjoyed every single day working on it.

Last but not least, I would like to thank my mother for all her care and energy she invested in me throughout my academic path and my girlfriend for her unconditional support and always being my partner in crime.

Abstract

Controller which arise during the model checking process provide valuable insights into the system under examination. But often this advantage gets buried under their opaque size. Recent work addressed this problem and demonstrated the use of decision trees for controller representation which resulted in several advantages. However, the explainability of certain decision trees is still insufficient. To overcome this problem we present a concept to enrich decision trees with algebraic predicates. We propose several different approaches to leverage domain knowledge presented by the user, either automatically or semi-automatic. Furthermore, we extend the current version of `dtControl` with all proposed concepts and methods. Finally, we demonstrate the advantages and applicability of our approach on different controller models where we yield great gains in both size and explainability.

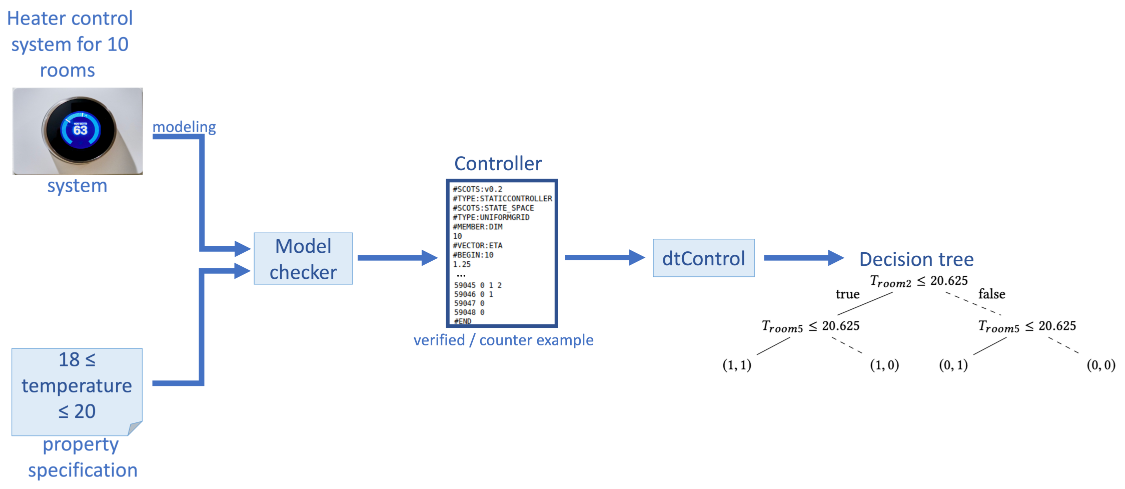
Contents

1. Introduction

We live in a world entirely surrounded by computer systems. Their development in the past several decades led up to the point that almost every aspect of our day-to-day life can be improved by a technical device. From the smart electric toothbrush providing plaque detecting mechanism [Col20] to the connected fridge with Spotify [Sam] nearly every single object we interact with can be enhanced. Above all, our reliance on those systems developed in such a fast pace that society is not even aware of the sheer amount of hidden computer systems they depend on, on a daily basis. A common and popular example can be found within a car. Safety-relevant components like braking or airbags are all already based on embedded software solutions. Coupled with the increased flexibility and improved cost-efficiency more components are going to follow [BK08, Foreword]. Beyond this development of higher usage and dependency on those systems, we could also observe their growing complexity. While there were 400,000 lines of source code in the first space shuttle [Dun], a car in 2009 could already contain close to 100,000,000 lines of code [Cha]. Meanwhile, [Lip82] traced the correlation of growing complexity and incorrect systems and concluded that an increase in complexity leads to more errors per line of code. History has shown even one single accidentally duplicated line of code can already cause a huge disaster, as happened in February 2014 with the bug known as "The Apple goto fail bug". This single incorrect line of code caused a significant security flaw by letting keys pass a test incorrectly. A more recent and more dramatic example can be found within Boeing's malfunctioning system called "Maneuvering Characteristics Augmentation System" which was designed to correct stabilisation issues of the Boeing 737 MAX. This incorrect software caused two planes to crash in October 2018 and five months later in March 2019, causing the death of 346 people [DOT20].
One general lesson which can be extracted from every disaster is presented by Dias et al., who investigated this accident, in their golden rule:

"[S]afety always comes first. Critical safety issues are non-negotiable and should always be addressed in the first place, no matter the costs." [DOT20, Chapter 7.1, p.193]

But how to prove a system's correctness despite the ever-increasing complexity? This question opened up a whole new challenge in computer science and one auspicious approach is presented within the field of Model Checking.



```
#SCOTS:v0.2
#TYPE:STATICCONTROLLER
#SCOTS:STATE_SPACE
#TYPE:UNIFORMGRID
#MEMBER:DIM
10
#VECTOR:ETA
#BEGIN:10
1.25
 ...
 59045 0 1 2
 59046 0 1
 59047 0
 59048 0
 #END
```

Figure 1.1.: A schematic overview of an example workflow with 10 rooms [JZ17] and SCOTS [RZ16]. Heater control symbol from [LeF18], controller obtained from [MPM20] and decision tree acquired from [Ash+20a].

Model Checking

Model Checking [CGP01] is currently one of the most effective verification techniques to automatically detect and prove the absence of bugs. Figure 1.1 depicts an exemplary model checking workflow. The first input to the model checker is the system which is going to be checked. As a concrete example, consider a heater control system that controls the temperature of 10 connected rooms by turning the heaters on or off. The second input to the model checker is a list of properties which the heater control system should satisfy. In 1.1, the concrete property expects the temperature of the 10 connected rooms to always be between 18 and 20. The model checker itself validates if the system always satisfies the given properties. Edmund M. Clarke describes in [Cla+18, Chapter 1.1] the basic classical form of model checking in three separated parts:

Model

The system under examination will be mapped to a finite transition system, called System Model. Depending on the Property Specifications and the actual behaviour of the system under examination, the exact modeling language can vary. In Figure 1.1, the model is represented by the heater control system for 10 rooms.

Specification
: The Property Specification contains a list of properties which the system under examination should satisfy. Typical checked properties are presented by [BK08] e.g. deadlocks, valid response or general response time. In Figure 1.1, the specification expects the temperature always to be between 18 and 20.

Model Checking
: This part defines a process in which an algorithm checks whether the given property specifications are satisfied at every given state of the System Model. If the model checking algorithm succeeds (i.e. memory was sufficient and it doesn't take infeasibly long) it will either verify or detect a violated property specification. Similar to the modeling process, the exact algorithm can vary depending on the initial conditions. However, this thesis focuses on algorithms constructing controllers as in `SCOTS` [RZ16], Uppaal Stratego [Dav+15] and `PRISM` [KNP11].

The resulting controllers returned by the model checking algorithm are represented as lookup tables. These lookup tables typically store the corresponding actions for every state of the system, also known as state-action pairs. These lookup tables either describe precisely how the system exactly fulfils the property specifications or present a specific violated specification through a counterexample. In addition, when a controller satisfies a given property specification, its state-action pairs can directly be used to implement an embedded device which will therefore also satisfy the property specification. However, since controllers are typically represented as lookup tables, this results in several disadvantages, as reported in [Jac20] and [Ash+20a].
One of their major drawbacks can be traced back to their size. Not only do controllers end up having several millions of different states, but moreover they can also contain non-determinism. Thereby, the key advantage of providing valuable insights gets buried under their sheer, opaque size, and complexity. Additionally, since controllers and their strategies can be utilised to operate on embedded devices, either those devices need large amounts of memory or the controller-strategy size needs to be drastically compressed.

Current Situation. A practical solution towards this problem is presented by the open-source tool `dtControl` [Ash+20a]. `dtControl` enables a controller to be modeled as a decision tree and thus typically achieves, for standard benchmarks from literature, a 96% size reduction while still preserving all guarantees. Apart from transferring controllers into a space-saving format, these resulting trees can reveal patterns within the controller and therefore increase the overall understandability and confidence of the controller correctness.

Limitations. Nevertheless, the comprehensibility of controllers like Cruise Control suffers frequently. Although their size is effectively compressed, the outcome is not guaranteed to provide valuable insights into the exact behavior of the controller. In many cases, there is additional domain knowledge about the controller's behavior available, which could potentially simplify the representation of controller. Furthermore, for some controllers like Cruise Control there are even preexisting handcrafted strategies available which could effectively represent the whole controller. However, the main problem until now was that controllers were treated like black boxes and even if there was additional domain knowledge available it could not be productively used to simplify the representation. There only exists a prototype that requires the user to handcraft a grammar from the domain knowledge. This process is prohibitively complicated and error prone; also, it does not scale to more than a few knowledge items.

Contribution This work addresses the problem of small and explainable controller representation. To achieve both of these goals, we will presents several theoretical concepts to enrich decision trees with algebraic predicates. Additionally, we propose several different approaches to leverage domain knowledge presented by the user, either automatically or semi-automatic. Hereby we additionally introduce a dedicated user-interface for the semi-automatic approach. Through this interface a user can both provide domain knowledge and control the decision tree induction process. Furthermore, we provide a detailed explanation on exactly how we extended and implemented all concepts and methods within the current version of `dtControl`. Finally, we demonstrate our approach on several different controller models.

Structure. This thesis is organized as the following:

- Chapter 2 contains all the related work which is relevant for understanding the theory and its implementation.
- Chapter 3 provides the definitions for all important notions and terms we build upon throughout this work.
- Chapter 4 introduces all theoretical concepts to enrich decision trees with algebraic predicates.
- Chapter 5 focuses on the practical implementation of the theory and provides detailed information on how exactly we extended the current version of `dtControl`.
- Chapter 6 demonstrates our approach on several different controller models.
- Chapter 7 provides different concepts and ideas for the future which could potentially improve `dtControl`.

- Chapter 8 concludes this work by providing a summary of the presented approach.

2. Related Work

In this chapter we introduce the related work which is relevant for the theory and its implementation presented in this thesis. The first half of this chapter summarizes theory related work which mainly consists of decision trees and curve fitting. The second half concludes this chapter by presenting all related tools which we either build upon or compare our work to.

Decision Tree. *Decision trees* can be typically found within the field of Machine Learning. One of the standard references for decision trees is given by [Mit97, Chapter 3]. In general, every decision represents decision-making guidelines, based on empirical data. Their key advantage of hierarchically presenting these decision-making guidelines make them especially profitable for our use-case, namely representing controller models. Previous work of [Ash+20a] and [Brá+15] provide a detailed analysis of the benefits which decision trees provide for controller representation.

One key extension for decision trees is presented within the concept of *oblique* decision trees [Mur+93]. This concept extended decision trees in a way, that hyperplane predicates could be used and not only axis-parallel predicates. The last key extension was the concept of *algebraic* decision trees [Yao92], which enabled the usage of algebraic predicates. Currently there only exist automatic induction algorithms for axis-parallel decision trees and oblique [HKS93] decision trees. Popular examples for axis-parallel induction algorithms are ID3 [RM14, Chapter 7.2] [Qui86] or CART [RM14, Chapter 7.4] [Bre+84, Chapter 2.4].

Curve Fitting. *Curve Fitting* as presented in [Arl94; Kol84] describes a common optimization technique which can be typically found in the field of Data Science. Typically, the objective of Curve Fitting is to calculate certain parameters within a function in a way that this function will reproduce desired data points. Within the field of Curve Fitting we mostly utilize the concept of *Least Square Fitting* [Adc77; ADC78; ARW01].

Tool. Throughout this thesis, all presented concepts are implemented within the open-source tool `dtControl`. `dtControl` was firstly presented in [Ash+20a] and enables a controller to be modeled as a decision tree. Previous work of [Jac20] already extended `dtControl` by providing several different decision tree induction algorithms. Nevertheless, the explainabilty of these decision trees is still insufficient. The aim of this thesis is to provide new features which increase the explainability of the decision tree representation. Hereby only one alternative to `dtControl` exists, a prototype which is presented by [Akm19]. This prototype requires the user to handcraft a grammar from the domain knowledge which is prohibitively complicated and error prone.

3. Preliminaries

In this chapter, all important notions and terms are introduced. In particular, we cover all definitions used in Chapter 4, starting with the general concept of a *controller*, followed by an introduction of our highly utilized data structure *decision tree*, with its related concepts, namely *Predicate*, *Impurity Measure* and *Information Gain*. This chapter ends with an explanatory description of *Curve Fitting*.

3.1. Controller

Controllers form the cornerstone of this work as we try to reduce their size and increase their understandability. As previously mentioned in chapter 1, Controllers can be described as state-action pairs synthesized by model checking algorithms[1]. In general, these state-action pairs provide detailed information about the behavior of the modeled system. They either describe how the modeled system exactly satisfies the given property specifications or represent a specific violated specification via a counterexample. Similar to the definition of [Jac20, Chapter 3.1], which was inspired by [Brá+15; Ash+19], we introduce the term Controller as the following:

Definition 1 Controller. Let $\mathcal{S}$ be the set of all possible states and $\mathcal{A} = \{f : \mathcal{S} \to \mathcal{S}\}$ be the set of all possible actions.
A *Controller* $\mathcal{C} \subseteq S \times A$ is a set of state-action pairs. Important properties distinguished in this thesis are:

- Performing the action f to a corresponding state is called *Transitioning*. Transitions do not have to be deterministic.

  Controller $\mathcal{C}$ is *deterministic* if $\forall (s, a) \in \mathcal{C}.\ \nexists\ (s, a') \in \mathcal{C}$ with $a \neq a'$. This work assumes the determinism of every Controller but it should be remembered that any non-deterministic controller can be converted into a deterministic one by assigning labels to sets of actions [Jac20, Chapter 5.4].

[1]This work focuses on model checking algorithms implemented in tools such as `SCOTS` [RZ16], Uppaal Stratego [Dav+15] and `PRISM` [KNP11].

- Some controllers may require memory usage.

  This thesis only encompasses *memoryless* controllers. However, it should be noted that any controller with memory usage can be converted into a memoryless controller by mapping the memory into different states [BK08, Chapter 10.6.4].

3.2. Decision Tree Learning

This subsection starts by introducing an example data set used throughout the following sections. Within this chapter, standard decision tree learning is applied to the example data set and the corresponding definitions are introduced on the fly.

Running Example. The data set displayed in Table 3.1 contains nutrition information of raw fruits. Each row is an own instance of a represented *Label*, visualized by the last column "Fruit". The first four columns "Calories", "Vitamin C Percentage", "Tasty" and "Gram protein" are referred to as *Features*. Features can be described as the attributes of a label. The basic concept of every decision tree learning algorithm is to process and generalize the feature columns in order to predict the label of an unknown object.

Note. Many other resources refer to Labels as *Target Variables* or to *Attributes* instead of Features.

Note. Any arbitrary Controller $\mathcal{C}$ with $h = |\mathcal{C}|$ can be converted to a data set with h rows by mapping its States to Features and its corresponding Actions to Labels.

Note. Features and Labels can either be referred to as *Categorical*, e.g the "Tasty" feature in Table 3.1 or *Numerical*, e.g. the remaining Features. For simplification, this work insists on numerical features and encodes internally every categorical feature with a unique value (similar to the concept of a dummy variable [Guj70]). A concrete example for the "Tasty" feature: True $\rightarrow$ 1 and False $\rightarrow$ 0.

Note. To specify the explicit values inside features of a data set $\mathcal{D}$, with rows h and number of features b, the variable $\mathcal{D}_\text{X} = ((v_{1,1}, \dots, v_{1,b}), \dots, (v_{h,1}, \dots, v_{h,b}))$ will be used. Analog for the label-space with $\mathcal{D}_\text{Y}$. Throughout this work, we will use the terms Features/Labels as synonyms for $\mathcal{D}_\text{X}/\mathcal{D}_\text{Y}$.
Running Example. $\mathcal{D}_\text{X}$ and $\mathcal{D}_\text{Y}$ for the data set displayed in Table 3.1 have the following structure: $\mathcal{D}_\text{X}$ = ((130, 8, True, 1), . . . , (50, 50, True, 1)) and $\mathcal{D}_\text{Y}$ = ((Apple), (Apple), (Banana), (Honeydew-Melon), (Tangerine), (Strawberries), (Pineapple)).

Note. For a higher compatibility within mathematical definitions used throughout this thesis, a mapping between features/labels and (x,y)-coordinates can be performed. $\mathcal{D}_X = ((v_{1,1},\dots,v_{1,b}),\dots,(v_{h,1},\dots,v_{h,b}))$ can also be described by a *feature vector*, containing h b-dimensional vectors, representing x-coordinates. Analog for converting $\mathcal{D}_Y$ into a *label vector*, representing y-coordinates.

Table 3.1.: Running Example data set containing exemplary nutrition values of fruits. Data set consists of four feature columns and one label ("Fruit"). Nutrition values extracted from [FN].

Calories	Vitamin C Percentage	Tasty	Gram protein	Fruit
130	8	True	1	Apple[2]
130	8	False	1	Apple[3]
110	15	True	1	Banana
50	45	True	1	Honeydew-Melon
50	45	True	1	Tangerine
50	160	True	1	Strawberries
50	50	True	1	Pineapple

3.2.1. Predicate

A *Predicate* is a (composition of) function(s) that maps all members of $\mathcal{D}_X$ within a data set $\mathcal{D}$ to the boolean domain.

Definition 2 Predicate. Let $\mathcal{F}$ be a set containing all features represented in a data set $\mathcal{D}$. The predicate function $p(x) = g(f(x))$ over a data set $\mathcal{D}$ takes as input a variable set of feature combinations $X_1 \times X_2 \times \cdots \times X_n$ with $X_n \subseteq \mathcal{F}$ resulting in

$$p : (X_1, X_2, \dots, X_n) \to \{True, False\}$$

Within this context, the function $f : (X_1, X_2, \dots, X_n) \to \mathbb{R}$ represents an arbitrary algebraic term and the function $g : y \to \{True, False\}$ with $y \in \mathbb{R}$ represents an

[2]Red Apple.
[3]Green Apple.

arbitrary function into the boolean domain. Inspired by [Brá+15, Chapter 5.2], we place restrictions on g and assume predicates of following structure:

$$[f(x) \sim \mathit{const}]$$

with $\sim \in \{\leq, \geq, >, <, =\}$ and $\mathit{const} \in \mathbb{R}$.

Note. To provide a higher consistency throughout the following chapters, an alternative Feature referencing pattern will be introduced: x_i ("x_i" in source-code) for referencing the Feature at index i, with $\mathrm{i} \in \mathbb{N}_0$.

Note. To reference the portioned sub data sets after applying a predicate, we will use: $\mathcal{D}_{\text{true}}$ and $\mathcal{D}_{\text{false}}$.

Running example. For the data set of Table 3.1, with

$$\mathcal{F} = \{\mathit{Calories},\ \mathit{Vitamin\ C\ Percentage},\ \mathit{Tasty},\ \mathit{Gram\ Protein}\}$$

possible Predicates are depicted in Figure 3.1 and Figure 3.2.

3.2.2. Impurity Measure

There are several different impurity measurements available[4] which all use a variety of metrics. In general, every impurity measurement calculates the "randomness" of labels within a data set $\mathcal{D}$ by comparing its heterogeneity.

Entropy

One common exemplary impurity measurement, as introduced by [Sha48], is referred to as Entropy:

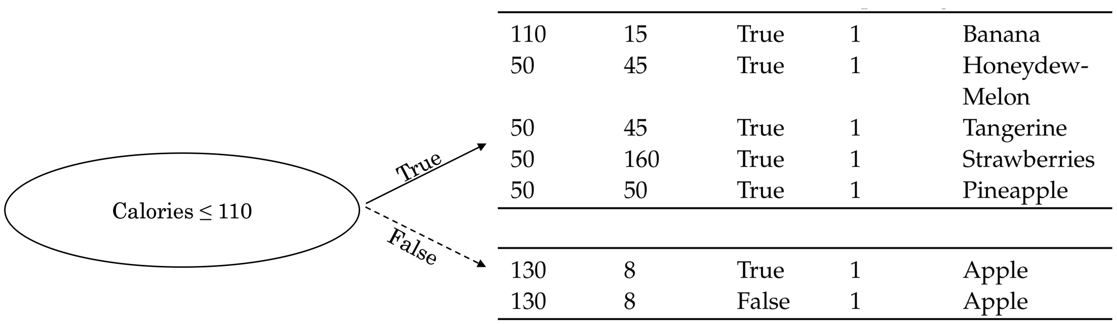
$$\text{Entropy}(\mathcal{D}) = -\sum_{\text{label} \in \mathcal{D}} \Pr[\text{label}] * \log_2(\Pr[\text{label}])$$

Running example. Consider the two sub data sets $\mathcal{D}_{\text{true}}$ and $\mathcal{D}_{\text{false}}$, depicted in the example predicate of Figure 3.1:

$\text{Entropy}(\mathcal{D}_{\text{true}})$
$= -(\Pr[\text{"Banana"}] * \log_2(\Pr[\text{"Banana"}]) + \cdots + \Pr[\text{"Pineapple"}] * \log_2(\Pr[\text{"Pineapple"}]))$
$= -(\frac{1}{5} * \log_2(\frac{1}{5}) + \cdots + \frac{1}{5} * \log_2(\frac{1}{5})) \approx 2.3$

$\text{Entropy}(\mathcal{D}_{\text{false}}) = -(\Pr[\text{"Apple"}] * \log_2(\Pr[\text{"Apple"}])) = 0$

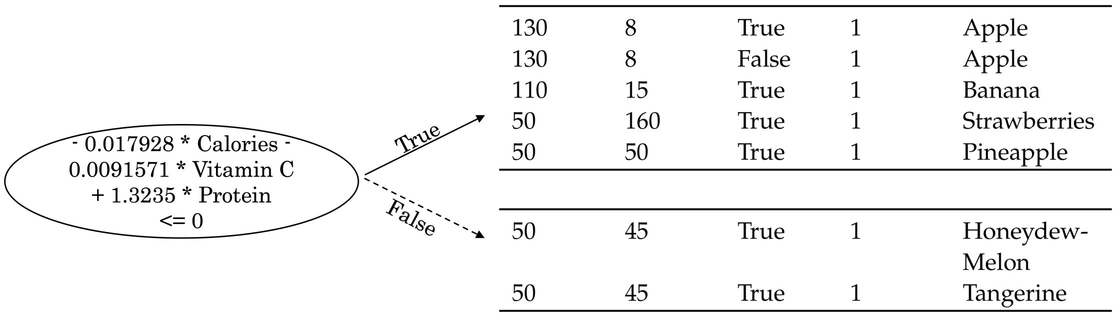
[4]See [Jac20, Chapter 5.3] for an overview of all impurity measurements present in `dtControl`.



110	15	True	1	Banana
50	45	True	1	Honeydew-Melon
50	45	True	1	Tangerine
50	160	True	1	Strawberries
50	50	True	1	Pineapple

130	8	True	1	Apple
130	8	False	1	Apple

Figure 3.1.: An example predicate using the Calories Feature to split the data set.


- 0.017928 * Calories - 0.0091571 * Vitamin C + 1.3235 * Protein <= 0
True
False

130	8	True	1	Apple
130	8	False	1	Apple
110	15	True	1	Banana
50	160	True	1	Strawberries
50	50	True	1	Pineapple

50	45	True	1	Honeydew-Melon
50	45	True	1	Tangerine

Figure 3.2.: An example predicate using a combination of Calories, Vitamin C Percentage and Gram Protein to split the data set.

Information Gain

Information Gain provides an extension to the Entropy measurement for general usage within the context of predicates and decision trees. It quantifies the reduction of "randomness" of labels after using a predicate p on the current data set $\mathcal{D}$. One standard *Information Gain* Formula is given by [Mit97, Chapter 3.4.1.2]:

$$\mathrm{IG}(\mathcal{D}, p) = \mathrm{Entropy}(\mathcal{D}) - (\frac{|\mathcal{D}_{\text{true}}|}{|\mathcal{D}|} * \mathrm{Entropy}(\mathcal{D}_{\text{true}}) + \frac{|\mathcal{D}_{\text{false}}|}{|\mathcal{D}|} * \mathrm{Entropy}(\mathcal{D}_{\text{false}}))$$

Running example. Consider the predicate $p = [\text{Calories} \leq 110]$ depicted in Figure 3.1, over the data set $\mathcal{D}$ illustrated by Table 3.1:

$$\mathrm{IG}(\mathcal{D}, p) = 2.52 - (\frac{5}{7} * 2.3 + \frac{2}{7} * 0) \approx 0{,}88$$

3.2.3. Decision Tree

Decision trees (DT) can be typically found within the field of Machine Learning. Over the years two different main application tasks developed. Regression trees[5] and classification trees. This work focuses on the latter one. The aim of a classification tree is to provide decision-making guidelines, based on empirical data, to predict the discrete label of an unknown object. Their ability to hierarchically present a large amount of information enables them to display those guidelines both efficiently and explainable. Previous work of [Ash+20a; Brá+15] has also shown the applicability and benefits of using decision trees for controller representation.

Terminology. Every decision tree consists of several *nodes* illustrated by circles. Each node represents a unique subset of data. Every node is connected by a directed edge (arrow) which represents a possible decision. The first node on top of the decision tree and without any incoming edge, is called the *root* node. Every other node has one incoming edge. Every node with an outgoing edge represents a predicate and a subset of the initial data. Due to the fact that we work with boolean predicate functions, every node can have maximum two outgoing decision edges. Nodes without any outgoing edges are called *leaves* and represent the estimated Labels at that current data subset.

Note. The *height* of a decision tree is the longest possible consecutive path from the root node to an arbitrary leaf node. The decision tree depicted in Figure 3.3 has a height of 3.

[5]Typically used to predict continuous Labels/Target Variables. Common and popular example is estimating house prices based on data of similar house sales.

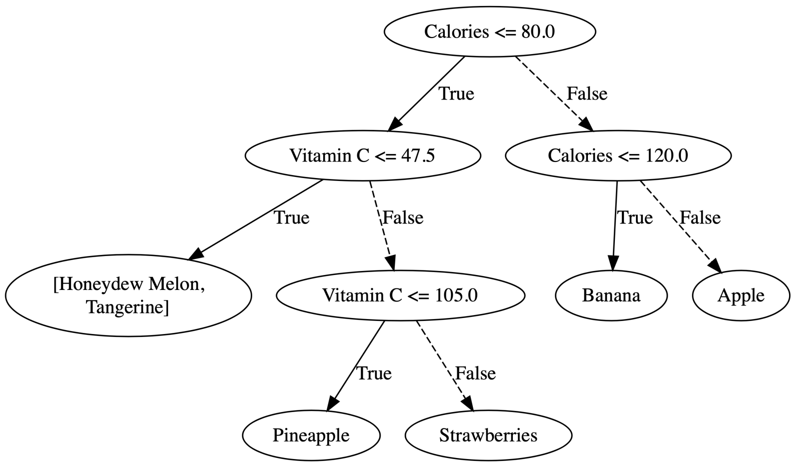


Figure 3.3.: Decision tree representation of Table 3.1 created with Axis Aligned Splitting Strategy.

Note. At every node within a decision tree the current sub data set gets partitioned. Therefore, every node represents its own unique subset of feature-label combinations.

Note. In Figure 3.3, the leaf labeled "[Honeydew Melon, Tangerine]" contains a set of two possible predicted Labels. This is because the data set of table 3.1 "Honeydew Melons" and "Tangerines" can not be differentiated and therefore this behavior is correct. The remaining labels $\{Apple, Banana, Strawberries, Pineapple\}$ can all be differentiated and thus every remaining label has their own leaf.

Definition 3 Decision Tree. Let $\mathcal{L}$ be a set containing all labels and p be the predicate function over the current data set $\mathcal{D}$. Building upon the definition presented by [Jac20], we define a *decision tree* recursively as:

1. A DT of height h = 0 is a leaf containing a subset of labels Y $\subseteq \mathcal{L}$.
2. A DT of height $\mathrm{h}+1$ is a tuple $(p, \mathcal{T})$ with
   - $\mathcal{T} = (\mathcal{T}_{\text{true}}, \mathcal{T}_{\text{false}})$ being a tuple representing the two child sub-trees with $\text{height}(\mathcal{T}_{\text{true}}) \leq h \land \text{height}(\mathcal{T}_{\text{false}}) = \mathrm{h}$ or
     $\text{height}(\mathcal{T}_{\text{false}}) \leq h \land \text{height}(\mathcal{T}_{\text{true}}) = \mathrm{h}$.
   - p is the predicate function used at the current node.

3.2.4. Predicate Generator

Definition 4 Predicate Generator. A *Predicate Generator* for a data set $\mathcal{D}$ can be described as a function $f : \mathcal{D} \rightarrow p$ with p being the generated predicate. In other words, a predicate generator contains a concrete pattern to create individual predicates for a specific data set.

Note. The aim of this thesis is to provide a new predicate generator that is able to include domain knowledge.

Example Axis Aligned Strategy. One common pattern to create predicates is called *Axis Aligned*, e.g. [Jac20, Chapter 5.2.1]. Let $\mathcal{D}$ be the current data set and $\mathcal{F}$ be the set containing all Features within $\mathcal{D}$. For every feature $f \in \mathcal{F}$ we introduce a sorted tuple $f_{\text{sort}} = (v_1, \ldots, v_n)$ with $v_i < v_{i+1}$ and n being the number of unique values present in f. Hereby, every member of f_{sort} represents one of these unique values. In other words, f_{sort} can be described as a sorted list of unique values present in the Feature f. The concrete pattern to create Axis Aligned Predicates is given by the following:

```
Iterate over every feature $f \in \mathcal{F}$ and generate $f_{\text{sort}}$:
        For $i < n$ with $i \in \mathbb{N}$:
            Calculate the average value of $v_i$ and $v_{i+1}$:
                $avg = \frac{v_i + v_{i+1}}{2}$
            Add predicate of structure [f $\leq avg$)] to collection.
```

Running example. To illustrate all possible Axis Aligned predicates created by the algorithm, the modified decision tree in Figure 3.4, displays the collections of created predicates at every given node. This modified tree is based on the data of Table 3.1. The encoding of the categorical features within the "Tasty" Feature was according to their numerical value, meaning True $\rightarrow$ 1 and False $\rightarrow$ 0.

3.2.5. Decision Tree Induction Algorithm

The *Decision Tree Induction Algorithm* can be described as a strategy to automatically construct a decision tree for a data set $\mathcal{D}$. In general, every induction algorithm consists of two separate components, a predicate generator and an impurity measure. As previously mentioned in Section 3.2.2, there are several different impurity measures to choose from. Technically, every decision tree induction algorithm follows the same greedy top-down approach [Mit97]:

1. Create a node for the current data set $\mathcal{D}'$.
2. Utilize the predicate generator to create a collection of suitable predicate candidates.
3. If every predicate candidate partitions $\mathcal{D}'$ in one single subset,
(e.g. $\mathcal{D}'_{\text{true}} = \mathcal{D}' \wedge \mathcal{D}'_{\text{false}} = \emptyset$ or $\mathcal{D}'_{\text{false}} = \mathcal{D}' \wedge \mathcal{D}'_{\text{true}} = \emptyset$)
return the labels of $\mathcal{D}'$.
4. Else use the best predicate candidate according to the chosen impurity measurement and apply the induction algorithm on the sub trees $\mathcal{D}'_{\text{true}}$ and $\mathcal{D}'_{\text{false}}$.

Note. Throughout this work, we will often use the term *strategy* or *decision tree building strategy* as synonym for a decision tree induction algorithm.

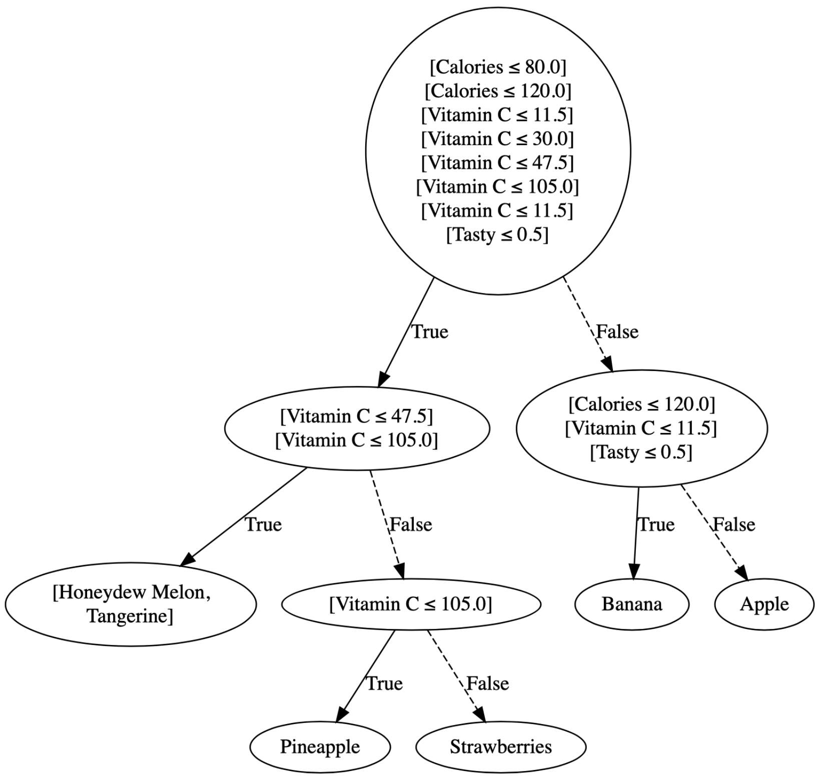


Figure 3.4.: Schematic DT containing all possible Axis Aligned Split for every node. Based on data set depicted by Table 3.1.

3.3. Curve Fitting

Curve fitting is a common optimization technique especially prominent within the field of Data Science. It can be used as a tool to calculate parameters within a function f, in a way that f will reproduce desired data points. The process of determining the optimal parameters is also called *fitting*.

When applying curve fitting within this work, we assume that we have a desired function[6] we would like to fit to preexisting data points, as depicted in Figure 3.5 by black crosses.

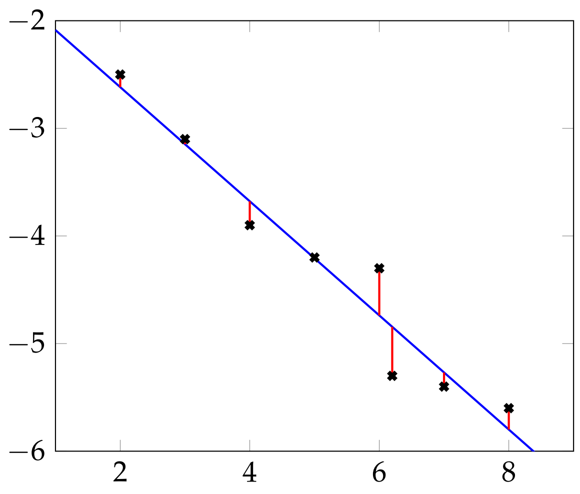


Figure 3.5.: Example scatter-plot of collected data points (black cross) with the desired fitted function (blue) and an illustration of the error in accuracy (red).

At first glance, a trend across the scattered points can easily be detected. To optimize the process of Curve Fitting, it is recommended to fit a function which already embodies the existing trend of the data points. Therefore, for this example, the function $f(x) = m * x + t$ will be fitted to the data points. Before the actual calculation can start, an exact metric for the accuracy of the current fit has to be defined.

One standard technique to fit a function to an arbitrary set of data is called *Least Square Fitting*. The idea behind this strategy is to calculate the parameters of our function f in order to minimize the *Root Mean Square* which is given by:

$$RMS(f) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2}$$

with n being the number of different data points and x_i,y_i being the coordinates of the data point i. Intuitively, the goal can be illustrated by trying to minimize the accuracy

[6]Generally, any arbitrary function can be fitted to any data points. It only depends on the desired accuracy.

error represented as a red line in Figure 3.5. For an overview of the general concept of *Least Square Problems*, we refer to [NW06, Chapter 10]. With assistance of the library [Oli06] or respectively its function [Num], one can immediately obtain the optimized values for the example function f. The final result for the example in Figure 3.5 was

$$f(x) = -0.53024607 * x - 1.55673274$$

with $RMS(f) \approx 0.255$.

Note. To maintain consistency throughout code examples, we will reference every occurrence of Curve Fitting by:

$$\texttt{curveFit}(f, \text{X}, \text{Y})$$

with f being the function to be fitted, X describing an ordered list of x coordinates and Y representing the corresponding y coordinates (see notes in Chapter 3.2.3 for mapping between features/labels and coordinates). To reduce complexity we assume that `curveFit` will directly substitute the parameters within function f for their fitted values.

Example. In Figure 3.5, `curveFit` returns instead of $[-0.53024607, -1.55673274]$ directly the term$f(x) = -0.53024607 * x - 1.55673274$.

4. Richer Domain Predicates

This chapter introduces several different extensions for predicates and decision tree induction algorithms. We start by providing a detailed explanation of the theory behind Richer Domain Predicates and introduce its related concepts namely Finite Coefficients, Infinite Coefficients and Feature Constraints. Building upon the concept of Richer Domain Predicates, several different general and domain specific extensions for decision tree induction algorithms are introduced. Throughout the sections, we also provide further optimization techniques, to increase the productiveness of our concepts.

4.1. Predicate Tailoring

The limits of predicates following the simplified structure defined in Section 3.2.1 are quickly reached. Especially, when expressions contain individual restrictions or only a certain premonition for a (potentially) more promising structure is available. To overcome those limitations and provide a broader range of individual customization options, the concept of *Predicate Tailoring* is introduced. *Predicate Tailoring*, as presented within this work, describes the general possibility to both extend and restrict predicates in a variety of different ways which will be presented in the following subsections.

4.1.1. Coefficients

The first step towards a variety of different customization options is done by enabling the usage of general coefficients within a predicate. This subsection introduces all possible modification capabilities available to coefficients.

Note. To maintain consistency, the following sections refer to coefficients as c_i ("c_i" in source-code) with $i \in \mathbb{N}_0$.

Finite Coefficients

Finite Coefficients form the basic framework for the usage of coefficients within predicates. Previously, several individual predicates were needed to express alternatives. Now, Finite Coefficients provide the opportunity to bundle these variations into one single predicate. Building upon Definition 2 introduced by Section 3.2.1, the structure of predicates using Finite Coefficients can be described as the following:

$$[term_f \sim const;\ coef\_def]$$

with the extension of $term_f$ being an arbitrary algebraic term using Finite Coefficients defined in *coef_def*. The general idea is to utilize Finite Coefficients c_i within $term_f$ and extend the finished predicate with a semicolon, followed by the concrete coefficient definition $C_i \subseteq \mathbb{R}$ with $c_i \in C_i$.

Note. For computational reasons, the only restriction we insist on is $|C_i| \in \mathbb{N}$. However, it should be noted that specifying too many or too large sets results in an infeasibly large set of predicates.

Running Example. Exemplary predicates, demonstrating the range of modification capabilities for Finite Coefficients (based on data set depicted in Table 3.1):

- $[\sqrt{x_0} * c_0 + log(x_1) - \frac{x_2}{c_1} \leq c_2;\ c_0 \text{ in } \{-\frac{1}{3}, \frac{1}{3}\};\ c_1 \text{ in } \{1{,}2\};\ c_2 \text{ in } \{\sqrt{2}, \pi\}]$
- $[1.0546e - 34 + 90 - c_0 \geq x_0;\ c_0 \text{ in } \{0, 12\}]$
- $[e^{10} + ln(2) + \frac{x_2}{c_0} \geq 0;\ c_0 \text{ in } \{-\frac{1}{9}, \frac{2}{3}\}]$
- $[sin(cos(12)) + c_0^{c_1} \leq x_1;\ c_0 \text{ in } \{1, 2, 3\};\ c_1 \text{ in } \{4\}]$

Note. We extend the work of existing studies and include the possibility to utilize Elementary functions within the term of a predicate (cf. [Lio33b; Lio33c; Lio33a]).

Infinite Coefficients

Infinite Coefficients extend the concept presented by Finite Coefficients and enable the usage of more generic terms. Building upon the structure of Finite Coefficients, predicates using Infinite Coefficients can be described as:

$$[term_i \sim const]$$

with the extension of $term_i$ being an arbitrary algebraic term using Infinite Coefficients. The general concept is to utilize coefficients without a definition and instead use Curve Fitting to heuristically determine their exact substituted value.

Determine Infinite Coefficient Values. In order to find a suitable value for an Infinite Coefficient, we introduce Algorithm 1. The function `getMask(label, Y)`, utilized within Algorithm 1, returns a Label Mask LM = $(\mathrm{lm}_1, \ldots, \mathrm{lm}_{|Y|})$. Depending on $\sim$, the concrete values of lm_i can vary. Let $Y = (y_1, \ldots, y_{|Y|})$ be the tuple representation of all labels within the current data set $\mathcal{D}$ (similar to definition of $\mathcal{D}_Y$) and `label` be an arbitrary member of Y:

- $\sim \in \{\geq, \leq, >, <\}$: lm_i = 1 if `label` equals y_i, else lm_i = -1
- $\sim \in \{=\}$: lm_i = 0 if `label` equals y_i, else lm_i = -1

Running Example. Considering the data set $\mathcal{D}$ depicted in Table 3.1 with $\sim \in \{\leq\}$. Evaluating `getMask`("Apple",$\mathcal{D}_Y$} would return LM = $(1, 1, -1, -1, -1, -1, -1)$.

Algorithm 1 Determine Infinite Coefficient c_i

```
for label in unique($\mathcal{D}_Y$) do
    labelMask ← getMask(label, $\mathcal{D}_Y$)
    curveFit($term_i$, $\mathcal{D}_X$, labelMask)
    collection ← $term_i$
end for

return term in collection with min. Impurity
```

Note. Algorithm 1 computes for every unique label one individual substitution value for the coefficient. In the end, the predicate with the lowest impurity will be returned.

Running Example. The general concept of Infinite Coefficients alone is already capable of replacing and extending complete predicate generator patterns like Axis-Aligned or Linear Classifier [Jac20, Chapter 5.2]. The whole linear classifier strategy for data set 3.1 can be represented by a predicate with the following structure: $[x_0 * c_0 + x_1 * c_1 + x_2 * c_2 + x_3 * c_3 \leq 0]$. Within Chapter 6.5 we recapitulate this concept.

Finite and Infinite Coefficients

This subsection introduces a concept to combine both the usage of Finite and Infinite Coefficients within a predicate of structure:

$$[term_{\mathrm{fi}} \sim const; coef\_def]$$

In general this concept consists of:

1. Processing Finite Coefficients
2. Processing Infinite Coefficients

The first step starts with generating all possible combinations of Finite Coefficients and subsequently substituting these combinations into copies of the original term $term_{\mathrm{fi}}$. Each copy represents one possible unique combination and after the first step, all remaining coefficients will be infinite ones. Therefore, it is possible to apply the concept of Algorithm 1 to each copy.

Note. However, it should be avoided to combine too many Finite and Infinite Coefficients together as that can easily end up in a combinatorial explosion.

4.1.2. Feature Constraints

The concept of *Feature Constraints* is provided for explicit situations where the usage of a predicate is only valid under certain constraints within the current data set. Previously, a predicate could be applied potentially at every state of a data set. Now, Feature Constraints provide the opportunity to restrict certain predicate usage. Hereby, we provide the opportunity to define a valid range for selected features to be in. Only if every value of a selected feature is in this valid range, the predicate can be used. The general structure of predicates using Feature Constraints can be described as the following:

$$[term_{\mathrm{fc}} \sim const; feat\_def]$$

with $term_{\mathrm{fc}}$ being an arbitrary algebraic function. The explicit Feature Constraints are specified in *feat_def*. To reduce the number of definition combinations for an arbitrary feature $\mathrm{x_i}$ and its restriction set $\mathrm{X_i}$, we restrict $\mathrm{X_i}$ to be one single part of its domain space. This work assumes that every definition of $\mathrm{X_i}$ will be given by either a finite set representation with $|\mathrm{X_i}| \in \mathbb{N}$ or an Interval representation (e.g $[d,e)$, $[d,e]$, (d,e), $(d,e]$ with $d,e \in \mathbb{R} \cup "\infty" \cup " - \infty"$).

Running Example. Extending the predicate depicted in Figure 3.1, to be only valid at a "Vitamin C Percentage" greater or equal to 45, one would obtain: $[x_0 \leq 110$; x_1 in $[45, \infty)]$

4.1.3. Further Optimizations

Union Feature Constraints. In order to improve and simplify the explicit definition of sets utilized throughout Feature Constraints (and Finite Intervals), the mathematical Union Operator for sets is provided. Previously several individual predicates were needed to express various different valid Feature Constraint interval options.

Now, one can bundle all this variations into one single *feat_def*. For instance, the variety of valid Feature Constraint Intervals given by X_i, $X'_i, \ldots$, can now be arranged into one single predicate with

$$[\textit{term} \sim \textit{const} \text{ ; } x_i \text{ in } X_i \cup X'_i \cup \ldots]$$

Curve Fitting Strategies. At this time, we are proving four different Curve Fitting strategies, namely *Levenberg-Marquardt*, *Trust Region Reflective*, *Dogleg Algorithm* and *Optimized*.

The Levenberg-Marquardt strategy as mentioned by [Lev44; Mar63] and later improved by [TS12], is the fastest fitting technique and therefore applied the most within this thesis. However, within the chosen implementation of this work this strategy can only be utilized whenever the number of scattered data points is greater or equal to the number of parameters to fit. For further information on the detailed implementation we refer to [NW06, Chapter 10.3] and Chapter 5.

The Trust Region Reflective Approach [GQT66; Sor82] is slower than the Levenberg-Marquardt strategy, but its advantage is that it can potentially compute a fit even if the number of scattered data points is less than the number of parameters to fit.

To combine both the advantages of the Levenberg-Marquardt strategy and the Trust Region Reflective approach, we introduced a new strategy called Optimized. The Optimized strategy compares the number of scattered data points to the number of parameters and utilizes as often as possible the Levenberg-Marquardt strategy. Only in the edge case where the number of data points is less than the number of parameters the Trust Region Reflective Approach will be deployed.

Finally, the Dogleg Algorithm which was first introduced by Michael J. D. Powell in [LA05], builds upon the Levenberg-Marquardt strategy and combines it with explicit aspects of the Trust Region approach. The accuracy of the Dogleg Algorithm is higher than previously presented strategies but it is by far the slowest strategy which is the reason why we refrain from using it.

Curve Fitting Process Optimization. Predicates covered within this work only contain relations $\sim \in \{\geq, \leq, >, <, =\}$. Every single one of these relations partitions its whole input space. Therefore, the Curve Fitting process provides room for optimizations due to the fact, that not a "perfect" fit, to the corresponding label mask, has to be computed. For instance: Let LM = $(1, 1, -1, 1)$ be an exemplary label mask for the predicate p = $[\mathrm{x}_1 - \mathrm{c}_1 \geq 0]$. It is not necessary for the Curve Fitting process to exactly fit to the y-coordinates $(1, 1, -1, 1)$. The label mask LM = $(1, 1, -1, 1)$ is only a representation which describes the destination partition. In other words, every coefficient assignment, which evaluates to a result (a, b, c, d) with $a, b, c, d \in \mathbb{R}$ and $a, b, d \geq 0$ and $c \leq 0$, would be a 100% accurate fit.

4.2. Decision Tree Induction Algorithm

Building upon the introduced Richer Domain Predicates, we propose several different extensions to increase the usability and performance of standard decision tree induction algorithms.

4.2.1. Richer Domain Strategy

The *Richer Domain Strategy* is the standard strategy to process Richer Domain Predicates. The main difference between this decision tree induction algorithm and preexisting algorithms is that the collection of possible predicates $\mathcal{C}$ is already fixed at the beginning. This means that the predicate generator does not have to create the predicates itself. Similar to section 3.2.5, the workflow of the Richer Domain Strategy can be described as the following:

1. Create a node for the current data set $\mathcal{D}'$.
2. Pre-process the predicate collection $\mathcal{C}$.

   Create all combinatorial combinations of Finite Coefficients.

   Apply curve fitting on predicates with Infinite Coefficients.
3. If every predicate candidate partitions $\mathcal{D}'$ in one single subset,
   (e.g. $\mathcal{D}'_{\text{true}} = \mathcal{D}' \wedge \mathcal{D}'_{\text{false}} = \emptyset$ or $\mathcal{D}'_{\text{false}} = \mathcal{D}' \wedge \mathcal{D}'_{\text{true}} = \emptyset$)
   return the labels of $\mathcal{D}'$.
4. Else use the best predicate candidate according to the chosen impurity measurement and apply the induction algorithm on the sub trees $\mathcal{D}'_{\text{true}}$ and $\mathcal{D}'_{\text{false}}$.

4.2.2. Priority Strategy

The *Priority Strategy* encapsulates the concept of using several different decision tree induction algorithms at the same time. Additionally, every decision tree induction algorithm $(\mathrm{St})_\mathrm{i}$ gets an individual priority $(\mathrm{pr})_\mathrm{i} > 0$ assigned. This corresponding priority is later taken into account when calculating the impurity of the predicate p_i. For an exemplary data set $\mathcal{D}$, every algorithm $(\mathrm{St})_\mathrm{i}$ proposes a predicate p_i. Thereby, the extended impurity is calculated as the following:

$$\text{Impurity}_{\text{new}}(p_i) = \frac{\text{Impurity}(p_i)}{(\mathrm{pr})_\mathrm{i}}$$

After updating the new impurities, the Priority Strategy will use the predicate candidate with the lowest impurity.

Fallback Strategy. The concept of a *Fallback Strategy* extends the Priority Strategy by providing a safety backup predicate. In general, the Fallback Strategy developed out of the idea to use one or several main strategies whenever possible and in the edge case within a node where all main strategies fail[1], the Fallback Strategy suggests an alternative predicate. To maintain consistency within the Priority Strategy, we will assign a Fallback Strategy the exclusive priority of 0.

Note. It is important to make sure that all Fallback predicates with a priority of 0 have to be processed separated from the predicates with a priority > 0, to avoid dividing by zero.

4.2.3. Semi Automatic Strategy

The *Semi Automatic Strategy* comes with its own graphical user interface and the idea is to provide control for the user. Within this graphical user interface, the user can enter and process custom Richer Domain Predicates and manually select the predicates at every node. In addition, this interface also provides suitable alternative predicate options from which the user can choose from. To the current state, following alternatives are offered:

- Axis Aligned Strategy.
- Linear Classifier [Jac20, Chapter 5.2].

[1]A decision tree building strategy fails at a node, when the strategy is not capable of generating possible predicates which reduce the impurity of that node.

- Linear Classifier with the extension of only producing custom linear combinations, specified by the user.

At every node, the user is presented a list of all manually entered predicates as well as the proposed alternatives. This list is sorted by impurity. Then the user can select their preferred split, taking into account both the impurity as well as the explainability of the predicates. This possibility of taking explainability into account is the key advantage of the semi automatic strategy, as this is something that is extremely important, yet almost impossible to judge for a machine.

4.2.4. Further Optimizations

Within this thesis, one of the most computationally intensive processes is to determine the impurity. The reason behind that lies in the massive size of most of the controller data set. Therefore, we introduce the idea of an optimized *tree search* before actually computing the impurity. The idea is to store all used predicates which were used on the direct ancestor path (between the current node and the root node). Through this concept two advantages arise.

1. The impurity calculation of predicates which evaluate to a term that can be found within the ancestor path, can directly be skipped since one predicate cannot reduce the impurity of a consecutive path twice. The reason for that is based on our definition of a predicate in Section 3.2.1. A subset that has already been partitioned by p cannot be partitioned again by p.

2. Predicates which cannot be found within the ancestor path but still do not reduce the impurity at a node will also not reduce the impurity of descendant nodes. Therefore, the impurity calculation for that predicate within descendant nodes can also be skipped. Intuitively, this can be described by the fact that if one predicate p assigns every feature of $\mathcal{D}'$ to the same partition, then it will also assign every sub set of features with $S' \subseteq \mathcal{D}'$ to the same partition.

5. Implementation

Building upon the related concepts presented within Chapter 4, this section examines their concrete implementation. The first section contains an introduction to our relevant and frequently utilized libraries namely *NumPy*, *SymPy* and *SciPy*. The second section provides a general overview of `dtControl` and describes all implemented extensions. All extended and added functionalities for `dtControl` are introduced in their chronological order, as they appear in the extended workflow. We start by introducing our new custom predicate parser and follow up with an introduction to the extended internal predicate object-representation. Finally, we propose several different decision tree induction algorithm extensions, for the preexisting DT Learning component in `dtControl`.

5.1. Libraries

5.1.1. NumPy

NumPy [Oli06] is an open-source Python library providing both efficient functions, as well as data structures to process and store large amounts of sorted data. We used NumPy data structures to store and represent all controllers used within `dtControl`.

5.1.2. SymPy

SymPy [Meu+17] is an open-source Python library, which enables symbolic arithmetic computation. The concept of symbolic computation is to use symbols as representation for mathematical objects instead of numerical terms. Every term will remain in its unevaluated symbolic form. Therefore, every mathematical object is represented exactly until we explicitly evaluate the term to its numerical value [Symb].
Frequently used features. Apart from the common usage of SymPy as a library to represent arbitrary terms, especially the features `Sympify` [Syma] and `Lambdify` [Symc] were used. `Sympify` provides a parser to convert mathematical expressions presented within a string into processable SymPy object-representations. `Lambdify` is widely used within the context of Curve Fitting. It provides the efficient functionality to convert a SymPy Expression into a lambda function.

5.1.3. SciPy

SciPy [Vir+20] is an open-source Python library which builds upon the NumPy library and provides several different numerical and mathematical algorithms for scientific computing.
Frequently used features. SciPy's curve fitting implementation called `curve_fit`, is extensively used within the context of Curve Fitting due to its robustness and user friendliness.

5.2. dtControl

`dtControl` as presented by [Ash+20a] and [Jac20] is an open-source tool which can be applied on synthesized controllers. The general concept encapsulates the idea to make use of the in many ways beneficial data structure decision tree and use it for the controller representation. `dtControl` thereby provides the unique opportunity to translate the advantages of decision trees [Ash+20a; Brá+15] to controllers. This results in:

1. Hierarchical decision making support, resulting in an efficient controller representation which reduces the overall controller size.
2. Enabling the usage of decision tree learning techniques to increase the understandability of the overall modeled controller and thereby increasing the confidence of controller-correctness.

Workflow. The general workflow of `dtControl` as presented by [Ash+20a; Jac20], is depicted in Figure 5.1. Here, `dtControl` accepts one argument as input which can be of several different types. Typically, the input argument represents a controller, synthesized by tools like `SCOTS` [RZ16], Uppaal Stratego [Dav+15] or `PRISM` [KNP11]. After the controller input is provided, the user can select from several different predefined decision tree induction algorithms [Jac20, Chapter 4.2]. Afterwards, `dtControl` starts to construct the decision tree which represents the input controller. The component "DT Learning" in Figure 5.1, represents the decision tree induction algorithm, as defined in section 3.2.5. Respectively, the component "Predicates" represents section 3.2.1 and the component "Impurity measures" represents section 3.2.2. Note that `dtControl` acts as a closed system which does only take several inputs at the beginning. Especially, there is no possibility to directly influence the DT Learning process. In other words, there is no chance to manipulate the decision tree while it is being built. It describes a closed workflow where after the input phase one final output will be generated. The generated output consists of a DOT and a C file which encodes the decision tree representation of the input controller. Additionally, several benchmarks are presented.

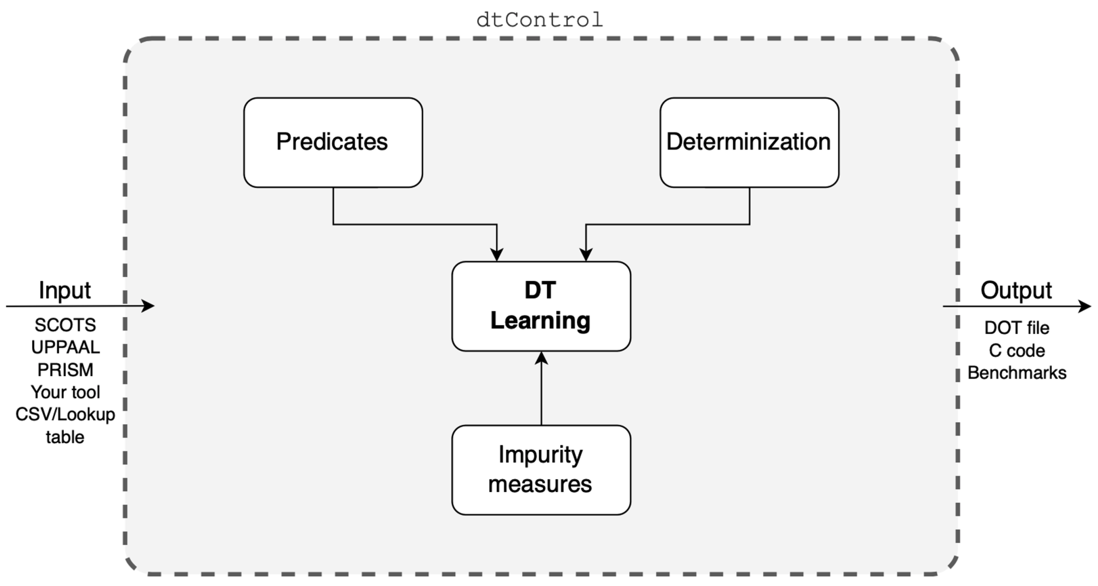


Figure 5.1.: Overview of the workflow of `dtControl`. Graphic extracted from [Jac20, Chapter 4.2.1] which was adapted from [Ash+20b].

Extended Workflow. To actually make use of the introduced theoretical concepts presented in Chapter 4, the general workflow of `dtControl` was improved. The improved workflow of `dtControl` is depicted in Figure 5.2. The blue color hereby represents extended or added components. The main extension comes with the user interface. Through this interface, a user can directly influence the DT Learning component or directly add custom predicates as presented by Chapter 4. Thereby, it is possible for a user to fully control the exact decision tree induction process during the constructing of the final decision tree. The following sections cover all extended and added functionalities for `dtControl` in their chronological order, as they appear in the extended workflow.

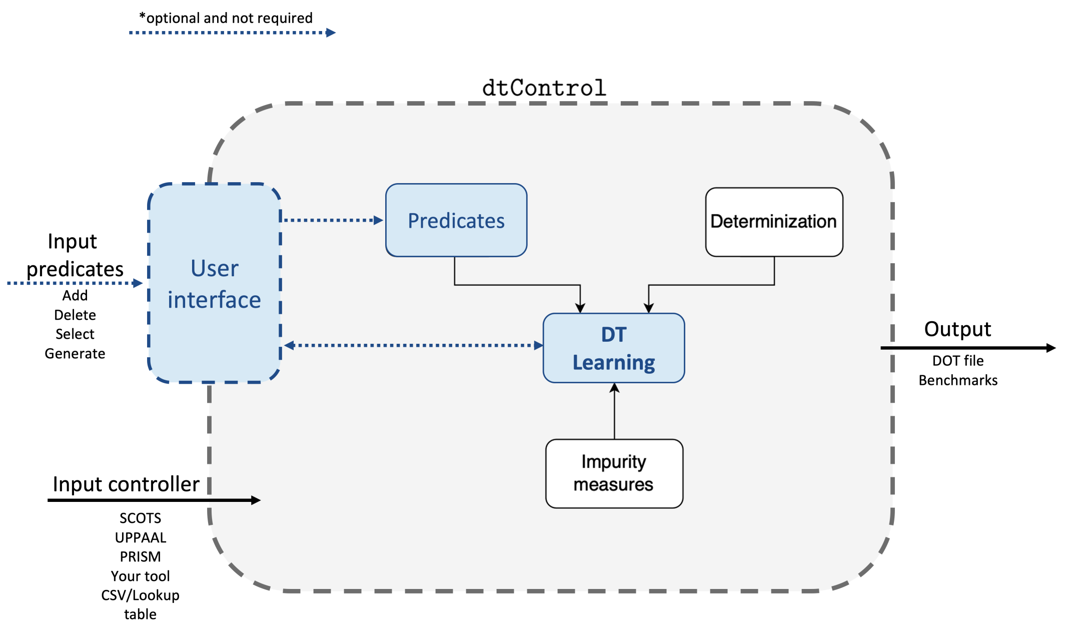


Figure 5.2.: Overview of the extended workflow of `dtControl`. The blue color represents extended or added components. Graphic based on [Jac20, Chapter 4.2.1] which was adapted from [Ash+20b].

5.2.1. Predicate Parser

The largest component within the extended user interface is a new predicate parser. The parser converts single expressions represented by a string into processable predicate-objects which can be used throughout the DT Learning component. The parser accepts all Predicate Tailoring concepts, presented in Chapter 4.1. The accepted predicate structure can be summarized as the following:

$$[term \sim const; def]$$

with *term* being an arbitrary arithmetic term (also possibility for Elementary functions), $\sim \in \{\leq, \geq, >, <, =\}$ and *def* containing all definitions for utilized Finite/Infinite Coefficients or Feature Constraints within *term*. In order to simplify the parsing process of *def*, we defined a grammar G which represents all accepted definitions by the parser. The explicit grammar can be found in the developer documentation of `dtControl`[1]. Every single parsed predicate is represented by an object called "RicherDomainPredicate".

[1]https://dtcontrol.readthedocs.io/en/latest/

The overall design goal for the "RicherDomainPredicate" class was to build upon the preexisting "Split" class (depicted in Figure 5.3) and extend it with the concepts of Predicate Tailoring.

Note. Additional design goals of the predicate parser are its user friendliness and an extensive input validation. To ensure the user friendliness, the predicate parser reacts to over 30 common structural mistakes individually and presents the explicit invalid part. Furthermore, to ensure the general correctness of the predicate parser, we created several different test cases for every individual parser functionality.

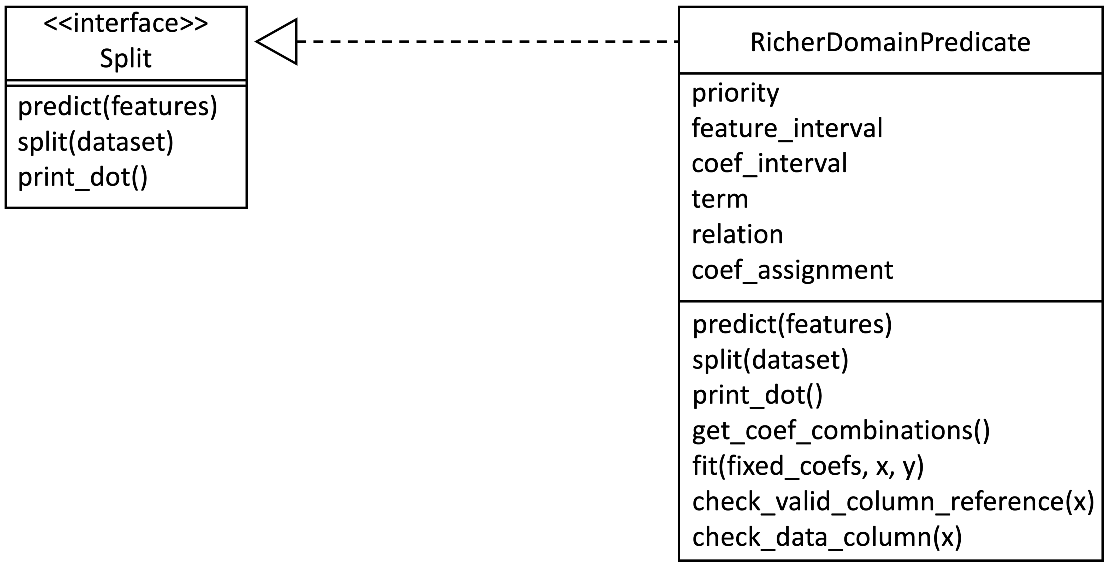
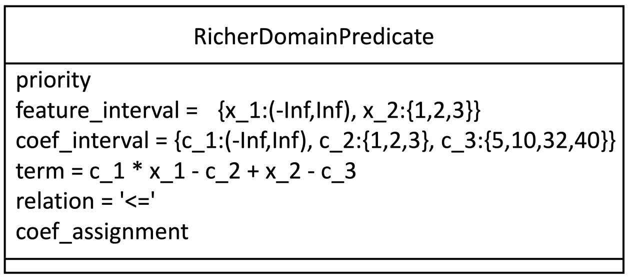


Figure 5.3.: A schematic overview of the new introduced "RicherDomainPredicates" class, implementing/extending the preexisting "Split" class. ("Split" class extracted from [Jac20, Chapter 6.3.2]).

Example. The following example examines the correct usage of the "RicherDomain-Predicate" attributes. Consider an exemplary predicate p, containing all presented concepts within the field of Predicate Tailoring:

$$p = [c_1 * x_1 - c_2 + x_2 - c_3 \leq 0; x_2 \text{ in } \{1,2,3\}; c_2 \text{ in } \{1,2,3\}; c_3 \text{ in } \{5,10,32,40\}]$$



```
RicherDomainPredicate
priority
feature_interval =   {x_1:(-Inf,Inf), x_2:{1,2,3}}
coef_interval = {c_1:(-Inf,Inf), c_2:{1,2,3}, c_3:{5,10,32,40}}
term = c_1 * x_1 - c_2 + x_2 - c_3
relation = '<='
coef_assignment
```

Figure 5.4.: Exemplary "RicherDomainPredicate" class representing predicate p.

In Figure 5.4, the exact "Richer-DomainPredicate" representation for p is displayed. The attribute "feature_interval" contains all Feature Constraints as presented in Chapter 4.1.2. To symbolize the absence of Feature Constraints we assign the infinity interval for unconstrained features. The attribute "coef_interval" contains all coefficients used throughout p, with their corresponding intervals. Note that in order to differentiate a Finite Coefficient from an Infinite Coefficient, we directly assign the Infinity Interval to Infinite Coefficients. The "term" attribute contains the actual term used within the predicate p. All coefficients, terms and intervals are directly represented as SymPy objects to provide an increased compatibility later on. The "priority" attribute enables the usage of the Priority Strategy and contains by default the value 1. Its theoretical concept is covered in section 4.2.2. In the following section 5.2.2, we present its actual usage. The "coef_assignment" attribute is assigned during the execution of the Richer Domain Strategy and will later contain the actual selected/calculated values for the Finite and Infinite Coefficients.

5.2.2. DT Learning

Within this chapter, all extensions regarding the DT Learning component are covered. The "Automatic" subsection thereby contains all decision tree induction algorithms which follow a linear and automatic workflow. The Semi Automatic subsection contains the introduced concept of section 4.2.3 and combines all presented concepts within this thesis.

Automatic

Richer Domain Strategy. Similar to section 4.2.1, the Richer Domain Strategy describes the most general decision tree induction algorithm to automatically process Richer

Domain Predicates. However, this strategy in combination with the predicate parser is already capable of processing complete handcrafted strategies and utilize them to represent an input controller. The general workflow of the Richer Domain Strategy can be described as the following:

1. Pre-process the predicate collection $\mathcal{C}$, presented by the predicate parser.

    Create all combinatorial combinations of Finite Coefficients by executing `get_coef_combinations()`.

    Apply curve fitting on predicates with Infinite Coefficients by executing `fit(fixed_coefs`[2]`, x, y)`.

    Store the current coefficient assignment in the attribute "coef_assignment".

2. Return the best predicate candidate according to the chosen impurity measurement.

Limitations. It should be noted that in order to use the (automatic) Richer Domain Strategy as only decision tree induction algorithm, a complete collection of predicates (domain knowledge) is required at startup of the system. However, the problem is that most of the time only a limited amount of domain knowledge is available. In order to overcome this problem we provide the Priority Strategy.

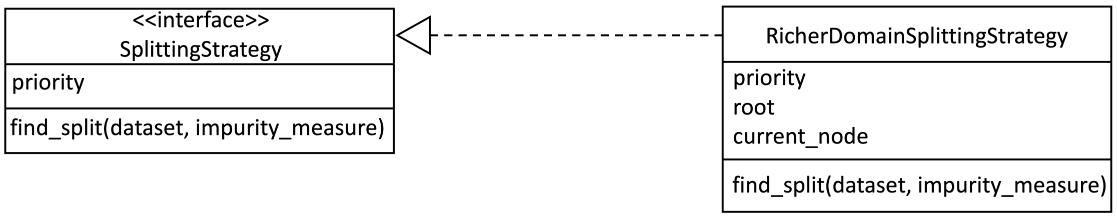
Priority Strategy. The Priority Strategy, as presented in Chapter 4.2.2, encapsulates the concept of combining different decision tree induction algorithms and assigning each algorithm one priority. This priority will later be taken into account when calculating the impurities of their corresponding predicates. It should be kept in mind, that the exclusive priority of 0 should only be assigned to fallback strategies which are only used when every other strategy with priority > 0 fails.

Note. The key advantage of this strategy is that one could easily and automatically "enrich" every arbitrary decision tree induction algorithm with custom predicates.

Further Optimizations. To provide a general comparability with the Priority Strategy and the concept of the optimized tree search presented in section 4.2.4, we also extended the preexisting "SplittingStrategy" class (depicted in Figure 5.5). The "priority" attribute thereby represents the assigned priority value for the Priority Strategy. The "root" attribute contains a reference to the root node of the current DT which is being built.

[2]The fit functionality for Richer Domain Predicates can directly be called on top of a combinatorial combination.

In order to generate the ancestor path needed for the optimized tree search (section 4.2.4), the current node is stored in the attribute "current_node".

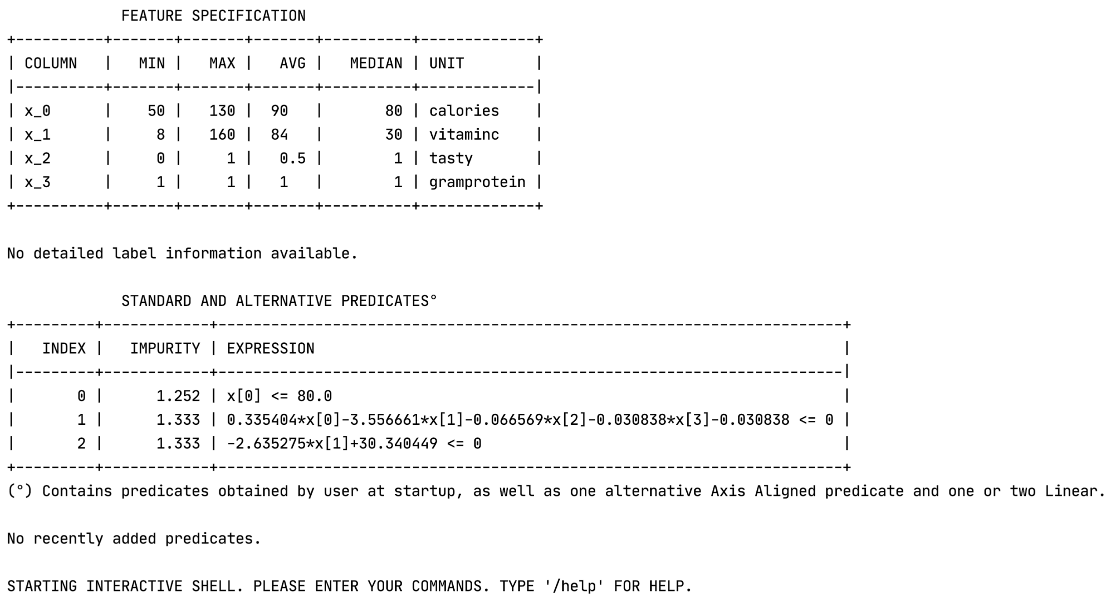


<<interface>>
SplittingStrategy
priority
find_split(dataset, impurity_measure)
RicherDomainSplittingStrategy
priority
root
current_node
find_split(dataset, impurity_measure)

Figure 5.5.: A schematic overview of the new introduced "RicherDomainPredicates" class, implementing/extending the preexisting "SplittingStrategy" class. ("SplittingStrategy" class extracted from [Jac20, Chapter 6.3.2]).

Semi Automatic

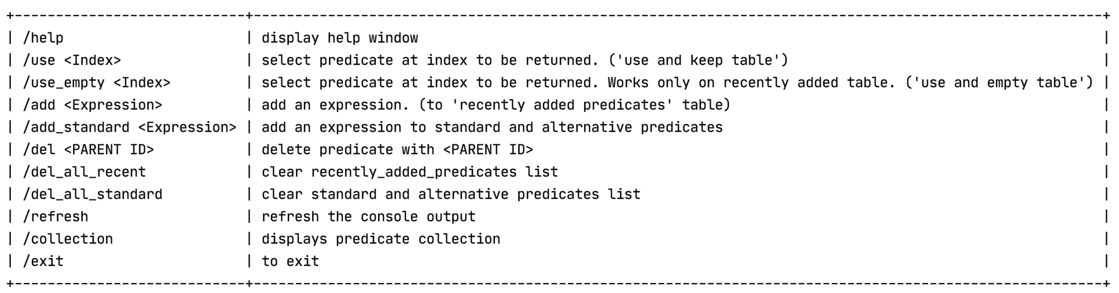
The Semi Automatic Strategy, as theoretically presented in section 4.2.3, comes with its own command line interface. An exemplary state of the interface for the Semi Automatic Strategy, applied on the data set of Table 3.1, is depicted in Figure 5.6.



```
             FEATURE SPECIFICATION
+----------+-------+-------+-------+----------+-------------+
| COLUMN   |   MIN |   MAX |   AVG |   MEDIAN | UNIT        |
|----------+-------+-------+-------+----------+-------------|
| x_0      |    50 |   130 |  90   |       80 | calories    |
| x_1      |     8 |   160 |  84   |       30 | vitaminc    |
| x_2      |     0 |     1 |   0.5 |        1 | tasty       |
| x_3      |     1 |     1 |  1    |        1 | gramprotein |
+----------+-------+-------+-------+----------+-------------+

No detailed label information available.

             STANDARD AND ALTERNATIVE PREDICATES°
+---------+------------+------------------------------------------------------------------------+
|   INDEX |   IMPURITY | EXPRESSION                                                             |
|---------+------------+------------------------------------------------------------------------|
|       0 |      1.252 | x[0] <= 80.0                                                           |
|       1 |      1.333 | 0.335404*x[0]-3.556661*x[1]-0.066569*x[2]-0.030838*x[3]-0.030838 <= 0 |
|       2 |      1.333 | -2.635275*x[1]+30.340449 <= 0                                          |
+---------+------------+------------------------------------------------------------------------+
(°) Contains predicates obtained by user at startup, as well as one alternative Axis Aligned predicate and one or two Linear.

No recently added predicates.

STARTING INTERACTIVE SHELL. PLEASE ENTER YOUR COMMANDS. TYPE '/help' FOR HELP.
```

Figure 5.6.: An exemplary state of the semi-automatic command line interface, obtained with the data set of Table 3.1.

At the top of the command line interface, current feature information is displayed. If there is additional information about labels available, a similar information table for labels will be displayed. The "Standard And Alternatives" section at the bottom is hereby the most important. This table displays all alternative predicates proposed by the Axis Aligned Strategy (section 3.2.4), the Linear Classifier [Jac20, Chapter 5.2] and the extended Linear Classifier (section 4.2.3). At this point of the command line interface we accept several different commands, as depicted in figure 5.7:



```
+----------------------------+-----------------------------------------------------------------------------------------------------+
| /help                      | display help window                                                                                 |
| /use <Index>               | select predicate at index to be returned. ('use and keep table')                                    |
| /use_empty <Index>         | select predicate at index to be returned. Works only on recently added table. ('use and empty table') |
| /add <Expression>          | add an expression. (to 'recently added predicates' table)                                           |
| /add_standard <Expression> | add an expression to standard and alternative predicates                                            |
| /del <PARENT ID>           | delete predicate with <PARENT ID>                                                                   |
| /del_all_recent            | clear recently_added_predicates list                                                                |
| /del_all_standard          | clear standard and alternative predicates list                                                      |
| /refresh                   | refresh the console output                                                                          |
| /collection                | displays predicate collection                                                                       |
| /exit                      | to exit                                                                                             |
+----------------------------+-----------------------------------------------------------------------------------------------------+
```

Figure 5.7.: Help window of the semi-automatic command line interface, displaying all possible commands.

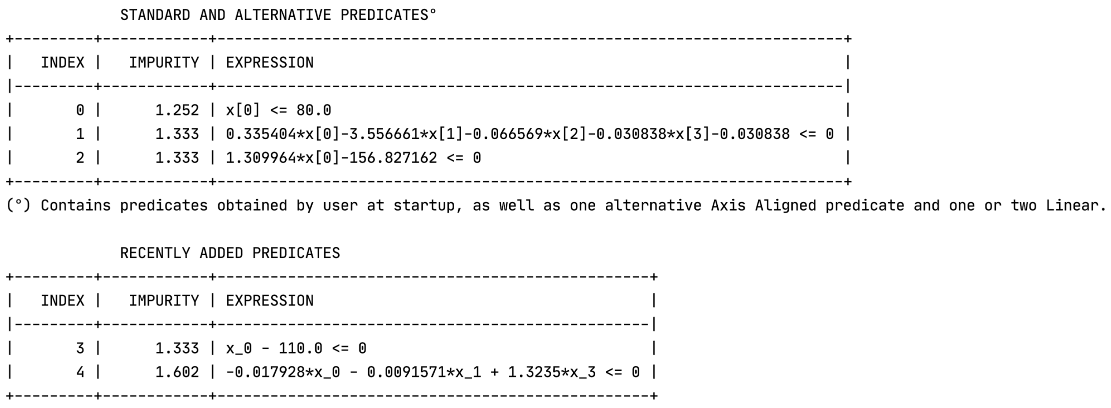
One of the most used commands at this state would be `/add <Expression>`. This command allows the user to directly call the predicate parser to parse an expression.

Running Example. Consider the example predicates $p_1 = [\text{Calories} \leq 110]$ and $p_2 = [-0.017928 * \text{Calories} - 0.0091571 * \text{VitaminC} + 1.3235 * \text{Protein} \leq 0]$, depicted in Figure 3.1 and Figure 3.2. To parse these predicates, the user must simply enter `/add` followed by the predicate expression. The result of the commands is depicted in Figure 5.8. The concrete commands used in the example:

- `/add x_0 <= 110`
- `/add -0.017928 * x_0 - 0.0091571 * x_1 + 1.3235 * x_3 <= 0`

Note. We only allow feature references of structure "x_i" with $i \in \mathbb{N}_0$ (see Definition 3.2.1) since not every controller data set contains specific feature/label names. Analog structure for coefficients "c_i" with $i \in \mathbb{N}_0$.

To finally select a predicate, the user can choose from several commands, as depicted in Figure 5.7. Thereby, the simplest option is the `/use <Index>` command. With this command the user can manually select an index from the list and the corresponding predicate is returned/used for the current data set. In conclusion, the semi automatic strategy provides the possibility to generate completely individual decision trees for an input controller.



```
             STANDARD AND ALTERNATIVE PREDICATES°
+---------+------------+-----------------------------------------------------------------------+
|   INDEX |   IMPURITY | EXPRESSION                                                            |
|---------+------------+-----------------------------------------------------------------------|
|       0 |      1.252 | x[0] <= 80.0                                                          |
|       1 |      1.333 | 0.335404*x[0]-3.556661*x[1]-0.066569*x[2]-0.030838*x[3]-0.030838 <= 0 |
|       2 |      1.333 | 1.309964*x[0]-156.827162 <= 0                                         |
+---------+------------+-----------------------------------------------------------------------+
(°) Contains predicates obtained by user at startup, as well as one alternative Axis Aligned predicate and one or two Linear.

             RECENTLY ADDED PREDICATES
+---------+------------+-------------------------------------------------+
|   INDEX |   IMPURITY | EXPRESSION                                      |
|---------+------------+-------------------------------------------------|
|       3 |      1.333 | x_0 - 110.0 <= 0                                |
|       4 |      1.602 | -0.017928*x_0 - 0.0091571*x_1 + 1.3235*x_3 <= 0 |
+---------+------------+-------------------------------------------------+
```

Figure 5.8.: Semi-automatic command line interface after parsing the predicates of Figure 3.1 and Figure 3.2.

5.2.3. Documentation

Throughout the development of the presented extensions several different test cases were created. Additionally, very detailed logger statements (including information statements) were used throughout the code. For more information on test cases, logger or general implementation details we refer to the developer documentation of `dtControl`[3].

[3]https://dtcontrol.readthedocs.io/en/latest/

6. Evaluation

This chapter contains the evaluation of our presented concepts. We examine important capabilities of `dtControl` and also point out its limitations. We start by introducing the controller model we mainly used for evaluation. Within this section, we also comment on a preexisting work to which we compare our current version of `dtControl`. The following chapter evaluates the Optimized Tree Search technique, presented in Section 4.2.4. Additionally, we also evaluate the performance of Infinite Coefficients, by using an increasing number of Infinite Coefficient within terms of a predicate. Finally, the last chapter concludes the evaluation by pointing out a limitation of Infinite Coefficients, which are present when working with significantly large data sets. For general time tracking we used the benchmark subsystem introduced by [Jac20, Chapter 6]. All evaluations were performed on a machine with a 2.9 GHz 6-Core Intel Core i9 processor and 32 GB 2400 MHz DDR4 RAM.

6.1. Cruise Control

Cruise Control as presented by [LMT15], describes a common problem in adaptive cruise control. Two cars called *Ego* and *Front* drive behind each other along a one-lane road, as depicted in Figure 6.1.

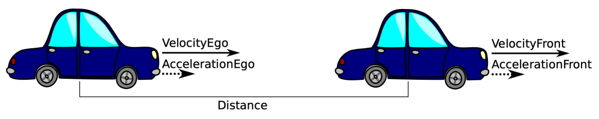


Figure 6.1.: Exemplary overview of the Cruise Control model. Figure extracted from [LMT15].

We are in control of Ego and our goal is avoid crashing into the car in front of us (Front). Therefore we try to maintain a safety distance of at least 5. Depending on given environment variables like distance, velocity or acceleration of both cars, we can choose the acceleration of our own car, Ego. However, while generating this controller, we can specify different individual min/-max ranges within the environment-variables (velocity, acceleration, sensor distance). Therefore, different controller sizes were generated, to evaluate the performance of our proposed theory. Additionally, Cruise Control was used throughout a preexisting work

[Akm19] which introduced a similar prototype to include domain knowledge into the decision tree induction algorithm. In order to enable a higher compatibility with the preexisting work, we have chosen to utilize the same controller as input, namely Cruise Control.

Handcrafted Strategy. Within the preexisting work of mentioned prototype, [Akm19, Chapter 5.2] also introduces a specific handcrafted strategy for Cruise Control. This handcrafted strategy can be described as a set of 4 predicates, which were specifically constructed for Cruise Control, in order to represent this controller as small and understandable as possible. We therefore utilized this handcrafted strategy frequently within the evaluation of our own concepts. The detailed handcrafted strategy of [Akm19] can be found in the appendix A.1.

6.2. Preexisting Prototype

Within this section, we evaluate the execution time of the Richer Domain Strategy (optimized version) against the preexisting prototype of [Akm19]. Both `dtControl` and the prototype are using the same handcrafted strategy presented in [Akm19, Chapter 5.2]. In [Akm19], the prototype was evaluated on several different sizes of Cruise Control. The best execution times (in seconds) are summarized in Table 6.1.

Table 6.1.: Summarized evaluation results of the prototype in [Akm19].

Name	Min velocity	Max velocity	Size[1]	Time (s)
Cruise_medium	0	8	48 781	42.00
Cruise_large	−6	16	130 269	128.00

Unfortunately, we were not able to replicate the exact same model with the same number of state-action pairs as in [Akm19], due to the fact that several other variables were not described (e.g. sensor distance). Therefore, we set the values we knew from that work and then overapproximated the size of the controller. The exact model specification we used in our evaluation can be found in the appendix A.3. The obtained execution times (in seconds) are depicted in Table 6.2.

[1]Number of state-action pairs

Table 6.2.: Obtained result for comparison with prototype of [Akm19].

Name	Min velocity	Max velocity	Size[2]	Time (s)
Cruise_medium	0	8	262 251	13.43
Cruise_large	−6	16	390 213	15.56

Results. Even tough our input controllers were at least twice as big as the input in [Akm19], we were still able to outperform the prototype. In detail, we decreased the run time of the handcrafted strategy on Cruise_medium by 68% and the run time on Cruise_large by 88%.

6.3. Optimization

Within this section, we evaluate the execution time of our optimization technique called Optimized Tree Search, defined in section 4.2.4. We compare an optimized version of the Richer Domain Strategy which uses the Optimized Tree Search technique with an "un-optimized" version of the Richer Domain Strategy which does not use this technique. To increase the reliability of the comparison, we generated 5 different versions of the Cruise Control model with each having a different size. To reproduce our obtained results, we provide detailed information about our used models in Table 6.3.

Table 6.3.: Overview of all generated Cruise Control versions.

Name	Sensor distance	Min velocity	Max velocity	Size[3]
Cruise_150	150	−6	16	390 213
Cruise_200	200	−6	16	562 488
Cruise_500	500	−6	16	1 602 891
Cruise_800	800	−6	16	2 643 291
Cruise_1100	1100	−6	16	3 683 691

In Table 6.4, we compare the execution time of the "un-optimized" strategy to the execution time of the optimized strategy. As input predicates for the Richer Domain

[2]Number of state-action pairs
[3]Number of state-action pairs

Strategy, we use the handcrafted strategy of [Akm19]. The execution time is hereby measured in seconds.

Table 6.4.: Execution time overview of different sizes of Cruise Control, depending on sensor distance.

Strategy	Cruise_150	Cruise_200	Cruise_500	Cruise_800	Cruise_1100
un-optimized	20.47	28.51	82.82	141.25	193.02
optimized	15.56	21.97	77.03	129.24	179.76

Results. Note that the optimized Richer Domain Strategy in combination with the handcrafted strategy is capable of processing over 3.5 Million state-action pairs in under 3 minutes. Unfortunately we only had one handcrafted strategy available which typically generates a decision tree with 11-13 nodes. In theory however, the Optimized Tree Search is expected to increase the performance even more once there are more predicates and a tree with a sufficient height. Within our evaluation, the Optimized Tree Search always decreased the overall run time. However, due to the fact that our selected handcrafted strategy only consists of 4 predicates which mostly generate a decision tree with 11-13 nodes, the run time reduction does not have a significant impact. On Cruise_150 for example, we were able to reduce the run time by at least 24%. Due to the small number of predicates, we only achieved a run time reduction of 6% on Cruise_1100.

6.4. Infinite Coefficient Performance

Within this section, we evaluate the performance of Infinite Coefficients. For this task, we delete an increasing number of expressions within the predicates of the handcrafted strategy of Cruise Control and substitute them with Infinite Coefficients. This concept is evaluated on "Cruise_medium" (with 262251 state-action pairs, see section 6.2) and the resulting execution time is measured in seconds. The predicates which were used throughout this section can be found in the appendix A.2.

Results. The results depicted in Table 6.5 clearly emphasize once again that an increasing number of infinite coefficients is also associated with an increasing run time. Within our evaluation we obtained on average around 30%-47% run time increase per Infinite Coefficient. Additionally, an increase of Finite Coefficient also resulted in a bigger decision tree with more nodes. Nevertheless, in the last column it can be seen that despite five deleted expressions the number of nodes is very low considering that "Cruise_medium" contains 262251 state-action pairs.

Table 6.5.: Comparison of number of Infinite Coefficients and execution time.

Coefficients	Nodes	Inner Nodes	Time (s)
0	13	6	13.43
1	37	18	55.39
2	37	18	59.40
3	47	23	101.45
4	47	23	107.79
5	99	49	140.26

6.5. Limitations

Within this final section, we recapitulate the range of capabilities and limitations for the Richer Domain Strategy. Consider the running example data set presented in Table 3.1. To imitate the Axis Aligned Strategy (3.2.4) on the mentioned data set[4], we could simply provide the following predicates to the Richer Domain Strategy:

- $[x\_0 <= c_0]$
- $[x\_1 <= c_1]$
- $[x\_2 <= c_2]$
- $[x\_3 <= c_3]$

Results. Within the following section, we compare the execution time (in seconds) of the dedicated Axis-Aligned implementation with the Richer Domain Strategy (Rebuilt Axis). Additionally to the fruits data (Table 3.1), we evaluate the same concept on "Cruise_medium" (section 6.2) and "Cruise_200" (section 6.3). The results are depicted in Table 6.6. Within the comparison of the fruits case study, the execution time of the dedicated Axis Aligned implementation was 99.9% faster. Nonetheless, we obtained a final decision tree of the same structure and accuracy for both strategies. However, for the comparison of Cruise_medium and Cruise_large, it is important to note that the safety guarantees are not preserved anymore. Here we only obtained an accuracy of

[4]Identical concept can be applied on Cruise Control, by adding remaining axis predicates for "x_4", "x_5" and "x_6"

Table 6.6.: Comparison of dedicated Axis Aligned and Richer Domain Strategy imitating Axis Aligned.

Case Study	Strategy	Nodes	Inner Nodes	Time (s)	Accuracy (%)
Fruits	Dedicated Axis	9	4	0.002	100
	Rebuilt Axis	9	4	7.78	100
Cruise_medium	Dedicated Axis	293	146	2.53	100
	Rebuilt Axis	493	246	**1548.13**	**99.99**
Cruise_200	Dedicated Axis	661	330	6.52	100
	Rebuilt Axis	41 143	571	**3625.68**	**99.99**

99.99%. Additionally, the execution time of the dedicated Implementations were 99% faster.

Limitations. Especially while working on controllers with several millions of state action pairs, it is important to avoid using Infinite Coefficients alone. Although it is theoretically possible to use predicates which only consist of one Infinite Coefficient, as seen in the fruits case study, this concept should be used with caution. The reason for that can be found within the curve fitting algorithm. As seen in Section 6.4, a reasonable amount of Infinite Coefficients can be used within an actual term. However, a term should definitely not only consist of one Infinite Coefficient. Above a certain value of state-action pairs almost every curve fitting algorithm will not be able to compute a fit with a sufficient accuracy and therefore the resulting decision tree will not preserve all guarantees.

7. Future Work

In this chapter we introduce several different concepts and ideas which could potentially improve `dtControl`. The first section proposes an alternative concept based on tree edit distance and syntax tree representation of predicates to automate the process of predicate generation. The second section proposes a new customized Curve Fitting technique based on partitioning. The last section contains two extensions for both coefficient-types. The first of these extensions proposes the concept of interval constraints for Infinite Coefficients and the second one introduces a potentially promising optimization to avoid the combinatorial explosion of Finite Coefficients.

However, these extensions only represent suggestions to get closer to the overall goal of this research branch. In general, the overall goal is to create a tool which represents controllers as decision trees in a both intuitive and space-saving format (while still preserving guarantees). Within this thesis, we proposed a semi-automatic concept which relies on frequent user interaction. Ideally, future work only requires one or two domain expressions at the startup and automatically return this "ideal" decision tree.

7.1. Tree Edit Distance Predicate Generation

Parallel to the presented concepts within this work we followed a completely alternative approach to automatically generate predicates. In general, the idea was that a user presents a collection of general expressions which are relevant within the area of application of the controller model. For example physics equations of motion for the Cruise Control model. Afterwards, the second step consists of generating all possible expressions within a fixed tree edit distance of the syntax tree of mentioned equations. However, one important aspect while generating predicates with the tree edit distance approach is to maintain the type compatibility of the generated expressions. Unfortunately, during the time of this thesis we did not find any suitable tool for checking this compatibility. One very promising approach for the future is presented by a SymPy module which is currently under development. The aim of this module is to provide unit support for the preexisting SymPy library [Symd].

For this alternative approach we already implemented the functionality to create the corresponding syntax tree of a Richer Domain Predicate. Additionally, we provide the functionality to compute the lowest tree edit distance between two Richer Domain Predicates. For this purpose we implemented an adapter interface to make use of the state-of-the-art algorithm for tree edit distance, called APTED [PA16; PA15].

7.2. Specialized Curve Fitting algorithm

To overcome the limitations presented in Chapter 6.5, it might be advantageous to develop a customized Curve Fitting algorithm. Inspired by Chapter 4.1.3, it would be reasonable to create a new metric of accuracy depending on the obtained partitions. The general idea would be to redefine the aim of the Curve Fitting process. Instead of achieving a high accuracy in the sense of Least-Square Error, the accuracy should solely depend on the obtained partitions.

7.3. Coefficients

7.3.1. Interval constraints for Infinite Coefficients

Currently, Infinite Coefficients as presented by Chapter 4.1.1 do not support the possibility to restrict their values to a certain interval. Enabling the possibility to define such constraints could result in a useful extension for this work. During the experimental phase of this work, parallel to the Curve Fitting implementation of Scipy [Vir+20], we investigated an alternative Curve Fitting library called LMFIT [New+14]. LMFIT provides the possibility to define such constraints. However, after several case studies we have made the decision to use the SciPy version, as it was much more flexible, robust and overall easier to connect with the features provided by the SymPy Library. LMFIT required an extensive problem definition in advance to the Curve Fitting process which we mostly could not provide due to the continuously changing structure of terms.

7.3.2. Curve Fitting for Finite Coefficients

To overcome the problem of obtaining an infeasibly large set of potential predicates whenever the user specifies too many or too large sets of possible Finite Coefficient-values, it would be worth investigating whether curve fitting could be advantageous. As a concrete example, consider a predicate p with $p =$ [x_0 < c_0; c_0 in {1,2,...,9999}]. Instead of brute-forcing every combination and calculating its corresponding impurity, we could use curve fitting on p' with $p' =$ [x_0 < c_0′] and substitute c_0 with the member in $\{1, 2, \ldots, 9999\}$ which is the closest to the fitted value of c_0′. In this context it would also be worth considering to determine the best label mask combination in advance instead of applying Curve Fitting to every label mask combination.

8. Conclusion

In this work, we have addressed the problem of small and explainable controller representation. To achieve both of these goals, we have enriched decision trees with algebraic predicates - probably coming from domain knowledge. We have proposed several approaches to leverage domain knowledge given by the user, either automatically or semi-automatic. To demonstrate the advantages and applicability of our approach, we have implemented all concepts in the current version of `dtControl`. Our experiments showed that this approach can yield great gains in both size and explainability; however, the method is still limited in its scalability. This thesis is one step in the direction of automatically compressed and explainable controllers and thus a step leading to a world where we can verify and validate all the computer systems that are present in our day-to-day life.

A. Appendix

A.1. Handcrafted Strategy for Cruise Control

Handcrafted strategy for Cruise Control, extracted from [Akm19]; converted to a valid format, accepted by the predicate parser of Section 5.2.1. "minV" and "maxV" describe the minimum velocity and the maximum velocity of the Cruise Control model.

- $x\_2 + ((-2-2)/(2)) + (x\_5 - x\_3) + ((x\_5 + 1 * (-2)) - (x\_3 + 1 * 2)) * (((\mathrm{minV} - x\_5)/(-2)) - 1) + (((0 - (-2)) * (((\mathrm{minV} - ((x\_3 + 2) - 2 * ((((\mathrm{minV} - x\_5)/(-2)) - 1))))/(-2)))^2)/(2)) + (\mathrm{minV} - ((x\_3 + 1 * 2) + ((((\mathrm{minV} - x\_5)/(-2)) - 1)) * (-2))) * ((\mathrm{minV} - ((x\_3 + 2) - 2 * ((((\mathrm{minV} - x\_5)/(-2)) - 1))))/(-2)) <= 5$

- $x\_2 + ((-2-0)/(2)) + (x\_5 - x\_3) + ((x\_5 + 1 * (-2)) - (x\_3 + 1 * 0)) * (((\mathrm{minV} - x\_5)/(-2)) - 1) + (((0 - (-2)) * (((\mathrm{minV} - ((x\_3 + 0) - 2 * ((((\mathrm{minV} - x\_5)/(-2)) - 1))))/(-2)))^2)/(2)) + (\mathrm{minV} - ((x\_3 + 1 * 0) + ((((\mathrm{minV} - x\_5)/(-2)) - 1)) * (-2))) * ((\mathrm{minV} - ((x\_3 + 0) - 2 * ((((\mathrm{minV} - x\_5)/(-2)) - 1))))/(-2)) <= 5$

- $x\_3 <= \mathrm{minV}$

- $x\_3 <= \mathrm{maxV} - 2$

A.2. Infinite Coefficient Performance - Predicates

The exact predicates used to evaluate the performance of the Infinite Coefficients; converted to a valid format, accepted by the predicate parser of Section 5.2.1. For coefficient usage $i \in \{0, 1, 2, 3, 4, 5\}$ only coefficients "c_j" with j < i were used. The original handcrafted strategy can be found in A.1.

- $x\_2 + c\_0 + (x\_5 - x\_3) + ((x\_5 + 1 * (-2)) - (x\_3 + c\_4)) * (((0 - x\_5)/(-2)) - 1) + (((0 - (-2)) * (((0 - ((x\_3 + 2) - 2 * ((((0 - x\_5)/(-2)) - 1))))/(-2)))^2)/(2)) + (0 - ((x\_3 + 1 * 2) + ((((0 - x\_5)/(-2)) - 1)) * (-2))) * ((0 - ((x\_3 + 2) - 2 * ((((0 - x\_5)/(-2)) - 1))))/(-2)) <= c\_1$

- $x\_2 + c\_2 + (x\_5 - x\_3) + ((x\_5 + 1 * (-2)) - (x\_3 + 1 * 0)) * (((0 - x\_5)/(-2)) - 1) + (((0 - (-2)) * (((0 - ((x\_3 + 0) - 2 * ((((0 - x\_5)/(-2)) - 1))))/(-2)))^2)/(2)) +$

$(0-((x\_3+1*0)+((((0-x\_5)/(-2))-1))*(-2)))*((0-((x\_3+0)-2*((((0-x\_5)/(-2))-1))))/(-2)) <= c\_3$

- $x\_3 <= 0$
- $x\_3 <= 12$

A.3. Cruise Control Model Specification

Exact Cruise Control model specifications used throughout the Evaluation (Chapter 6).

Name	Sensor Distance	Min velocity	Max velocity	Size[1]
Cruise_medium	200	0	8	48781
Cruise_large	150	-6	16	390213
Cruise_150	150	-6	16	390213
Cruise_200	200	-6	16	562488
Cruise_500	500	-6	16	1602891
Cruise_800	800	-6	16	2643291
Cruise_1100	1100	-6	16	3683691

Table A.1.: Overview of all generated Cruise Control versions.

[1]Number of state-action pairs

List of Figures

List of Tables

Bibliography

[Adc77] R. J. Adcock. "Note on the Method of Least Squares." In: *The Analyst* 4.6 (1877), pp. 183–184. ISSN: 07417918.

[ADC78] R. J. ADCOCK. "A problem in least squares." In: *The Analyst* 5.1 (1878), pp. 53–54.

[Akm19] S. M. Akmese. "Generating Richer Predicates for Decision Trees." Bachelorarbeit. Technische Universität München, 2019.

[Arl94] S. L. Arlinghaus. "Practical handbook of curve fitting." In: (1994).

[ARW01] S. J. Ahn, W. Rauh, and H.-J. Warnecke. "Least-squares orthogonal distances fitting of circle, sphere, ellipse, hyperbola, and parabola." In: *Pattern Recognition* 34.12 (2001), pp. 2283–2303. ISSN: 0031-3203. DOI: `https://doi.org/10.1016/S0031-3203(00)00152-7`.

[Ash+19] P. Ashok, J. Kretínský, K. G. Larsen, A. L. Coënt, J. H. Taankvist, and M. Weininger. "SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes." In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*. Ed. by D. Parker and V. Wolf. Vol. 11785. Lecture Notes in Computer Science. Springer, 2019, pp. 147–164. DOI: `10.1007/978-3-030-30281-8\_9`.

[Ash+20a] P. Ashok, M. Jackermeier, P. Jagtap, J. Kretínský, M. Weininger, and M. Zamani. "dtControl: Decision Tree Learning Algorithms for Controller Representation." In: *CoRR* abs/2002.04991 (2020). arXiv: `2002.04991`.

[Ash+20b] P. Ashok, M. Jackermeier, P. Jagtap, J. Kretínský, M. Weininger, and M. Zamani. "dtControl: decision tree learning algorithms for controller representation." In: *HSCC '20: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020*. Ed. by A. Ames, S. A. Seshia, and J. Deshmukh. ACM, 2020, 30:1–30:2. DOI: `10.1145/3365365.3383468`.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008. ISBN: 026202649X.

[Brá+15] T. Brázdil, K. Chatterjee, M. Chmelik, A. Fellner, and J. Kretínský. "Counterexample Explanation by Learning Small Strategies in Markov Decision Processes." In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 158–177. DOI: `10.1007/978-3-319-21690-4\_10`.

[Bre+84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN: 0-534-98053-8.

[CGP01] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4.

[Cha] R. N. Charette. *This Car Runs on Code - IEEE Spectrum*. URL: `https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`.

[Cla+18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. 1st. Springer Publishing Company, Incorporated, 2018. ISBN: 3319105744.

[Col20] Colgate-Palmolive. *Colgate Announces Breakthrough Technology Designed To Revolutionize Oral Health*. Jan. 2020. URL: `https://web.archive.org/web/20200902071858/https://www.prnewswire.com/news-releases/colgate-announces-breakthrough-technology-designed-to-revolutionize-oral-health-300981306.html?tc=eml_cleartime` (visited on 09/02/2020).

[Dav+15] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. "Uppaal Stratego." In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 206–211. DOI: `10.1007/978-3-662-46681-0\_16`.

[DOT20] M. Dias, R. de Oliveira Albergarias Lopes, and A. Teles. "COULD BOEING 737 MAX CRASHES BE AVOIDED? FACTORS THAT UNDERMINED PROJECT SAFETY." In: 8 (Apr. 2020), pp. 187–196. eprint: `https://www.researchgate.net/publication/340621972_COULD_BOEING_737_MAX_CRASHES_BE_AVOIDED_FACTORS_THAT_UNDERMINED_PROJECT_SAFETY`.

[Dun] B. Dunbar. *Shuttle Computers Navigate Record of Reliability*. URL: `https://web.archive.org/web/20200819123136/https://www.nasa.gov/mission_pages/shuttle/flyout/flyfeature_shuttlecomputers.html` (visited on 08/19/2020).

[FN] C. for Food Safety and A. Nutrition. *Nutrition Information for Raw Fruits, Vegetables, and Fish*. URL: `http://web.archive.org/web/20200902073917/https://www.fda.gov/food/food-labeling-nutrition/nutrition-information-raw-fruits-vegetables-and-fish` (visited on 08/07/2020).

[GQT66] S. M. Goldfeld, R. E. Quandt, and H. F. Trotter. "Maximization by Quadratic Hill-Climbing." In: *Econometrica* 34.3 (1966), pp. 541–551. ISSN: 00129682, 14680262.

[Guj70] D. Gujarati. "Use of Dummy Variables in Testing for Equality between Sets of Coefficients in Two Linear Regressions: A Note." In: *The American Statistician* 24.1 (1970), pp. 50–52. DOI: `10.1080/00031305.1970.10477181`. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/00031305.1970.10477181`.

[HKS93] D. G. Heath, S. Kasif, and S. Salzberg. "Induction of Oblique Decision Trees." In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*. Ed. by R. Bajcsy. Morgan Kaufmann, 1993, pp. 1002–1007.

[Jac20] M. Jackermeier. "dtControl: Decision Tree Learning for Explainable Controller Representation." Bachelorarbeit. Technische Universität München, 2020.

[JZ17] P. Jagtap and M. Zamani. "QUEST: A Tool for State-Space Quantization-Free Synthesis of Symbolic Controllers." In: *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*. Ed. by N. Bertrand and L. Bortolussi. Vol. 10503. Lecture Notes in Computer Science. Springer, 2017, pp. 309–313. DOI: `10.1007/978-3-319-66335-7\_21`.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems." In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[Kol84] W. Kolb. *Curve Fitting for Programmable Calculators*. Syntec, Incorporated, 1984. ISBN: 9780943494029.

[LA05] M. L. A. Lourakis and A. A. Argyros. "Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment?" In: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. Vol. 2. 2005, 1526–1531 Vol. 2.

[LeF18] D. LeFebvre. *Photo by Dan LeFebvre on Unsplash*. Dec. 2018. URL: https://web.archive.org/web/20200902073526/https://unsplash.com/photos/mAwE-fqgDXc (visited on 09/02/2020).

[Lev44] K. Levenberg. "A METHOD FOR THE SOLUTION OF CERTAIN NON – LINEAR PROBLEMS IN LEAST SQUARES." In: *Quarterly of Applied Mathematics* 2 (1944), pp. 164–168.

[Lio33a] J. Liouville. "Note sur la détermination des intégrales dont la valeur est algébrique." In: *Journal für die reine und angewandte Mathematik* 10 (1833), pp. 347–359. eprint: http://gdz.sub.uni-goettingen.de/en/dms/loader/img/?PID=GDZPPN002139332.

[Lio33b] J. Liouville. "Premier mémoire sur la détermination des intégrales dont la valeur est algébrique." In: *Journal de l'École Polytechnique* 14 (1833), pp. 124–148. eprint: https://gallica.bnf.fr/ark:/12148/bpt6k433678n/f127.item.r=Liouville.

[Lio33c] J. Liouville. "Second mémoire sur la détermination des intégrales dont la valeur est algébrique." In: *Journal de l'École Polytechnique* 14 (1833), pp. 149–193. eprint: https://gallica.bnf.fr/ark:/12148/bpt6k433678n/f152.item.r=Liouville.

[Lip82] M. Lipow. "Number of Faults per Line of Code." In: *IEEE Trans. Software Eng.* 8.4 (1982), pp. 437–439. DOI: 10.1109/TSE.1982.235579.

[LMT15] K. G. Larsen, M. Mikucionis, and J. H. Taankvist. "Safe and Optimal Adaptive Cruise Control." In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. Ed. by R. Meyer, A. Platzer, and H. Wehrheim. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 260–277. DOI: 10.1007/978-3-319-23506-6_17.

[Mar63] D. W. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." In: *Journal of the Society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441. DOI: 10.1137/0111030. eprint: https://doi.org/10.1137/0111030.

[Meu+17] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. "SymPy: symbolic computing in Python." In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

[Mit97] T. M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[MPM20] J. Mathias, A. Pranav, and W. Maximilian. *dtControl*. 2020. URL: `https://dtcontrol.model.in.tum.de/` (visited on 09/02/2020).

[Mur+93] S. K. Murthy, S. Kasif, S. Salzberg, and R. Beigel. “OC1: A Randomized Induction of Oblique Decision Trees.” In: *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*. Ed. by R. Fikes and W. G. Lehnert. AAAI Press / The MIT Press, 1993, pp. 322–327.

[New+14] M. Newville, T. Stensitzki, D. B. Allen, and A. Ingargiola. *LMFIT: Non-Linear Least-Square Minimization and Curve-Fitting for Python*. Version 0.8.0. Sept. 2014. DOI: `10.5281/zenodo.11813`.

[Num] NumPy. *numpy.polyfit*. URL: `http://web.archive.org/web/20200902074339/https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html` (visited on 08/08/2020).

[NW06] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer New York, 2006. DOI: `10.1007/978-0-387-40065-5`.

[Oli06] T. E. Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[PA15] M. Pawlik and N. Augsten. “Efficient Computation of the Tree Edit Distance.” In: *ACM Trans. Database Syst.* 40.1 (Mar. 2015). ISSN: 0362-5915. DOI: `10.1145/2699485`.

[PA16] M. Pawlik and N. Augsten. “Tree edit distance: Robust and memory-efficient.” English. In: *Information Systems* 56 (2016), pp. 157–173. ISSN: 0306-4379. DOI: `10.1016/j.is.2015.08.004`.

[Qui86] J. R. Quinlan. “Induction of Decision Trees.” In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: `10.1023/A:1022643204877`.

[RM14] L. Rokach and O. Maimon. *Data Mining With Decision Trees: Theory and Applications*. 2nd. USA: World Scientific Publishing Co., Inc., 2014. ISBN: 9789814590075.

[RZ16] M. Rungger and M. Zamani. “SCOTS: A Tool for the Synthesis of Symbolic Controllers.” In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. Ed. by A. Abate and G. E. Fainekos. ACM, 2016, pp. 99–104. DOI: `10.1145/2883817.2883834`.

[Sam] Samsung. *Play music on your Family Hub smart fridge.* URL: `https://web.archive.org/web/20200902072347/https://www.samsung.com/us/support/answer/ANS00062814/` (visited on 09/02/2020).

[Sha48] C. E. Shannon. "A mathematical theory of communication." In: *Bell Syst. Tech. J.* 27.4 (1948), pp. 623–656. DOI: `10.1002/j.1538-7305.1948.tb00917.x`.

[Sor82] D. Sorensen. "Newton's method with a model trust region modification." In: *SIAM Journal on Numerical Analysis* 19 (1982), pp. 409–426.

[Syma] SymPy. *core.sympify.* URL: `http://web.archive.org/web/20200902074429/https://docs.sympy.org/latest/modules/core.html` (visited on 08/08/2020).

[Symb] SymPy. *Introduction- What is Symbolic Computation?* URL: `http://web.archive.org/web/20200617155325/https://docs.sympy.org/latest/tutorial/intro.html` (visited on 08/08/2020).

[Symc] SymPy. *Lambdify.* URL: `http://web.archive.org/web/20200902074528/https://docs.sympy.org/latest/modules/utilities/lambdify.html` (visited on 08/08/2020).

[Symd] SymPy. *Unit systems.* URL: `http://web.archive.org/web/20200902074745/https://docs.sympy.org/latest/modules/physics/units/index.html` (visited on 08/08/2020).

[TS12] M. K. Transtrum and J. P. Sethna. *Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization.* 2012. arXiv: `1201.5885 [physics.data-an]`.

[Vir+20] 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. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, 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 S. 1. 0. Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: `https://doi.org/10.1038/s41592-019-0686-2`.

[Yao92] A. C. Yao. "Algebraic Decision Trees and Euler Characteristics." In: *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992.* IEEE Computer Society, 1992, pp. 268–277. DOI: `10.1109/SFCS.1992.267765`.