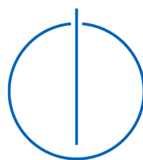
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Learning Algebraic Predicates for Explainable Controllers

Florian Jüngermann



DEPARTMENT OF INFORMATICS

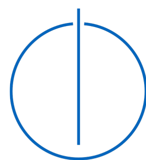
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Learning Algebraic Predicates for Explainable Controllers

Lernen von algebraischen Prädikaten für erklärbare Controller

Author:	Florian Jüngermann
Supervisor:	Univ.-Prof. Dr. Jan Křetínský
Advisor:	M.Sc. Pranav Ashok, M.Sc. Maximilian Weininger
Submission Date:	June 4, 2021



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

Munich, June 4, 2021 Florian Jüngermann

Acknowledgments

I want to thank my supervisor Prof. Dr. Jan Křetínský for introducing me to the field of formal verification and giving me the opportunity to work on this exciting topic.

Then I want to thank my advisors M.Sc. Pranav Ashok and M.Sc. Maximilian Weininger for supporting me along the way. Pranav could always help me when I felt stuck and Maxi's extensive feedback on my drafts greatly improved my thesis' structure and readability.

Finally, I want to thank my family and friends who always supported me. Especially with Tobias Schindler, I could always talk about my ideas and get valuable feedback on whether my explanations are understandable.

Abstract

For safety-critical applications, model checking tools can verify safety criteria of systems and synthesize correct-by-design controllers. Instead of representing these controllers with huge lookup tables or hard-to-understand binary decision diagrams, recent work tried to find succinct and explainable representations with decision trees. To describe complex systems with small trees, recent efforts used more expressive algebraic predicates in the decision nodes. However, until now, there has not been a way of automatically generating those. In this thesis, we propose two solutions: The first one generates predicates from given domain knowledge. The second one learns polynomial expressions from the controller data using support vector machines. We implement the second approach into the open-source tool `dtcontrol` and evaluate it on 28 case studies. For many cases, we see a significant size reduction and for 10 case studies, we even reach the minimum decision tree size.

Contents

Acknowledgments **v**

Abstract **vii**

1. Introduction **1**

2. Related Work **5**

3. Preliminaries **7**
3.1. Controllers 7
3.2. Decision Trees 8
3.2.1. Impurity Measures 9
3.2.2. Predicates 9
3.3. Support Vector Machines 10
3.3.1. Separating Hyperplanes 10
3.3.2. Kernel Trick 12
3.3.3. Computational Complexity 14

4. Motivating Example **15**
4.1. Model Parameters 15
4.2. Problems with the Current Solutions 16
4.3. Handpicked Strategy 16

5. Predicates From Domain Knowledge **19**
5.1. Our Approach 19
5.2. Handcrafted Predicate Derivation 20
5.3. Performance 21
5.4. Identified Problems 22

6. Predicates From Controller Data **25**
6.1. Problems with Curve Fitting 26
6.2. Using Support Vector Machines 27
6.2.1. Problems With Higher Dimensions 27
6.2.2. Reconstructing the Algebraic Decision Function 28
6.3. Feature Importance 28
6.4. Rounding Coefficients 29
6.4.1. Rounding to Zero 30

6.4.2. Scaling the Predicate . . . 30
6.4.3. General Rounding . . . 30
6.4.4. Numerical Errors . . . 31
6.5. Min-label Entropy . . . 31
6.6. Predicate Priority . . . 33

7. Evaluation **35**
7.1. Domain Knowledge Approach . . . 35
7.2. Data-Driven Approach . . . 36
7.2.1. Cruise Control . . . 36
7.2.2. Minimum Tree Size . . . 36
7.2.3. Benchmarks . . . 38
7.2.4. Min-Label Entropy and Predicate Priority . . . 40
7.2.5. Explainability . . . 41

8. Future Work **43**

9. Conclusion **45**

List of Figures **47**

List of Tables **49**

Bibliography **51**

A. Support Vector Machine Details **59**
A.1. Minimization Problem . . . 59
A.2. Lagrangian Function and Dual Problem . . . 60
A.3. Soft-Margin . . . 63

B. The Cruise Control Model **65**
B.1. Cruise Control Modifications . . . 65
B.2. Cruise Control Parameters . . . 65

C. Predicate Generation **67**
C.1. Base Identities . . . 67
C.2. Handcrafted Predicate . . . 67

D. Predicates From Controller Data **69**
D.1. Predicate Without Prettifying . . . 69
D.2. Predicate Without Rounding . . . 69
D.3. Advanced Numerical Precision Problems . . . 69

E. Results **71**

1. Introduction

Today, we cannot imagine a world without software. From large applications with tens of millions of lines of code like an operating system to controllers running on embedded devices found in kitchen tools, we rely on these programs every day. Despite immense efforts, we continually see those programs fail. The impact of such failures can reach from unpleasant, if the thermostat fails to keep the correct temperature [Bil16], to severe in the case of computer-caused power-outages [BBC16], to even fatal in the case of car accidents caused by malfunctioning software [Isi13].

The standard solution to this problem is testing – either as specific unit tests or as end-to-end black-box tests. While testing software is great for finding bugs, in general, it cannot guarantee their absence. However, especially for safety-critical software, we want to be sure that certain criteria are fulfilled. For example, we want to ensure that a car with an emergency braking system will always break early enough to avoid any collisions.

Formal Verification This is where the field of *formal verification* comes into play. We use the general notion of an agent interacting with a surrounding system by performing actions. In the case of the emergency braking system, the agent would be the car we want to control and the available actions are different acceleration values. To verify that a certain property is satisfied, we model the dynamics of the surrounding system formally and express our safety criteria with formalisms such as *temporal logic*. Then, after implementing a control strategy (in short *controller*) for the agent, we can use *model checking* to verify whether the criteria hold.

Having precisely specified the surrounding system and the desired behavior even allows for a more automated approach. Instead of developing a controller by hand and then verifying it, we can take advantage of the recent advances in *controller synthesis* to automatically generate a controller for the agent. The controller ensures that our safety conditions are satisfied by design.

During the synthesis phase, we can usually decide if we want to generate a *deterministic* controller that expresses one way of fulfilling the safety constraints, or if we want to generate the *most-permissive* controller which includes all possible actions that will not violate any constraints. For example, for a thermostat, it might suffice to find a deterministic control strategy to keep the temperature in a given interval. For an emergency braking system, we would probably want to allow the human driver to take any action that does not lead to a safety violation, so we would want to generate the most-permissive controller.

There are multiple state-of-the-art model checking tools with controller synthesis options available, such as `UPPAAL Stratego` [Dav+15], `PRISM` [KNP11], `SCOTS` [RZ16], and `STORM` [Deh+17]. In the most basic form, they use discretization to represent the continuous input space with a finite set of states. For each of those states, the synthesized controller describes which actions are allowed. So, the controller can be expressed explicitly as a lookup table, often with millions of rows.

There are two main problems with this representation. First, storing such a large table can require several hundreds of Megabytes of storage. However, the devices on which the controller should run in the end, mostly are embedded chips with very limited storage capacity. This makes it infeasible to store the entire lookup table on the device.

Second, the sheer size makes it impossible to understand the behavior of the controller. The safety guarantees of the controller rely on the assumption that the formal model was correct and behaves as expected. To validate this, understanding the controller is crucial. For example, a non-permissive controller for the emergency braking system might try to immediately stop the car, thus fulfilling the safety criteria while clearly not being of any help in the real world. These flaws in the model can be detected if we can represent the safe controller in a succinct and explainable way.

Motivating Example In Chapter 4 of this thesis, we have a closer look at the adaptive cruise control model (in short `cruise`) from [LMT15a] which models a simplified emergency braking system for a car. Synthesizing a safe controller with `UPPAAL Stratego` gives us a file with more than six million lines although previous work [Akm19] has shown that there is a way of formulating the safe behavior with a handful of sentences or equations. Our goal is to find such a succinct and explainable representation automatically with techniques from machine learning.

Controller Representation with Decision Trees Recently, significant progress has been made with representing controllers with decision trees [Brá+15; Ash+19a; Ash+19b; Ash+20; Ash+21]. Decision trees [Mit+97] are simple in structure, making them easy to understand, but still expressive enough to represent complex controllers. The open-source tool `dtcontrol` [Ash+21] takes advantage of that and offers an automated way of generating succinct decision trees. It can read controllers from many commonly used model checkers and implements various heuristics to minimize the size of the decision tree.

Traditionally, in a decision tree, the predicate in the decision node has the form $v_i \leq c$ with some state variable v_i and some constant $c \in \mathbb{R}$. Such a split divides up the feature space with a hyperplane orthogonal to the feature axis v_i thus giving it the name *axis-aligned* split [Mit+97]. Those splits are easy to understand and efficient to find, forming the basis of the decision tree learning.

However, axis-aligned predicates are incapable of capturing more complex relationships as seen in Figure 1.1a. In this toy example, 5 predicates are needed to separate the red from the blue labels. For a real-world dataset with thousands of data points,

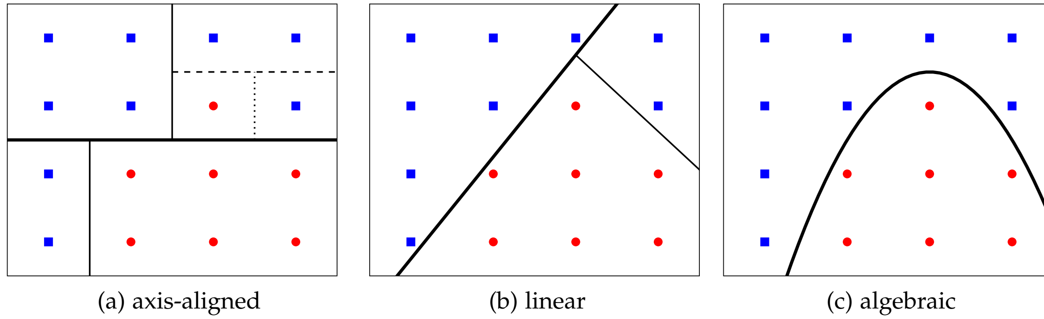


Figure 1.1.: Example showing how different types of predicates can separate a dataset.

this behavior can be even more extreme. For that reason, `dtcontrol` also supports *linear predicates* as proposed by [MKS94]. These splits are still hyperplanes but now can have arbitrary orientations (see Figure 1.1b). This makes the predicates harder to find, more difficult to comprehend as more variables are involved, but can ultimately give significantly smaller decision trees.

Extending the notion of using more complex decision predicates, the newest version of `dtcontrol` allows the use of *algebraic predicates* [Wei20; Ash+21] (see Figure 1.1c). A domain expert can provide arbitrary closed-form mathematical expressions that are then used in the decision tree construction. It is even possible to leave some constants unspecified and `dtcontrol` will find suitable values for those.

Limitations While `dtcontrol` has already greatly reduced the size of the controllers in many benchmarks, it still has room for improvements. Most of the implemented heuristics rely on clever ways of determinizing the controller on the fly. This means we start with the most-permissive controller in the lookup table format and then, given the set of safe actions for every state, we dynamically select an action so that the resulting strategy can be represented succinctly. However, in some instances such as the `cruise` model, we want to keep the permissiveness of the controller, for example, to give a human driver the maximum amount of freedom.

For accurately representing the most-permissive controller for the `cruise` model without any determination heuristics, `dtcontrol` needs several hundred decision nodes and the resulting decision tree is hardly explainable. When providing the right domain knowledge, significantly shorter decision trees can be found by using algebraic predicates [Akm19; Wei20]. However, so far, the supplied domain knowledge had to be narrowly tailored to the problem by hand or even had to explicitly contain a handcrafted solution.

Contributions This thesis addresses these limitations by proposing two solutions and critically evaluating them:

- We explore how we can automate the generation of relevant algebraic predicates. Starting with a set of general physical equations, we generate more complex

expressions that we then use in the decision tree construction.

- We facilitate the data from the model checker together with machine learning techniques to generate predicates. Specifically, we use support vector machines [HTF09, Chapter 12] to find quadratic polynomials that nicely separate our data. Then we use those polynomials in the decision nodes.

- We evaluate the second approach on 28 case studies and analyze the results. Most notably, we receive an explainable decision tree with only 5 decision nodes for the `cruise` example and find decision trees of minimum size for 10 out of 28 case studies.

Structure The remaining thesis has the following structure: We start by covering related work in Chapter 2, and then define the basic mathematical notions need for the rest of the thesis in Chapter 3. By analyzing the motivating example of the `cruise` model in Chapter 4, we get further insides about possible strategies. In Chapter 5 and Chapter 6 we describe the two approaches of generating predicates from domain knowledge and from the controller data, respectively. Then, we evaluate the suggested approaches on the entire benchmark set and analyze the performance in Chapter 7. Future work is discussed in Chapter 8 before we conclude the thesis by recapitulating the most important findings in Chapter 9.

2. Related Work

This work extends the open-source tool `dtcontrol` that was first presented in [Ash+20], covered in detail in [Jac20], and since then has been extended significantly [Ash+21]. In this thesis, we combine and adapt techniques from *machine learning* and *formal verification*.

Machine Learning Most of the algorithms we use for constructing decision trees come from the field of machine learning. For representing a controller, however, we have different objectives. In the classical machine learning setting, we are not interested in a perfect classification of the training dataset but rather want our decision tree to generalize well to new data – we want to avoid *overfitting* the training data. This overfitting issue is not present when we use decision trees for controller representation; here, we want to find a perfect classifier as our training and evaluation dataset is the same [Jac20, Section 5.1.5]. Still, those techniques provide a good starting point.

For an introduction to decision trees, we refer to [Mit+97, Chapter3] and [Bre+84]. The idea of more expressive linear or oblique predicates is discussed in [MKS94]. [Ash+19a; CE07] use *support vector machines* (SVM) to find those oblique predicates. For an introduction to SVM we refer to [HTF09, Chapter 12].

There have been several approaches using non-linear predicates in decision trees. [IS96] explicitly constructs new features by combining existing ones (for example take their product or ratio) while [BB98] explicitly uses SVMs inside the decision nodes. Both sources focus on the decision tree's ability to generalize but not on the explainability. Specifically, they do not explicitly reconstruct algebraic predicates from the SVM.

In previous versions of `dtcontrol` [Wei20; Ash+21] *curve fitting* [Arl94] has been used to find undetermined coefficients in algebraic splitting predicates. This approach is based on regression analysis and uses *least square fitting* [Lev44; Mar63]. In our work, however, we use the predicates to separate the data rather than fitting it. For a more detailed comparison, see Section 6.1

Binary Decision Diagrams Typically, *binary decision diagrams* (BDDs) [Bry86] are used to represent controllers in a compressed way [RZ16; ZVJ18]. As BDDs can only represent a binary function $\{0,1\}^n \to \{0,1\}$, this approach requires us to encode the list of state-action pairs of the controller in binary variables. As a result, the BDDs are hardly explainable. Additionally, the size of the BDD heavily depends on the variable ordering. Finding an optimal ordering is NP-complete [BW96] and currently known heuristics struggle with high-dimensional inputs.

Algebraic decision diagrams [Bah+97] extend BDDs to support representing a function $\{0,1\}^n \to S$ with $S \subset \mathbb{N}$. They have been used for representing controllers in [SHB00].

However, they suffer from the same issues we discussed for BDDs. Because of these problems, we focus on decision trees in this thesis.

Predicate Generation The prototype from [Akm19], enables a domain expert to generate algebraic predicates by providing a grammar. Unfortunately, because of the search space explosion, specifically tailored grammars had to be used. Constructing these grammars is tedious and inherently error-prone, especially for domain experts with little experience in grammar construction. `dtcontrol 2.0` [Ash+21] also adds the possibility to semi-automatically generate the decision tree. The user can suggest and evaluate manually entered predicates, though the tool does not provide the functionality to automatically generate new predicates.

Program Synthesis The problem of generating helpful algebraic predicates has similarities to the field of *program synthesis*. There, the goal is to generate program code according to some specification or input/output examples. Similar to the predicate generation, recent work in program synthesis uses generation grammars together with techniques to prune the search space. While these techniques cannot be directly transferred, they might give inspiration for future work. Promossing approaches include [HKO19], [Mor+20], and [CPS20]. [HKO19] uses domain knowledge partly generated from the problem description to reduce the search space for a genetic programming approach. Similarly, [Mor+20] predicts how useful individual parts of the generation grammar are to trim the search space. [CPS20] modifies the grammar to balance the expressiveness of a complex grammar with the low branching factor of a compact one.

In this work, we combine the insights we receive from the controller data with the domain knowledge to construct smaller and more explainable decision trees.

3. Preliminaries

In this chapter, we formally define the key concepts used throughout the thesis. Specifically, we cover controllers, decision trees, and support vector machines.

3.1. Controllers

When formally modeling a dynamic system, different tools can be used. Among the most prominent ones are *timed automata* and *markov decision processes*. Regardless of the specific tool chosen, the concept of a *controller* or sometimes called *strategy* or *policy* is almost always applicable. Intuitively, a controller describes how an agent should act in the dynamic system. What most modeling tools have in common is the concept of discrete *states*, usually defined by the combination of multiple state variables, and the concept of *actions* that an agent can perform. For example, the state of the car in the `cruise` example is defined by the combination of the discrete approximation of its current velocity, acceleration, the front vehicle's current velocity and acceleration, and their relative distance. The actions the agent can perform describe the car's acceleration. This motivates the definition we use here, inspired by [Ash+19b; Jac20]:

Definition 1. For $1 \leq i \leq M \in \mathbb{N}$, let v_i be a discrete state variable. The set of states $\mathcal{S} \subseteq \bigtimes_{i=1}^{M} \mathrm{Dom}(v_i)$ is defined by the M state variables. For a model $\mathcal{M}$ with states $\mathcal{S}$ and actions $\mathcal{A}$, a *controller* $C : \mathcal{S} \to 2^{\mathcal{A}}$ selects for every state $s \in \mathcal{S}$ a set of *safe* actions $C(s) \subseteq \mathcal{A}$.

Note that this definition allows for *permissive* controllers that can provide multiple possible actions for a state. So we define a *most-permissive* controller C as a controller for that every additional action included will lead to a safety violation. On the other side, we call a controller *deterministic* when $|C(s)| = 1$ for all $s \in \mathcal{S}$, meaning we can only choose exactly one possible action in every state. Figure 3.1 shows a deterministic and a permissive controller in a decision tree representation for a battery-powered temperature control system.

Also note that according to our definition, the controller's decision is solely based on its current state, not its past states. In practice, this limitation can oftentimes be circumvented by encoding additional information about the past into the state. For example, the decision of whether to water the plants may depend on the precipitation of the last three days. Then we can model our current state as a tuple (p_1, p_2, p_3) where p_i describes the precipitation i days ago.

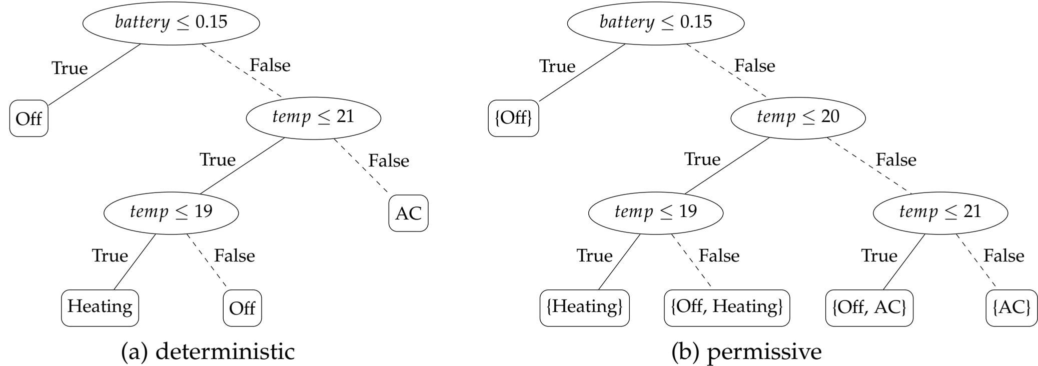


Figure 3.1.: An example of how a decision tree can represent a controller. (a) shows a determinized controller, (b) a permissive one with multiple safe actions at some states.

3.2. Decision Trees

A *decision tree* [Mit+97] is a well-known classification tool from machine learning used to predict discrete output labels of input data. Formally, this classification can be seen as learning a function $f : A \to B$ where B is a set of discrete output labels and $A = \bigtimes_i^M A_i$ is the not necessarily discrete, M-dimensional feature space or input data.

Definition 2. We define a *decision tree* T as following:

- T is a rooted full binary tree, meaning every node either is an *inner node* and has exactly two children, or is a *leaf node* and has no children.
- Every inner node v is associated with a decision predicates α_v. A decision predicate (or just predicate) is a boolean function $A \to \{0, 1\}$ over the input A.
- Every leaf l is associated with an output label $b_l \in B$.

For learning a decision tree, numerous methods exists such as `CART` [Bre+84], `ID3` [Qui86], and `C4.5` [Qui93]. In principle, they all evaluate different predicates by calculating some impurity measure (see Subsection 3.2.1) and then greedily pick the most promising one before splitting the dataset on that predicate and recursively continuing with the two children.

When we have built a decision tree, we can predict the output label of an input vector $\vec{x} \in A$ by starting at the root of T and traversing the tree until we reach a leaf node l. Then, the label of the leaf b_l is our prediction for the input $\vec{x}$. To decide at which child to continue at an inner node v, we evaluate the decision predicate α_v with $\vec{x}$. If the predicate evaluates to true, we pick the left child, otherwise, we pick the right one.

When we represent a controller with a decision tree, our input data is the set of states and an output label describes a subset of safe actions. Figure 3.1b shows an example

of such a tree. In this thesis, we want to find a decision tree that exactly represents the most-permissive controller C. So we make the following definition.

Definition 3. We call a classifier *perfect*, if every sample of our training data is classified correctly.

For us, this means that for every state $s \in \mathcal{S}$, the labels on the resulting leaf exactly match the set of safe actions of this state. In the machine learning setting, this would be considered overfitting, but in our case, that is exactly what we want to ensure safety [Jac20, Section 5.1.5].

3.2.1. Impurity Measures

To evaluate how promising a predicate α is, `dtcontrol` implements different impurity measures. A splitting predicate should aim to minimize the total impurity which is calculated as the weighted sum of the impurities of both children nodes. Here, we will only discuss the most frequently used impurity measure *entropy* and refer to [Jac20, Section 5.3] for a detailed explanation of the choices `dtcontrol` has to offer.

Entropy The impurity measure *entropy* originates from information theory. It measures how much uncertainty is left in a dataset. If the dataset is dominated by one specific label, the entropy is low, whereas a heterogeneous dataset has a higher entropy. For a dataset X with N data points and the label set B, let $n(X, y)$ be the number of data points in X with the label $y \in B$. Then the entropy $H(X)$ is defined as:

$$H(X) = -\sum_{y \in B} \frac{n(X, y)}{N} \log_2 \left(\frac{n(X, y)}{N} \right) \tag{3.1}$$

To evaluate a predicate α, we calculate the remaining entropy after the split. If α is a binary split and partitions X into $X = X_l \uplus X_r$, we define

$$H(\alpha, X) = \frac{|X_l|}{|X|} H(X_l) + \frac{|X_r|}{|X|} H(X_r) \tag{3.2}$$

3.2.2. Predicates

In every decision node of our tree, a predicate function $A \to \{0, 1\}$ is used to decide at which child we continue. We distinguish three categories of predicates according to their complexity:

Axis-Aligned Predicates The simplest and by far the most commonly used type of predicate has the form $v_i \leq c$ for a constant $c \in \mathbb{R}$. Geometrically speaking, the function $v_i = c$ describes a hyperplane orthogonal to the v_i axis, intersecting at $v_i = c$. That is why they are called *axis-aligned* predicates [Mit+97].

For finding the best axis-aligned predicate, we make the simple observation that for a feature v_i with k different values, there are only $k-1$ different relevant values for c. So we can simply evaluate all possible predicates for all features v_i and select the most promising one.

Linear Predicates *Linear predicates* [MKS94] or sometimes called *oblique predicates* have the form $\sum_i a_i v_i \leq c$ with $a_i, c \in \mathbb{R}$. This linear combination of different features describes a hyperplane with arbitrary orientation. Hence they are more expressive but it also makes it harder to find optimal predicates. Algorithms used to find suitable predicates include the OC1 algorithm [MKS94], logistic regression [HTF09, Chapter 4.4] and support vector machines [HTF09, Chapter 12].

Algebraic Predicates As outlined in [Akm19] and implemented in [Wei20; Ash+21], allowing even more powerful predicates can reduce the size of the decision tree and improve explainability. *Algebraic predicates* allow any closed-form expressions hence they are the most expressive. Though, automatically generating good algebraic splitting predicates is a difficult problem and oftentimes is only possible with close human guidance.

Figure 1.1 shows how the three types of predicates work on a toy dataset. As expected, the more expressive the predicate is, the fewer predicates are needed to build a perfect classifier.

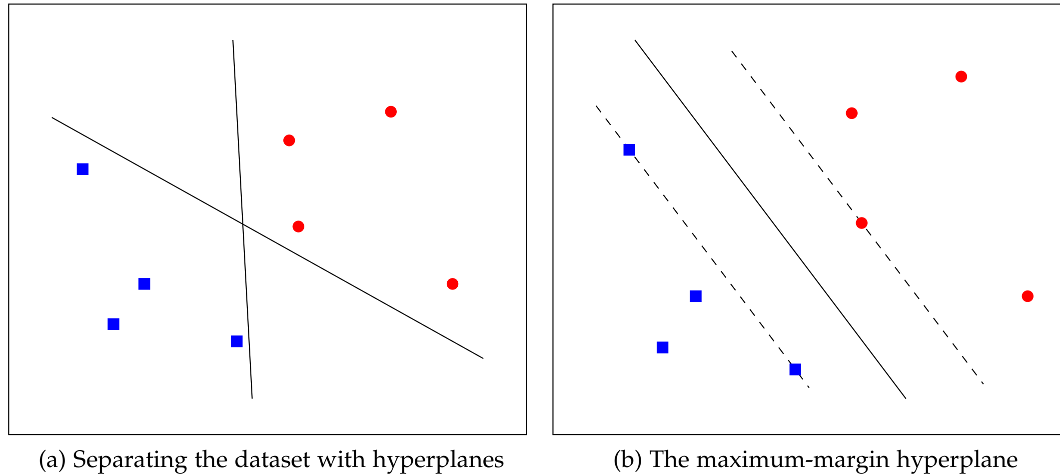
3.3. Support Vector Machines

Support vector machines [HTF09, Chapter 5] (also see [Win10] and [Vap00] for further instructive material) are a key tool we want to apply in this thesis. To understand how they can be used to learn predicates, we will briefly explain the most important concepts. At some places, we refer the interested reader to Appendix A where we elaborate on additional steps. However, these steps are not needed to understand the rest of this thesis.

3.3.1. Separating Hyperplanes

A support vector machine uses a hyperplane to split a dataset into two partitions. A hyperplane of arbitrary rotation is more expressive than an axis-aligned predicate from a decision tree but still general enough to avoid overfitting. Now, consider the dataset in Figure 3.2. We want to find a hyperplane – in this case a straight line – that separates the red and the blue labels. In this example, we can find uncountably many lines that perfectly separate our labels – but which one is the best?

As we want the classifier to be robust against noisy data, a reasonable choice is the line that maximizes the distance between the nearest points on both sides. In



(a) Separating the dataset with hyperplanes

(b) The maximum-margin hyperplane

Figure 3.2.: The sample dataset with different separating hyperplanes.

Figure 3.2b, this line is drawn, together with the margin it leaves to both sides, giving it the name *maximum-margin hyperplane* or *optimal separating hyperplane* [HTF09, Section 4.5.2]. Notably, the dividing line is only defined by three points, the so-called *support vectors*.

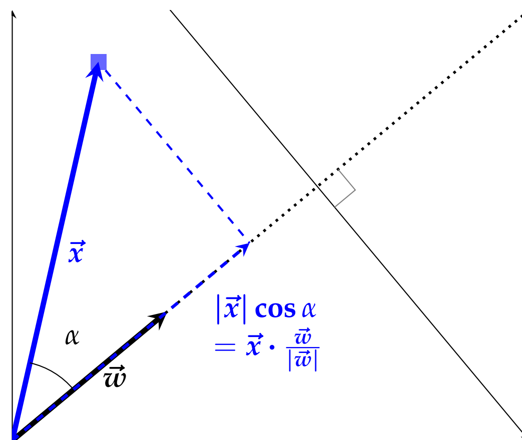


Figure 3.3.: Projecting $\vec{x}$ onto the normal direction $\vec{w}$.

To formally define a hyperplane, we use a (not necessarily normalized) normal vector $\vec{w} \in \mathbb{R}^M$. By taking the dot-product with an arbitrary point $\vec{x} \in \mathbb{R}^M$, we project $\vec{x}$ onto the direction $\vec{w}$ (see Figure 3.3). As all of our points on the hyperplane have the same component in $\vec{w}$ direction, we can represent the hyperplane with

$$\vec{w} \cdot \vec{x} = b$$

for some constant $b \in \mathbb{R}$. Further, for points not on the hyperplane, we can decide on which side they are located on by evaluating $\vec{w} \cdot \vec{x} - b$. In our example, if the expression

is less than 0, the point is located on the left of the line, as the component in $\vec{w}$ direction is smaller than it is for the points on the line. Otherwise, for values larger than 0, the point is located on the right side.

Corollary 1. A hyperplane satisfying $\vec{w} \cdot \vec{x} - b = 0$ acts as the classifier

$$\begin{aligned} c : \mathbb{R}^M &\to \{-1, 0, 1\} \\ \vec{x} &\mapsto \operatorname{sgn}(\vec{w} \cdot \vec{x} - b) \end{aligned} \tag{3.3}$$

where sgn is the sign function.

To find the maximum-margin hyperplane, we can calculate the size of the margin and formulate a constrained quadratic optimization problem. Using further observations, we can reformulate the problem and arrive at a so-called *dual* formulation which we will use in the next section. We refer the interested reader to Appendix Section A.1 and Section A.2. Also, we elaborate on the soft-margin variant of support vector machines in Section A.3. This notion extends support vector machines to cases where the data cannot be perfectly separated by a hyperplane by introducing a loss function.

3.3.2. Kernel Trick

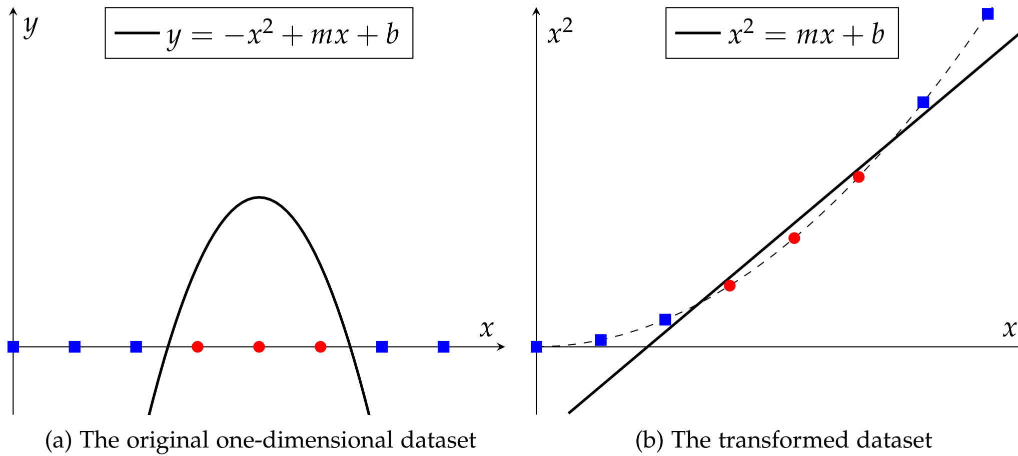
So far, we assumed that it is possible to separate our data with a hyperplane. In practice, this is often not the case. In this section, we learn how we would handle this problem in the machine learning context. Based on this approach, we later develop our own solution in Chapter 6.

Consider the one-dimensional dataset from Figure 3.4a. It is clear that we cannot find a single hyperplane that will separate the data. What can do, however, is to map the data into a higher-dimensional space. For that we define a transformation function $\phi : \mathbb{R}^M \to \mathbb{R}^K$ with $K > M$. In our example, we choose $\phi : x \mapsto (x, x^2)^T$. Our one-dimensional dataset now becomes two-dimensional and suddenly, we can find a separating hyperplane (Figure 3.4b). In our original space, this hyperplane corresponds to a quadratic function.

Of course, this comes at a cost. If we look at a three-dimensional dataset that we want to map with a quadratic transformation, ϕ might look like this: $\phi : (x, y, z)^T \mapsto (x, y, z, xy, xz, yz, x^2, y^2, z^2)^T$ – for the quadratic mapping, the dimensionality already increases quadratically. With that, the computation time for finding the maximum-margin hyperplane and evaluating the decision function increases significantly.

In the machine learning context, we deal with high-dimensional data in areas like natural language processing [Pra+04] and image classification [DS02]. Increasing the dimensionality quadratically or even more is simply not feasible in that case. This is why support vector machines usually make use of the so-called *kernel trick*.

The main idea of the kernel trick is that we do not need transform the dataset to the high-dimensional space explicitly to find the maximum-margin hyperplane. Instead, we use the so-called *dual* formulation of the margin-maximization problem that we derive



(a) The original one-dimensional dataset

(b) The transformed dataset

Figure 3.4.: A linearly inseparable dataset becomes linearly separable by transforming it into a higher-dimensional space. Transforming the decision function back gives a quadratic polynomial.

in Appendix A.2. There, our function only depends on the dot-products of pairs of $\vec{x}_i$. Hence, instead of calculating the high-dimensional dot-product $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$ explicitly, we define a so-called *kernel function* $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$ that we calculate implicitly.

Polynomial Kernel The *polynomial kernel* of degree d is defined as [HTF09, Chapter 12.3]

$$K(\vec{u}, \vec{v}) = (1 + \vec{u} \cdot \vec{v})^d \tag{3.4}$$

For example, for $d = 2$ and $\vec{u}, \vec{v} \in \mathbb{R}^2$, this is

$$\begin{aligned} K(\vec{u}, \vec{v}) &= (1 + u_1 v_1 + u_2 v_2)^2 \\ &= 1 + 2u_1 v_1 + 2u_2 v_2 + 2u_1 v_1 u_2 v_2 + u_1^2 v_1^2 + u_2^2 v_2^2 \end{aligned}$$

and therefore corresponds to

$$\phi(\vec{u}) = (1,\ \sqrt{2}u_1,\ \sqrt{2}u_2,\ \sqrt{2}u_1 u_2,\ u_1^2,\ u_2^2)^T$$

For calculating $K(\vec{u}, \vec{v})$ we only need to calculate the dot-product of the two-dimensional vectors $\vec{u}$ and $\vec{v}$, while the result is the same as the dot product of the six-dimensional vectors $\phi(\vec{u})$ and $\phi(\vec{v})$.

Radial Basis Kernel The *radial basis kernel* (sometimes called *RBF kernel* or *gaussian kernel*) with parameter $\gamma \in \mathbb{R}$ is defined as

$$K(\vec{u}, \vec{v}) = \exp(-\gamma |\vec{u} - \vec{v}|^2) \tag{3.5}$$

While this is one of the most widely used kernels, K is no longer based on a simple transformation function ϕ. In fact [Sha09] shows that ϕ maps to an infinite-dimensional space.

The kernel trick of only implicitly calculating the high-dimensional representation is a key contributor to the success of support vector machines. In our use-case, however, as our data is usually not as highly-dimensional, we propose a slightly different solution in Chapter 6.

3.3.3. Computational Complexity

When working with a large dataset $x_i \in \mathbb{R}^M$ for $i \in \{1, \ldots, N\}$ the runtime complexity is important and impacts the choice of our approach in Chapter 6. As most solvers use iterative methods that depend on the specific dataset, expressing the runtime only in terms of N and M sometimes does not capture the entire behavior – still it gives a good overview of the different methods.

Nonlinear Kernels When working with nonlinear kernels with the kernel trick, usually the dual formulation is used. This way, we do not have problems with high-dimensional data. However, simply evaluating our optimization function (Equation A.11) is already quadratic in the number of data points N. To help with that, solvers like `LIBSVM` [CL11] use decomposition methods and cache results. Depending on the efficiency of the cache, one iteration takes $\mathcal{O}(N)$ time if the cache can be used or $\mathcal{O}(NM)$ time in case the relevant data was not cached [CL11, Section 5.7]. While the exact number of iterations needed has not been formally shown, the expected number of iterations seems to scale "higher than linear" in N [CL11, Section 5.7]. In total, the training time is at least quadratic in N, making it impractical for large datasets.

Linear Kernels When no kernel is used (or the kernel is linear), the training time can be much improved. `LIBLINEAR` [Fan+08] implements fast solvers using trust region Newton methods [LWK07] as well as coordinate descent algorithms [CHL08; Hsi+08]. Here, either the primal or the dual formulation is used depending on the dimensionality of the data. In practice, these methods scale almost linearly to large numbers of data points and dimensions.

4. Motivating Example

As a motivating example, we have a look at the cruise control model from [LMT15a]. Here, we want to develop a controller for a car that ensures that the car will not crash into the front vehicle. As a secondary objective, the car should drive as fast as possible, thereby minimizing the distance between both cars.

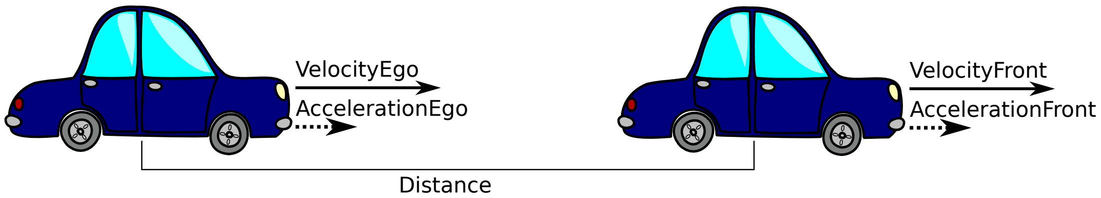


Figure 4.1.: An illustration of the cruise model. Source: [LMT15a]

The model is illustrated in Figure 4.1. We only consider two vehicles, our vehicle called *ego* and the next vehicle in front of us called *front*. We drive on a single lane without cars entering or leaving, therefore this constellation does not change. The state of the system is modelled by the velocities v_e, v_f of the cars and their relative distance d_r; the safety criteria $d_r \geq d_{safe}$ should hold at every state. In the model, both cars choose a constant acceleration a_e, a_f for the duration of one time step t_1. Then, the new state (d'_r, v'_e, v'_f) is given by

$$v'_e = v_e + a_e t_1 \tag{4.1}$$

$$v'_f = v_f + a_f t_1 \tag{4.2}$$

$$d'_r = d_r + (v_f - v_e)t_1 + \frac{1}{2}(a_f - a_e)t_1^2 \tag{4.3}$$

The model restricts the domains of the accelerations to $a_e, a_f \in \{-2, 0, 2\}$ describing the three actions deceleration, neutral, and acceleration. Similarly, the cars have a bounded minimum and maximum velocity v_{min}, v_{max} and the distance sensor has a limited reach of d_{max}. Depending on the values of these parameters, the size of the generated controller changes considerably.

4.1. Model Parameters

We made some adjustments to the `cruise` model compared to the version available at [LMT15b]. In addition to the modifications previous works made (described in

Appendix B.1) we also changed the constants v_{min}, v_{max}, and d_{max}. The `cruise` model used in [Ash+20] and [Ash+21] specified $v_{max} = 20$, $v_{min} = -10$, and $d_{max} = 200$. This provoked the following unwanted behavior. When the distance between the the front and the ego vehicle is at around 150 and both cars can drive at full speed. However, when the relative distance approaches 200, the ego vehicle needs to slow down. The reason being the following behavior of the front vehicle:

1. the front vehicle drives with $v_f = 20$ and $d_r = 190$,
2. the front vehicle disappears into the far-away state as $d_r > 200$,
3. a "new" car appears at the end of the sensor range $d_r = 200$. Independent of the velocity the front vehicle had before, the new car can have any velocity, for example $v_f = -10$.

This way, the front vehicle effectively changes its velocity from $v_f = 20$ to $v_f = -10$ in just a couple of time steps. Even the distance of 200 is not enough for the ego vehicle to react and avoid a crash in this scenario. We fix this flaw by increasing the minimal velocity and the maximum sensor distance so that the ego vehicle has enough time to break if a new car suddenly appears. An overview of the parameters we used for the `cruise` model is in Appendix B.2.

4.2. Problems with the Current Solutions

To illustrate the problem with the current solution, we consider the dataset `cruise_250` (see Appendix B.2). Here, the model checker `UPPAAL Stratego` [Dav+15] generates a controller file with over 400MiB comprising 320,523 states and 961,569 state-action pairs. Representing it with a binary decision diagram [Bry86] still uses over 1,800 nodes. With `dtcontrol` and axis-aligned or linear splits, we can get a decision tree with 869 or 369 nodes respectively which is still far too large to be understandable. Using the determinization heuristics discussed in [Ash+20], we find a decision tree with only 3 nodes. Unfortunately, this determinized controller is of little use – it simply lets the car decelerate until it reaches minimal velocity. Of course, this behavior satisfies the safety criteria but is not helpful in the real world. To also fulfill the secondary objective of minimizing the relative distance we have two options. We can pre-determinize the controller by always picking the largest safe acceleration or we keep the maximal permissiveness. In the latter case, the cruise controller acts as an emergency braking system by letting the human driver choose any action as long as it is a safe one.

4.3. Handpicked Strategy

As shown in [Akm19], there is a decision tree representing the most-permissive controller for the `cruise` example with just 11 nodes. Yet, there has not been a way of automatically generating it with `dtcontrol` so far.

To better understand the model, we will briefly explain how the handcrafted strategy works. In the worst case, the front vehicle will start decelerating in the next time step and will continue until it has reached its minimal velocity. For our car, we have to decide what action to take for the next time step t_1: accelerate, stay neutral or decelerate. To see if it is safe to accelerate, we calculate the relative distance after accelerating for one time step t_1 and then decelerating until the ego vehicle has reached the minimal velocity.

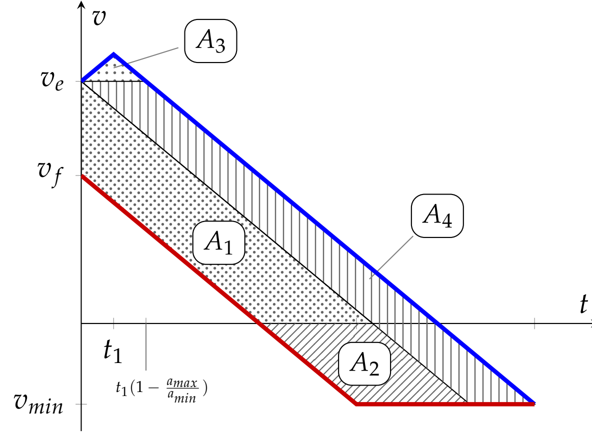


Figure 4.2.: A velocity-time graph showing the ego vehicle accelerating for one time step. The area between the blue and the red curve describes the change in distance between the two cars.

In Figure 4.2 we have plotted the velocity-time diagram describing the kinematics of both cars in case the ego vehicle accelerates in the next time step. The front vehicle (red) instantly decelerates with the rate a_{min} and then continues with minimal velocity. The ego vehicle (blue) starts with a higher velocity, accelerates for one step, and then decelerates with the same rate. The distance traveled is the time integral of the velocity, so the area between the curves describes the relative distance change. We can partition the area into four sections and calculate the respective areas:

$$
\begin{aligned}
A_1 &= -\frac{v_e^2}{2a_{min}} - \left(-\frac{v_f^2}{2a_{min}}\right) \\
A_2 &= v_{min}\frac{v_e - v_f}{a_{min}} \\
A_3 &= a_{max}t_1^2\left(1 - \frac{a_{max}}{a_{min}}\right) \\
A_4 &= (v_e - v_{min})\, t_1 \left(1 - \frac{a_{max}}{a_{min}}\right)
\end{aligned}
$$

With these values, we can write the predicate deciding whether it is safe to accelerate in the next time step as a quadratic polynomial of our state variables v_e, v_f, d_r:

$$\begin{aligned}
&\frac{1}{2a_{min}}v_e^2 - \frac{1}{2a_{min}}v_f^2 \\
&- \left(\frac{v_{min}}{a_{min}} + t_1(1 - \frac{a_{max}}{a_{min}})\right) v_e + \frac{v_{min}}{a_{min}} v_f \\
&- \left(1 - \frac{a_{max}}{a_{min}}\right) t_1(t_1 a_{max} - v_{min}) \\
&+ d_r \geq d_{safe}
\end{aligned} \tag{4.4}$$

5. Predicates From Domain Knowledge

As discussed in [Akm19; Ash+21], we want to facilitate domain knowledge provided by human experts to generate helpful splitting predicates. Providing very specific equations like the handcrafted predicate from Equation 4.4 is inherently tedious and error prone. Ideally, we want to be able to synthesize the specific predicates from general domain knowledge. In the case of the `cruise` model, we use the velocity and distance relations that follow from a constant acceleration a for at time period t:

$$v = at \tag{5.1a}$$

$$d = \frac{1}{2}at^2 + vt \tag{5.1b}$$

5.1. Our Approach

To generate new predicates from the base equations, [Akm19] used grammars. We identified three main problems with that:

- Constructing those is challenging for a non-experienced user.
- The search space explodes if the grammar is not narrow enough.
- It is hard to tell whether an intermediate result with non-terminals is useful until we have substituted all non-terminals.

This is why we propose a slightly different approach. Let $P = \{d, v, a, t\}$ be the set of physical quantities distance, velocity, acceleration, and time we consider in this example. Then, let $S = \{v_e, v_f, d_r\}$ be the set of state variables and $C = \{d_{safe}, v_{min}, v_{max}, a_{acc}, a_{neu}, a_{dec}, t_1\}$ be a set of constants describing the minimum safety distance, the minimum and maximum velocities of the cars, the acceleration values corresponding to the actions "accelerate", "neutral", "decelerate", and the duration of one time step. We observe that every constant and state variable describes exactly one physical quantity. So we define the function $\rho_p(A)$ that returns the subset of entities of the set A that are associated with the physical quantity $p \in P$. For example, $\rho_s(S \cup C) = \{d_r, d_{safe}\}$. Our approach can be described as following:

1. Initialize $V_p := \rho_p(S \cup C)$ for all $p \in P$ describing the values that the physical quantity p may have.
2. For every equation from the domain knowledge (5.1), solve it for every physical quantity. This gives a set of equations in the form $p = f(P \setminus \{p\})$ for $p \in P$ that we

call *base identities*. For example, the base identities for the acceleration are: $a = \frac{v}{t}$ and $a = \frac{2(d-vt)}{t^2}$. A complete list of all 8 base identities is in appendix C.1.

3. For every physical quantity p and every pair of values $x_1, x_2 \in V_p$, add $x_1 + x_2$ and $x_1 - x_2$ to V_p.

4. For every base identity α associated with the quantity p_α, and for every possible substitution function σ that maps physical quantities $p \in P$ to values $x \in V_p$, add $\sigma(\alpha)$ to V_{p_α}.

Steps 3 and 4 can be repeated, thereby creating increasingly complex expressions. As an example, let us see how we can generate the value d_{one} describing the difference in distance after one time step if the ego vehicle accelerates and the front vehicle decelerates. In Step 3 we add $a_{min} - a_{max}$ to V_a, as well as $v_f - v_e$ to V_v. In Step 4 we use the base identity $\alpha : d = \frac{1}{2}at^2 + vt$ with the substitutions

$$\sigma(a) = a_{min} - a_{max}$$
$$\sigma(t) = t_1$$
$$\sigma(v) = v_f - v_e$$

and we receive the expression for d_{one}:

$$d_{one} = \frac{1}{2}(a_{min} - a_{max})t_1^2 + (v_f - v_e)t_1$$

In contrast to the grammar approach, every predicate we generate can be used directly as a splitting predicate in a decision tree – we do not have any non-terminals we need to replace later. For example, we could try to use $d_{one} \leq c$ for some constant $c \in \mathbb{R}$ in our decision tree, where we would replace the constants a_{min}, a_{max} and t_1 in d_{one} with their respective numerical values, leaving us with a function of the state variables v_f and v_e. Or instead, we could use d_{one} in the next substitution to create more sophisticated expressions.

5.2. Handcrafted Predicate Derivation

We have seen a technique of generating compounded predicates, but how far away is the handcrafted predicate we want to synthesize? For that, we try to derive the handpicked predicate from Section 4.3 using the domain knowledge (cf. [Akm19]). For now, we assume $v_e \geq v_f$ as these are the interesting cases. Then, the kinematics can be described in three phases. First, the behavior in the next time step where the ego vehicle accelerates and the front vehicle decelerates. Second, the phase when both cars break lasting until the front car has reached minimum velocity. Third, the final phase where the front car continues at minimal velocity whereas the ego car continues to decelerate until it also reaches minimum velocity. We define the following expressions:

- d_{one}: the change in relative distance during the first time step.
- v_{fChg}, v_{eChg}: the change in velocity for both cars during the first time step.
- v'_f, v'_e: the new velocity of both cars after the first time step.
- t_f, t_e: the time after the first time step until the respective cars reach minimal velocity.
- d_f, d_e: the distance traveled by the respective cars after the first time step until they reached their minimal velocity.
- d_{fe}: the distance the front car travels with minimal velocity until the ego car also reaches its minimal velocity.

Using these expressions, Figure 5.1 shows how we can arrive at the handcrafted predicate. At the top, we have a subset of the base values $S \cup C$ comprising constants like a_{max} and state variable like d_r. The color encodes to which physical quantity a value belongs. For example, all velocities are drawn in blue. Then the diagram shows the iterations of our algorithm. Every iteration consists of two phases: Step 3 where values of the same type can be added or subtracted to form new values, and Step 4 where we use our domain knowledge base identities to calculate new values. We leave out the irrelevant values as the total number of generated values would be far too large as we will see in the next section.

We observe that the first meaningful distance predicates emerge after four iterations. Then, summing the distance expressions together takes another 3 iterations. The non-simplified version of the predicate that our algorithm would output is shown in Appendix C.2.

5.3. Performance

Having an understanding of the goal, we can now interpret the performance of our search. Unfortunately, the proposed approach is infeasible in practice. From 8 base identities and 9 starting values from $C \cup S$ (we leave out $a_{neutral}$), after one iteration, we already have 3,604 predicates. After the second iteration, we estimate the number of predicates to be in the realms of 10^{18}.

An important contributor to the growth is Step 3. Without Step 3, we generate 66 predicates in the first and 10,568 in the second iteration. Unfortunately, Figure 5.1 shows that these sums and differences are crucial throughout all iterations of the algorithm.

While the equation from the geometric interpretation (Equation 4.4) is less complicated, it does not offer a convenient way of deriving it from the domain knowledge, especially not in an explainable way.

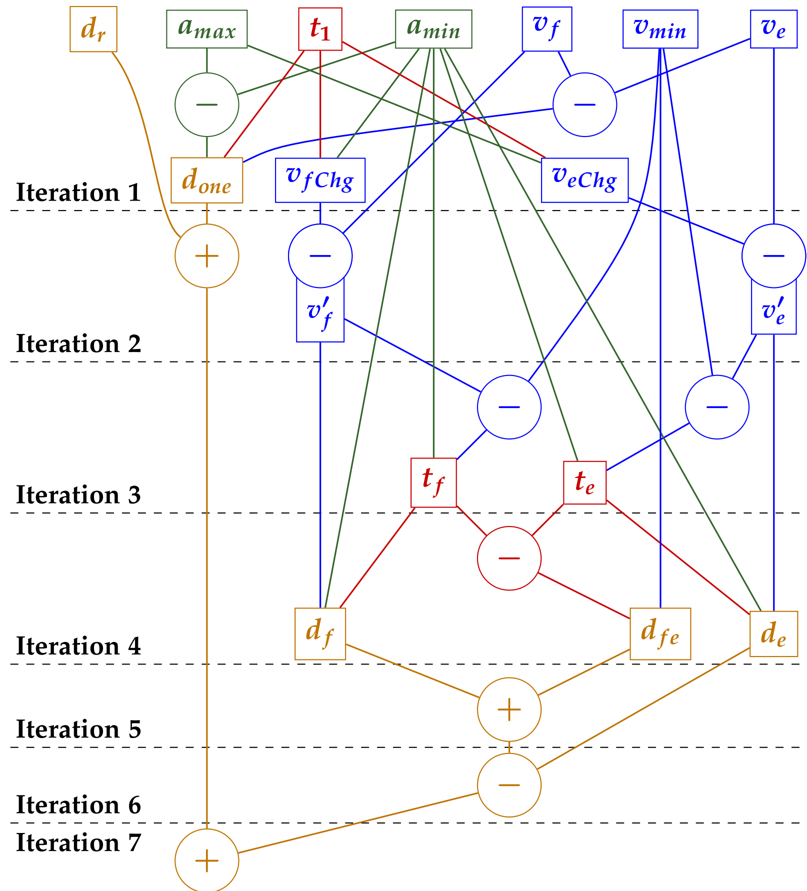


Figure 5.1.: A derivation of the "can-accelerate" predicate.

5.4. Identified Problems

We have identified two fundamental problems with this approach: the large search space and the missing uniqueness of the derivation.

First, as the search space is so large, we need a heuristic telling us which expression will be useful at a later stage. When introducing the approach we stressed that we can use any intermediate predicate in the decision tree. So a natural choice for a heuristic would be some kind of impurity measure on the controller data. Unfortunately, expressions like the time until we reach minimal velocity are not great splitting predicates. And even expressions close to the final predicate like d_{one} or d_e are of little use on their own. In Figure 5.2, we plot the handcrafted predicate d^* together with its individual components in a two-dimensional plot for the value $v_f = 2$. While each predicate individually is a bad classifier, their sum can perfectly classify the data. Given only the impurities of the predicates, there does not seem to be a way to conclude which predicates are useful. Thus, developing a better heuristic is an important step for future work.

Second, the handcrafted predicate is so complex that there are alternative expressions

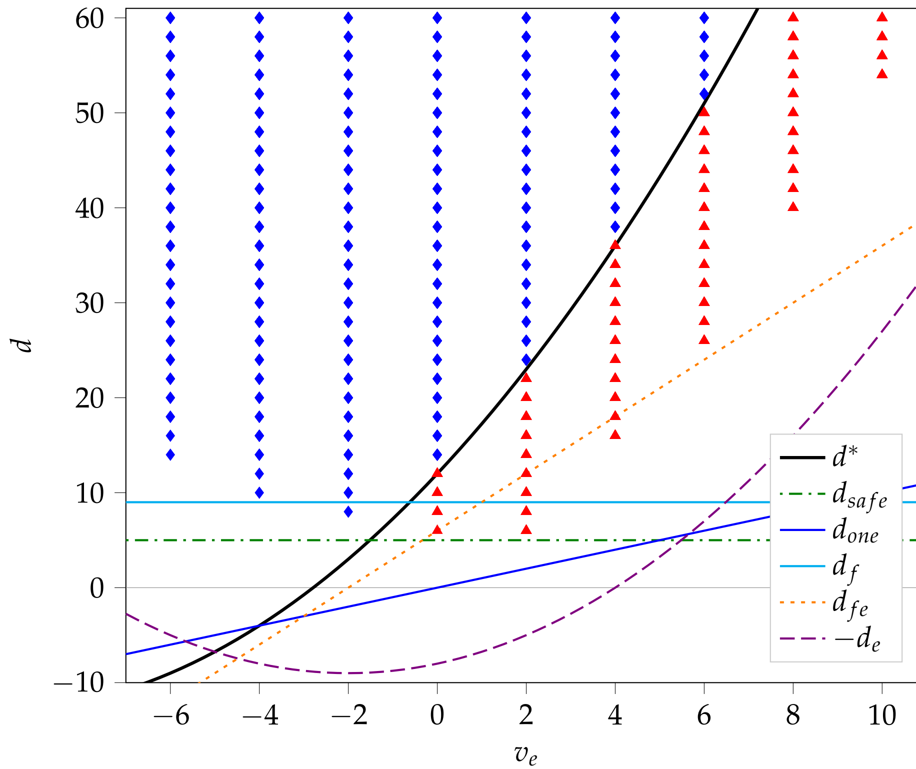


Figure 5.2.: The handcrafted predicate d^* and the terms used to derive it, plotted for a fixed value of $v_f = 2$. While the sum of the terms is a perfect classifier, individually, they are not helpful for the classification.

that evaluate to the same predicate after substituting the constants with their values. For example, when calculating d_{one} in Section 5.1, we set the acceleration to $a_{min} - a_{max}$ which evaluates to -4. Our approach would also try setting the acceleration to $a_{min} + a_{min}$ which also evaluates to -4 but lacks any relation to the situation we want to describe. So only the general domain knowledge and the controller data might not be enough to unambiguously derive an explainable predicate. Either a human has to steer the process and select the most sensible predicates, or we have to somehow incorporate additional information about the model.

With these realizations, we try to approach the problem from a completely different perspective in the next chapter.

6. Predicates From Controller Data

We now propose a different approach that does not use domain knowledge but instead precisely analyzes the controller data. In Figure 6.1, we have visualized a part of the controller data from the `cruise` model together with a handcrafted splitting predicate. The coordinate axes describe our three state variables v_e, v_f, d_r and the color of the data points describes which actions are allowed. We see that the handcrafted strategy perfectly separates the red and blue labels. By looking at a two-dimensional plot for a fixed value of v_f in Figure 5.2, we see that the predicate is well defined by the data points – we should be able to reconstruct the splitting function by solely looking at the data and fitting a function to it.

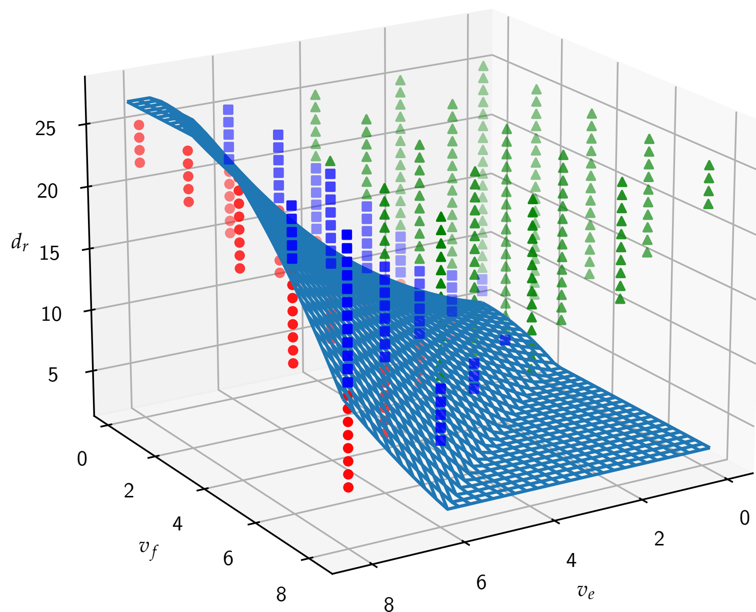


Figure 6.1.: Visualization of a handcrafted predicate perfectly separating the data.

In this chapter, we will first explore why the available curve fitting functionality is not sufficient for our goal and then propose an alternative solution using support vector machines. To make the decision trees even smaller and more explainable, we introduce four additional techniques. First, we simplify the individual predicates by removing

unimportant terms in Section 6.3 and rounding the coefficients to nice numbers in Section 6.4. Then, we optimize which predicates are selected when building the decision tree by proposing a new impurity measure in Section 6.5 and changing the predicates' priorities in Section 6.6.

6.1. Problems with Curve Fitting

The recent extensions of `dtcontrol` [Wei20; Ash+21] enable us to use curve fitting [Arl94] for finding unspecified coefficients. We know from Equation 4.4 that the handpicked strategy is a quadratic polynomial so we can try to use a general quadratic polynomial $c_1 v_e^2 + c_2 v_e v_f + c_3 v_f^2 + \dots$ and determine the coefficients with curve fitting. Unfortunately, this approach fails to find the correct predicate. To understand why we need to investigate how the curve fitting is implemented.

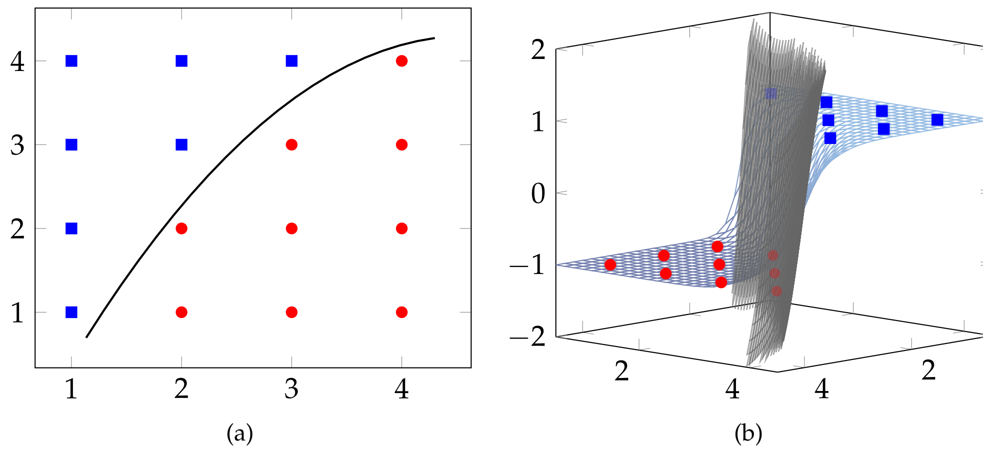


Figure 6.2.: In the current curve-fitting implementation, a two-dimensional dataset (a) is mapped to a three-dimensional space where the $z \in \{-1, 1\}$ is determined by the label. The old approach then fits a function to the new dataset. Our approach instead tries to separate the data like the gray surface does in (b).

For now, we always consider a *one versus the rest* split. This means, we pick a label y that we want to separate from the rest and set

$$y_i' = \begin{cases} +1 & \text{if } y_i = y \\ -1 & \text{else} \end{cases} \quad \text{for all } i$$

Consider the two-dimensional data from Figure 6.2a. What the current version of curve fitting does is the following. First, we map our two-dimensional data $x_i \in \mathbb{R}^2$ with label $y_i' \in \{-1, 1\}$ to the three-dimensional space where y_i' is used as the third coordinate. Then we use regression analysis to fit a function to the data with least-squares-fitting [Lev44; Mar63] (see Figure 6.2b). What we propose in this thesis, is to use a classification

approach rather than a regression approach. So in Figure 6.2b we are interested in the gray function *separating* the data points instead of fitting them. This way, we put most emphasis on the sample points close to the split rather than weighting every sample equally. Coming back to the two-dimensional space (Figure 6.2a), we want to find a function that smoothly separates the labels, ideally maximizing the distance to any specific sample. This is where support vector machines come into play.

6.2. Using Support Vector Machines

As described in Chapter 3, support vector machines (SVMs) exactly do what we want: find a function that separates the data and maximizes the margins. The main idea offered in this work is that we can reconstruct the algebraic decision function from the internal coefficients of the SVM. This is not feasible for tools like neural networks [HTF09, Chapter 11] but we will see how and under what conditions it is possible for SVMs in the next sections.

The `dtcontrol` tool already supports finding linear splitting predicates with SVMs. Though, for the `cruise` example, a linear predicate is not enough to perfectly split the data. So we are tempted to use a polynomial kernel to increase the expressiveness of our SVM. However, we recall from Subsection 3.3.3 that the runtime of common training algorithms for SVMs is at least quadratic in the number of samples. And in fact, the algorithms implemented in the open-source tool `scikit-learn` [Ped+11] do not terminate within an hour for the `cruise` example with a few hundred thousand sample points.

We can circumvent this issue by taking advantage of our specific use case. Usually, SVMs are used with high-dimensional datasets like images [DS02] or language models [Pra+04] where the number of features has the same order of magnitude as the number of samples. For the purpose of controller synthesis, the number of state variables is usually small as the number of states usually grows exponentially with the number of state variables. So while the kernel trick is useful for high-dimensional data, we can renounce the kernel trick in our case and explicitly construct the higher-dimensional space as we did at the beginning of Subsection 3.3.2. A similar idea is also described in [Cha+10].

For example, when we want to change from the linear three-dimensional space $(v_e, v_f, d_r)^T$ to the quadratic space, we will have the following 9 dimensions

$$(v_e,\ v_f,\ d_r,\ v_e v_f,\ v_e d_r,\ v_f d_r,\ v_e^2,\ v_f^2,\ d_r^2)^T \tag{6.1}$$

The moderate increase in dimensions is clearly outweighed by the much better performance of the linear SVM algorithms we can now use.

6.2.1. Problems With Higher Dimensions

At the moment, we only support mapping to the quadratic space, which means our predicates are quadratic polynomials. For higher degree polynomials, we have not seen

that the gained expressiveness justifies the significantly increased complexity of the predicates. For example, a cubic predicate with 5 variables already has 55 terms. Even with the methods we will discuss in Section 6.3 and Section 6.4, this predicate will not fulfill our goal of being explainable. Mapping to a space with features like e^x or $\sin(x)$ poses the challenge that we can only fit the coefficient, but not scale the function in x-direction like e^{cx} or $\sin(cx)$ and is therefore left for future work.

6.2.2. Reconstructing the Algebraic Decision Function

Assuming that our SVM finds a separating hyperplane that we want to use as a splitting predicate in our decision tree, how do we reconstruct the algebraic representation? The SVM algorithm finds a hyperplane $(\vec{w}^*, b^*)$ with $\vec{w}^* \cdot \vec{x} - b^* = 0$ where $\vec{x}$ corresponds to a transformed set of state variables in the form of Equation 6.1. This means the w_i are the coefficients of the quadratic polynomial of our state variables.

When implementing it in practice, there is a small intermediate step we want to mention for completeness. In order for the quadratic optimization algorithm to work properly, the input data needs to be normalized to have a mean of 0 and a standard deviation of 1. This standardization of course has to be taken into account when exporting the coefficients.

We have now seen how SVMs can help us generate predicates that nicely separate two label sets. Still, the polynomial predicates we receive for the `cruise` example consist of up to 25 terms (see Appendix D.1 for an example). To improve readability and explainability, we simplify the predicates with two methods: selecting the most important features and rounding coefficients.

6.3. Feature Importance

As we have discussed in Chapter 4, the state of the `cruise` model is defined by v_e, v_f, and d_r. However, the model checker `UPPAAL Stratego` also exposes four additional state variables. These comprise the current acceleration values a_e, a_f that do not impact the acceleration the cars choose in the next time step and the variables f_{choose} and e_{choose} that are an artifact from the internal model and have constant values for all relevant states.

To recognize such unimportant variables, we introduce a basic version of a feature importance measure. Consider the two-dimensional dataset shown in Figure 6.3a with features x_1 and x_2. To classify a data point, feature x_2 is not needed. We verify this, by removing feature x_2 and grouping the data points with the same x_1 value. We can now measure how many "collisions" occur. If zero collisions occur, the predicate is not needed. Otherwise, we can give a rough estimate of the importance of that feature by calculating the ratio of data points where a collision happened.

Note that for a dataset like Figure 6.3b, this approach would judge both features as irrelevant. Individually seen that is correct but we can only remove one of them without

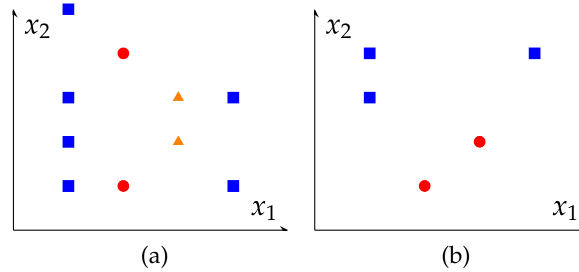


Figure 6.3.: Examples with redundant features. In (a) x_2 is not needed. In (b) x_1 and x_2 individually look redundant, but only one may be removed. Also, removing x_1 is preferred over removing x_2

causing collisions. This is why we calculate the feature importance incrementally. When we find an irrelevant feature, we remove it directly before calculating the importance of the next feature. As a result, the outcome may depend on the order of features we choose. For example, in Figure 6.3b, removing x_1 would result in a linearly separable dataset while removing x_2 would not. In general, there might even be a case where we can either remove a single feature x_i or all three features features $x_{i+1}, x_{i+2}, x_{i+3}$. However, we did not observe any behavior like this so far, so we leave this issue for future work.

6.4. Rounding Coefficients

With the feature importance, we remove variables that are clearly useless and reduce the number of terms in the `cruise` predicates from 25 to 9. Still, we generate predicates that contain unnecessary terms. For example, we know from our handcrafted predicate (Equation 4.4) that we do not need a d_r^2 term for the `cruise` predicates. But, in the predicate we generate (see Appendix D.2), the respective coefficient has a small positive value. To understand why that is the case, recall that the only objective the SVM has is to maximize the margin between the data points. For that, a small coefficient for d_r^2 seems to be beneficial. If we loosen the maximum margin objective, we can generate a predicate with equivalent accuracy but a simpler algebraic expression. Again, as we are not interested in the classifier's ability to generalize – as long as the accuracy for our controller data stays the same – we do not care about how large the margins are.
So, to prettify our predicate, we proceed in three steps:

1. Setting coefficients to zero.

2. Scaling the entire predicate.

3. Rounding coefficient to integers or *nice* numbers.

6.4.1. Rounding to Zero

If we can set a coefficient to zero, the predicate becomes significantly shorter and easier to understand. So this is our primary goal. A natural approach is to try setting a coefficient with a small absolute value to zero and checking if the classification for all samples stays the same. While this suffices for some coefficients, sometimes we need to change the remaining ones to counterbalance the change. So, what we do instead is to remove the feature temporarily and try re-training the SVM. If successful, we permanently remove the feature for this split and try the next feature. Similar to the feature importance approach (Section 6.3), the result may again depend on the order of coefficients we try to remove. Here, we use the heuristic of trying to remove the coefficient with the smallest absolute value first.

Compared to the feature importance approach, three key differences make this approach more powerful:

- We only consider the subset of the entire dataset available in the current subtree.
- We only focus on separating one specific label (we only have the two labels $+1$ and -1).
- We directly consider the features in the higher-dimensional space such as d_r^2

6.4.2. Scaling the Predicate

An additional step to improve readability is to scale the generated predicate. In principle, a predicate $\alpha : 0.5x + 0.1y \leq 0.3$ is equivalent to a scaled predicate $10\alpha : 5x + y \leq 3$ but the second one might be easier to read. The SVM uses an internal scaling constraint (Appendix Equation A.1) but for us, this is not relevant. We can again lift this constraint and scale all coefficients as well as the intercept value b arbitrarily. One could think of various heuristic of how to scale the predicate. We decided to use a simple one: we search for the coefficient with the value closest to 1 and scale the predicate so that it becomes exactly 1. This way, we have at least one term with a simple coefficient.

6.4.3. General Rounding

As the last step, we generalize the "rounding to zero" approach and use it on the coefficient we could not set to zero. This way, instead of having a predicate like $8.165839d_r^2 - 2.935846v_r \leq 0$ we can use a nicer looking one like $8d_r^2 - 3v_r \leq 0$. For that, we try the approach from above with increasing relative precision. For example, for the coefficient of d_r^2, we first try the value 10, then 8, then 8.2, and so on, until we find a value that does not change the classification for any sample. Note that we do not re-train the SVM in this step but simply change the coefficient and check if the classification stays the same.

With these techniques, we can finally generate pretty predicates. For example, one the predicate we find for the `cruise_250` dataset exactly corresponds to the handcrafted polynomial from Equation 4.4 after substituting all constants. The only difference is the constant offset:

$$-0.25v_e^2 + 0.25v_f^2 - 5v_e + 3v_f + d_r + 19.5 \leq 0 \tag{6.2}$$

But before we continue, we need to have a look at numerical problems we encounter while rounding.

6.4.4. Numerical Errors

One problem we need to handle concerns floating point precision errors. When testing our classifier, we use the internal coefficients of the SVM. The coefficients we output are different though, as we need to undo the normalization we applied. We must ensure that possible precision errors from these transformations do not change the classification. In the original predicate generated by the SVM, the classifier maximizes the margin between the label sets so we can be quite confident that small precision errors will not change the classification[1]. When trying out rounded coefficients, however, we lose this property. A rounded coefficient might classify everything correctly but the slightly different transformed coefficient might lead to other results.

As a heuristic against these problems, we over-approximate a change when trying out a rounded coefficient. For example, if our current coefficient is 2.953 and we want to try the rounded value 3, we over-approximate the change and try 3.00001 instead. If that works, we can be more confident that the value 3 will not lead to those problems.

Additionally, we also encounter precision problems if our SVM uses very large coefficients. How we deal with them is discussed in Appendix D.3.

When the tree construction is finished, `dtcontrol` verifies that every sample is classified correctly or outputs an error rate. In this step, we use the transformed coefficients of the polynomial we output so we can be sure that the decision tree is as accurate as the tool tells the user.

6.5. Min-label Entropy

Now that we have pretty predicates, we shift our focus to the decision tree construction for the next two sections. For the `cruise` dataset, we can now construct a decision tree with 37 nodes, only 10% of the size when using linear predicates. Moreover, we have seen that we generate the exact predicates we derived by hand in Section 4.3. Still, the decision tree is not as compact as the 11 node tree from [Akm19] as we do not directly use those predicates. To understand why this is the case, we have a look at Figure 6.4. We see that split A perfectly separates the blue label from the rest, while B separates the red and orange labels but distributes the blue one among both children. Considering

[1]Note that this only holds for cases where we find a perfect split.

only a single split, we would prefer split B because the dataset is nicely separated except for the small number of blue samples. The entropy impurity measure comes to the same conclusion and assigns split B a better entropy score.

However, when building a perfect classifier for representing the most-permissive controller, we have a different perspective than in machine learning. At some point, we need to separate the blue labels from the rest. If we do not separate them now and select split B, we have to add additional splits on *both sides* of the split B. If we rather start with split A, we can select split B as the next split in the left child and receive a smaller decision tree. Both options are shown in Figure 6.5.

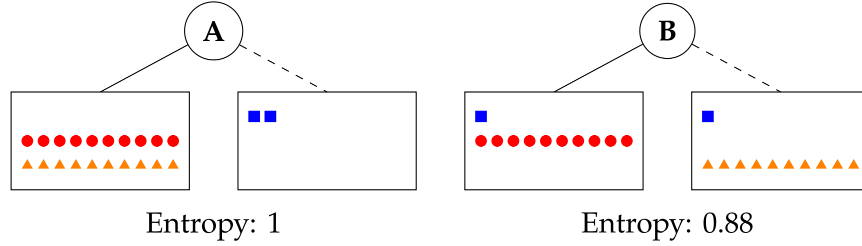


Figure 6.4.: Two different splits with their respective entropy values. While split B has a better entropy value and is preferred in machine learing, we want to use split A first when building a perfect classifier.

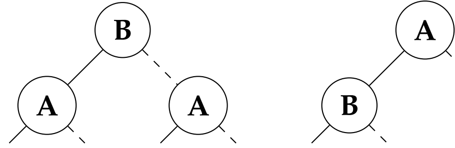


Figure 6.5.: The decision trees needed for perfect classification with the splits from Figure 6.4.

This effect is especially prevalent if the number of samples per label differs significantly. In the `cruise` example, we observe exactly that: the label "all actions are allowed" has 20 times more data points than any of the other labels. As a countermeasure we introduce a new impurity measure that we call *min-label entropy*:

Definition 4. For a dataset $X = X_l \uplus X_r$ with the label set B, let $n(X, y)$ describe the number of data points in X with label $y \in B$. For a predicate α that splits the dataset into X_l and X_r, we define the *min-label entropy* H^* as

$$\begin{aligned} K(p) &:= -p \log_2(p) \\ H^*(X, y) &:= K\left(\frac{n(X, y)}{|X|}\right) \\ H^*(\alpha, X) &:= \min_{y \in B} \left\{ \frac{|X_l|}{|X|} H^*(X_l, y) + \frac{|X_r|}{|X|} H^*(X_r, y) \right\} \end{aligned} \tag{6.3}$$

Intuitively, the *min-label entropy* measure estimates for every label y, how difficult it will be to to separate the label y in both partions after this split. Then it returns the value of the best label. The strategy we want to provoke with this impurity measure is to first fully separate one label and then continue with the next one. Specifically, if we can completely separate one label like in the example in Figure 6.4, the impurity for this split is 0 and we definitely select such a spilt.

6.6. Predicate Priority

With the min-label entropy, we reduce the decision tree size of the `cruise` example to 25. As a last optimization heuristic, we also adjust the priorities of the predicates. When deciding between an axis-aligned and a polynomial predicate that both have similar impurity values, we want to choose the axis-aligned one as it is considerably simpler to understand. For that reason, `dtcontrol` has implemented a priority function for predicate generators. For example, when we give the polynomial predicates the priority 0.5 and the axis-aligned ones the priority 1, we only choose a polynomial predicate if it is at least twice as good in terms of the impurity measure. In fact, we want to choose an even lower value as a priority for another reason. In the `cruise` example, we know that we can find a polynomial that distinguishes cases where we can accelerate from those where we cannot. In our handpicked strategy, we did however not consider the edge cases when we are already driving at minimal or maximal velocity. If we do not exclude those, the data is not perfectly separable, meaning we will find a polynomial split that almost classifies everything correctly, but misses a few data points. While this is not a huge problem, it turns out that it is more effective to first exclude the edge cases with axis-aligned predicates and then perfectly split the data with a complex predicate later. We can achieve it with a low priority value ≤ 0.2 for the polynomial splits in combination with our min-label entropy. This way, we will only choose the complicated splits if they are at least 5 times better. Note that the impurity is 0 if we can perfectly separate one label, so in this case, we are infinitely better than any non-perfect solution.

7. Evaluation

In this chapter, we will evaluate how well the two approaches proposed in Chapter 5 and Chapter 6 perform on our running example `cruise` as well as on other benchmarks.

Artifacts All resources such as generated domain knowledge predicates, model files, and synthesized controllers used in this thesis are available to download at [Jün21]. The repository also contains scripts to reproduce the benchmark tables presented in this chapter.

7.1. Domain Knowledge Approach

As we have seen in Chapter 5, our approach was unable to generate the handcrafted predicate for the `cruise` example. Still, we generated a lot of predicates that might be useful when building the decision tree. We evaluate three sets of predicates that we generated with our approach from Section 5.1. As the number of predicates increases so fast and we cannot even complete two iterations, we also try skipping Step 3 in the approach, meaning we do not add sums and differences of our values. The number of nodes of the resulting decision trees for the `cruise_250` dataset are shown in Table 7.1. In addition to the number of generated predicates, we also list the number of unique predicates as multiple predicates can evaluate to the same expression after substituting the constants with their values (we have seen the example $a_{min} = -a_{max}$ in Section 5.4). To build the decision tree, we use the generated predicates in addition to the axis-aligned predicates and choose the best splitting predicates using the entropy impurity measure.

As a comparison, we have included the decision trees we receive without domain knowledge using only axis-aligned splits and using linear predicates generated with the OC1 heuristic [MKS94].

We see that the generated predicates help find succinct decision trees. For the largest predicate set, we even find a smaller tree than we do with linear predicates. Still, it is not clear whether this is because the predicates describe the dynamics of the system well or whether this improvement is simply due to the large number of predicates we try. In fact, we try so many splitting predicates that the runtime increases from 1 minute when using linear predicates to over twelve hours for the large predicate set, even after implementing use-case-specific optimizations.

As we did not succeed in creating truly explainable decision trees for the cruise example, we do not try this approach on other case studies but instead focus on the second approach.

Table 7.1.: The decision tree sizes (number of nodes) while using different sets of predicates for the `cruise_250` dataset. The column "Sum?" describes if we use Step 3 of our generation approach from Section 5.1.

Iterations	Sum?	#Predicates	#Unique Pred.	DT Size
1	No	66	42	655
1	Yes	3,604	1,929	395
2	No	10,568	9,634	269
Comparision				
Axis-aligned predicates				869
Linear predicates				369

7.2. Data-Driven Approach

We will now evaluate how well generating quadratic polynomials with SVMs performs in practice. While developing the various techniques and heuristics, we mainly focused on the `cruise` dataset. In this section, we first analyze the results for this dataset but then investigate how well the approach generalizes for other case studies. For that, we compare our results to the existing approaches and to the minimum decision tree achievable in theory. Then we will look at our new impurity measure and its performance independently.

7.2.1. Cruise Control

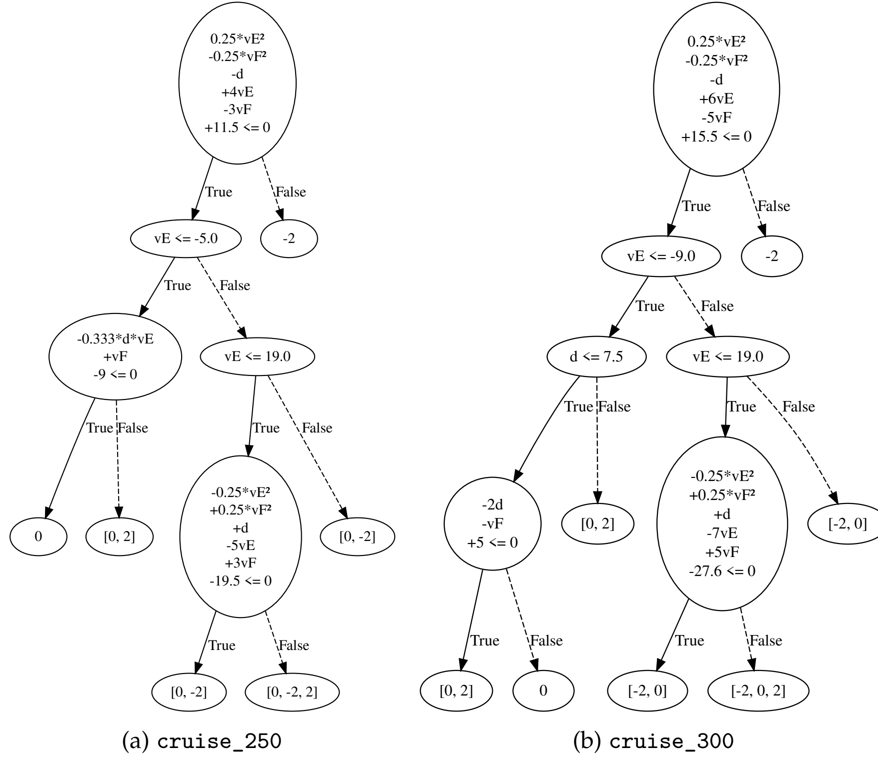
Using all the strategies discussed in Chapter 6 we achieve great results for the `cruise` model. For the `cruise_250` dataset, we find a succinct decision tree with only 11 nodes (see Figure 7.1a). This is exactly the number of nodes [Akm19] found with the handcrafted strategy. In fact, we precisely found the handcrafted "must break" and "can accelerate" predicate from Equation 4.4 (or in a different formulation in Appendix C.2). There is only a slight difference in the constant offset.

For the slightly larger `cruise_300` dataset, we generate a very similar but slightly larger decision tree with 13 nodes (Figure 7.1b). The quadratic predicates change in line with the change of the constant v_{min} (see Table B.1 in the appendix) and one complex splitting predicate is exchanged for two simpler predicates.

In both cases, the generated decision trees are almost 80 times smaller than the ones we receive with axis-aligned predicates and 30 times smaller than the ones with linear predicates.

7.2.2. Minimum Tree Size

To better understand the quality of our results, we compare them to the theoretical minimum-sized decision trees. We can give a lower bound on the number of nodes the decision tree must contain if we want to represent the entire controller without



(a) `cruise_250` (b) `cruise_300`

Figure 7.1.: The decision trees for the `cruise` example generated by our data-driven approach.

determinizing (the most-permissive controller) as following. At every state s, a subset of actions $C(s) \subseteq \mathcal{A}$ is allowed. We define $U := \{C(s) \mid s \in \mathcal{S}\}$ as the set of all possible allowed action subsets that occur in our controller at at least one state. To completely represent the controller, we need at least one distinct leaf in our decision tree for every distinct element $u \in U$. If we disregard the non-binary splits for categorical variables introduced in [Ash+21], we always build a full binary decision tree. As a full binary tree with n leaves has $n - 1$ inner nodes, the lower bound for the total number of nodes of our decision tree is $2|U| - 1$.

If the decision predicates are sufficiently complex we can always achieve this bound. However, in practice oftentimes this is not even desirable. For example, we see that our decision tree for the cruise example in Figure 7.1a does not have minimum size as it contains two leaves with the actions $a \in \{-2, 0\}$. Still, to keep an explainable decision tree, we would not want to merge those leaves as one describes that the car cannot accelerate because it has already reached its maximum velocity while in the other case, accelerating would be technically possible but would lead to unsafe behavior.

7.2.3. Benchmarks

To see how our approach and the individual heuristics generalize, we evaluate them on the case studies of cyber-physical systems from [Ash+20] as well as on the case studies from the quantitative verification benchmark set [Har+19] that were used in [Ash+21]. We do not use any determination heuristics but generate the most-permissive controllers. We ran all experiments on a server with the operating system Ubuntu 20.04, a 2.2GHz Intel(R) Xeon(R) CPU E5-2630 v4 and 250 GB RAM. Table 7.2 contains a selection of the results, with the case studies of cyber-physical systems at the top and quantitative verification at the bottom. In every row, we compare the number of nodes in the generated decision tree for

- the axis-aligned splitting strategy (Ax.Al.),
- the smallest decision tree we could generate with axis-aligned and linear predicates[1] (Linear),
- axis-aligned predicates and the quadratic polynomials generated by support vector machines with a priority value of 0.1 (Poly),
- and with the default priority value of 1.0 (PolyPrio1).

In every cell, the top number describes the result using the entropy impurity measure and the bottom number refers to the result using min-label entropy. TO indicates that we were not able to generate a decision tree within three hours. As a comparison, we list the number of states of the controller as well as the theoretical minimum size of the decision tree.

A complete table with all 28 case studies, a comparison with BDDs, and results for different linear strategies can be found in Appendix E.

Scatter Plot Additionally, Figure 7.2 visualizes the results in a logarithmic scatter plot. As a reference, we take the smallest tree we could generate with linear predicates and the entropy impurity. Then we compare it to the size of the tree with axis-aligned predicates and our quadratic polynomials. For example, the two blue points near the location $(370, 10)$ are the two cruise datasets. The x-coordinate is the size of the tree with linear predicates and the y-coordinate shows the size of the polynomial or axis-aligned results.

Statistics Our new approach gives smaller decision trees for almost all case studies, except for `helicopter` and `cdrive.10`[2] where the linear solution is smaller by 6% and `traffic_30m` where we run into a timeout. Table 7.3 shows the cumulated statistics. Most notably, we increase the cases where we find a tree of minimum size from 2 to 10 out of 28.

[1] We calculate this as the minimum over the three splitting strategies logistic regression, linear support vector machines, and the OC1 heuristic.

[2] see Appendix E

Table 7.2.: The number of nodes of the generated decision trees using axis-aligned splits, linear splits, and the proposed quadratic polynomial splits with priority 0.1 and 1.0. Each row displays the result using the entropy impurity measure at the top and using min-label entropy at the bottom. TO means time out after 3 hours. As a comparison, we show the number of states of the underlying controller and the minimum size a decision tree needs to have. The full table is in Appendix E

	Comparision		Previous		Quadratic	
Case Study	States	MinSize	Ax.Al.	Linear	Poly	PolyPrio1
cartpole [Jag+20]	271	**169**	253 263	183 187	243 **169**	189 **169**
10rooms [JZ17]	26,244	**49**	17,297 17,297	147 107	61 **49**	61 **49**
helicopter [Jag+20]	280,539	475	6,339 9,649	**3,769** 4,637	5,035 TO	3,787 TO
cruise_250 [LMT15a]	320,523	9	869 1,067	369 363	353 **11**	37 25
dcdc [RZ16]	593,089	5	271 265	139 173	**129** 147	199 273
truck_trailer [KZ19]	1,386,211	1,839	**338,283** 366,411	TO TO	TO TO	TO TO
aircraft [RWR15]	2,135,056	31	915,877 1,015,903	916,685 1,013,949	725,011 688,577	**602,335** 630,631
pacman.5	232	**37**	53 81	49 59	47 **37**	**37** **37**
philosophers-mdp.3	344	59	391 403	333 367	315 251	251 **223**
ij.10	1,013	19	1,291 1,405	753 735	897 **141**	209 177
elevators.a-11-9	14,742	129	16,341 17,809	9,865 9,955	9,779 2,023	2,859 **1,919**
exploding-blocksworld.5	76,741	149	16,913 20,273	2,687 2,845	4,511 TO	**829** TO
wlan_dl.0.80.deadline	189,641	175	3,369 3,675	701 2,841	693 **523**	667 TO
pnueli-zuck.5	303,427	173	171,371 263,955	156,165 221,645	114,979 95,879	**83,219** 83,951

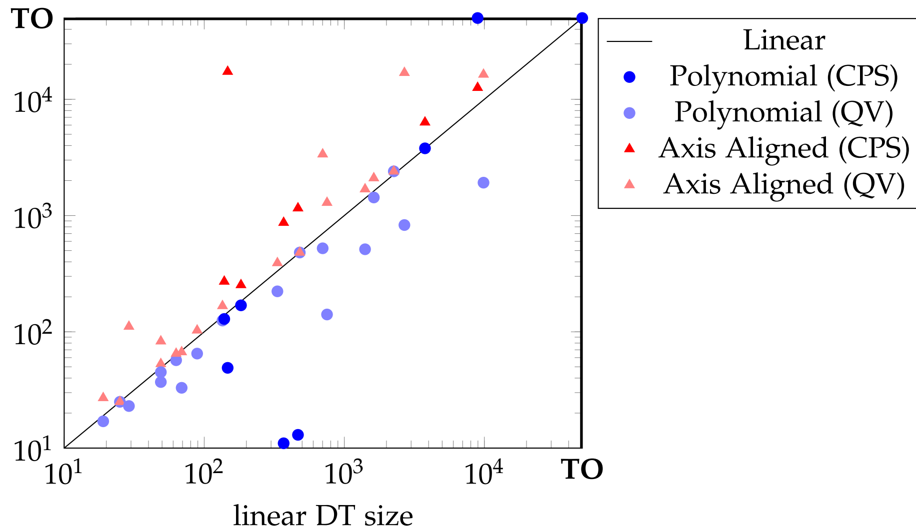


Figure 7.2.: Performance comparison of different predicate types. Based on the decision tree size using linear predicates, we compare how many nodes the decision trees with axis-aligned splits and quadratic polynomials have. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV).

7.2.4. Min-Label Entropy and Predicate Priority

We applied two significant changes to arrive at the small decision trees in the `cruise` example: the min-label entropy impurity measure and the modified predicate priority value. We now analyze how useful they are for the other case studies.

In Figure 7.3 we again make use of a logarithmic scatter plot to visualize the data from our tables. As a baseline, we take the size of the decision tree generated with our proposed approach using the entropy impurity measure and the default priority 1.0. We compare it to the size when using the proposed min-label entropy (blue) and when using the reduced priority value 0.1 (red).

Table 7.3.: Cumulated statistics over all **28** benchmarks. We compare the best linear strategy with entropy impurity with the best of our heuristics.

	Linear	Quadratic
Timeout	1	2
Minimal DT	**2** (7%)	**10** (35%)
DT is smaller or equal		**25** (89%)
DT has less than half the size		**8** (29%)

Min-Label Entropy The min-label entropy reduces the tree size in 14 out of 17 cases (82%) where we are not already at the minimum size and do not run into a timeout. Interestingly, this behavior is different when using the min-label entropy with axis-aligned splits or linear splits. There, the min-label entropy can only improve the result in 30 out of 106 cases (28%).

Also, we observe 5 cases where our approach only times out when using the min-label entropy but not when using the standard entropy. A reason for this might be that the min-label entropy encourages the formation of decision trees formed like a line. For all case studies where we generate minimum-sized trees like the `10rooms` case study, every leaf has a unique label. With the min-label entropy impurity, every splitting predicate separates out one of those labels. So the tree looks like a line. As a consequence, the runtime for finding predicates does not decrease as fast while constructing the tree. When we construct a perfectly balanced tree, the size of the dataset left at the subtree at depth d is only a small fraction (2^{-d}) of the original size. In the case of a line, however, the dataset size only decreases slowly.

Low Priority Heuristic While the low priority value helps in the cruise example in combination with the min-label entropy, the only other cases where this heuristic brings an improvement are the `dcdc` and `eajs.2.100.5.ExpUtil`[3] case studies. We conclude that our motivating idea of first separating the "outliers" and then using the more sophisticated splits later does not generalize well. Apparently, it is beneficial to just take the best available split right away in complex models.

7.2.5. Explainability

We have seen that we can significantly reduce the number of decision tree nodes with our proposed approach. But how explainable are the trees we generate?

Of course, reducing the number of decision nodes already helps create an explainable decision tree. Still, we have to consider that the complexity of the individual splitting predicate increases, thereby potentially reducing explainability. As an example, we consider the `10rooms` case study. Here, we find a decision tree with 49 nodes which is the minimum size for a most-permissive decision tree. Unfortunately, the decision tree is not particularly explainable as some predicates comprise up to 35 terms, even after trying to round coefficients to zero. The reason being the large number of 10 state variables. A quadratic polynomial with ten variables can already have 65 terms.

Regardless of the complexity of individual predicates, for some case studies, the minimum decision tree size is already too large to be easily understandable by a human. Any most-permissive decision tree for the case studies `helicopter` and `truck_trailer` will have more than 400 and 1,800 nodes respectively. So, in these cases, we might need to investigate determinized controllers as discussed in [Ash+20; Ash+21].

[3] see Appendix E

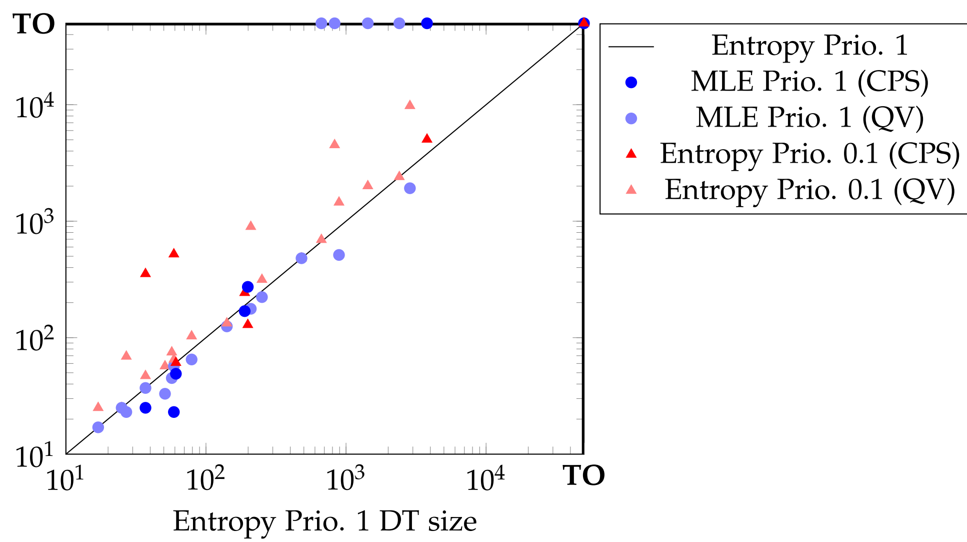


Figure 7.3.: Performance comparison of the min-label entropy (MLE) and the low priority heuristic. Based on the decision tree size using quadratic polynomials as predicates with entropy and priority 1.0, we compare how the heuristics change the tree size. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV).

8. Future Work

The tool `dtcontrol` achieves great performance generating compact decision trees, especially when using the determination heuristics proposed in [Ash+20; Ash+21]. In this work, we showed that more expressive quadratic polynomials can especially help generate succinct trees for most-permissive controllers. An important part that can still be improved upon, however, is the explainability. Of course, succinct decision trees are already easier to understand by nature, but more complex predicates again reduce explainability. Ideally, we would want to have a justification explaining the coefficients of the complex predicate. Also, synthesizing other types of functions and improving the current curve fitting are interesting directions for future work.

Predicate Generation From Domain Knowledge We have seen that predicate generation from general domain knowledge without an effective heuristic is infeasible. If existent, such a heuristic would probably have to use a completely different approach, as the impurity approach taken in this thesis does not seem to be a promising choice. With regards to the problem of uniqueness we discovered in Section 5.4, a key component for a heuristic might be the model checker or at least the dynamics equations of the system. With that, simulations could be used to judge if a predicate describes a relevant behavior of the system.

A different approach would be to use the polynomials we found in this work. Specifically, if we exactly know the coefficients where we want to arrive, it might be easier to find a derivation for that with some kind of meet-in-the-middle technique.

Feature Importance As we have described in Section 6.3, the proposed feature importance is far from perfect. The current implementation depends on the order of features. Future work could analyze this dependence and propose countermeasures. In the bigger picture, the feature importance could be lifted from a binary "relevant"/"irrelevant" classification to a continuous spectrum that could be used in the decision tree construction.

Decision Tree Balance We proposed the min-label entropy impurity measure that focuses on separating one label first. As a result, we encourage the formation of unbalanced, line-like decision trees (see Subsection 7.2.4). However, more balanced decision trees might be easier to understand by a human. Additionally, we might reduce the individual predicate complexity (number of terms in the polynomial) if we use more general splits first. For that, we would need to develop an impurity measure that prefers

split A from Figure 8.1 over split B, just like the standard entropy does, but still keeps the advantages of the min-label entropy and finds trees of minimum size.

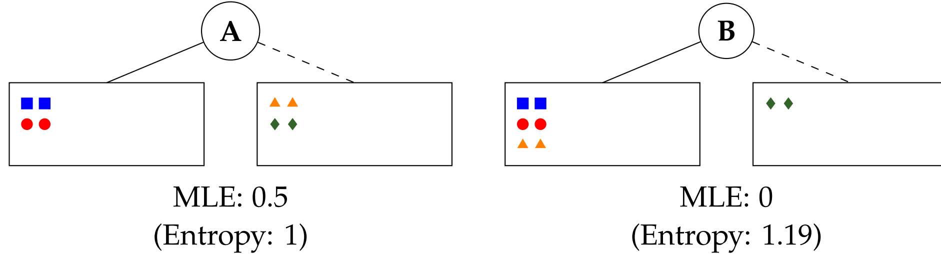


Figure 8.1.: Two different splits with their respective min-label entropy (MLE) and standard entropy values. Split A leads to a more balanced tree and has a better entropy value but split B is preferred by the min-label entropy as one label is completely separated.

More Expressive Predicates In this work, we restricted ourselves to quadratic polynomials as predicates generated by the support vector machines. To mitigate the problems we discussed in Subsection 6.2.1, techniques such as principal component analysis could be used to significantly reduce the dimensionality. This would allow us to use higher degree polynomials, maybe even other types of functions.

Analysis of Other Case Studies The `cruise` model has been extensively discussed in other work [Ash+19b; Wei20; Ash+21] as well as this thesis and we have found adequate solutions. However, for case studies such as `helicopter` [Jag+20] and `aircraft` [RWR15] the decision tree representation is still larger than the BDD representation (see Appendix E). A better understanding of the model could lead to new realizations. Maybe other types of functions as predicates are needed or our impurity measures do not work sufficiently well for those cases.

Improved Curve Fitting In Section 6.1 we show the difference between the current curve-fitting implementation and our new approach. In principle, the current approach uses regression analysis to find a function that fits the data with least-square-fitting. Our approach, however, tries to find a function separating the data instead of fitting it. As we have seen great success with our approach, we might be able to transfer some insight to the general curve fitting that is used to determine unspecified constants in user-entered expressions.

9. Conclusion

In this work, we have investigated two approaches to generating expressive algebraic splitting predicates for decision trees. We have seen that automatically generating predicates from domain knowledge is not yet feasible with the current method. Hence we proposed learning quadratic polynomials with support vector machines directly from the controller data. Additionally, we introduced a new impurity measure called *min-label entropy* that focuses on separating one specific label first. We integrated both ideas into the open-source tool `dtcontrol` and were able to generate significantly smaller decision trees in cases where the determination heuristics could not be applied. For the cruise model, we generated a tree with the same size a the one created with help of a human expert, and in 10 out of 28 case studies, we even found a decision tree of minimum size.

List of Figures

1.1. Example showing how different types of predicates can separate a dataset. 3

3.1. An example of how a decision tree can represent a controller. (a) shows a determinized controller, (b) a permissive one with multiple safe actions at some states. 8
3.2. The sample dataset with different separating hyperplanes. 11
3.3. Projecting $\vec{x}$ onto the normal direction $\vec{w}$. 11
3.4. A linearly inseparable dataset becomes linearly separable by transforming it into a higher-dimensional space. Transforming the decision function back gives a quadratic polynomial. 13

4.1. An illustration of the cruise model. Source: [LMT15a] 15
4.2. A velocity-time graph showing the ego vehicle accelerating for one time step. The area between the blue and the red curve describes the change in distance between the two cars. 17

5.1. A derivation of the "can-accelerate" predicate. 22
5.2. The handcrafted predicate d^* and the terms used to derive it, plotted for a fixed value of $v_f = 2$. While the sum of the terms is a perfect classifier, individually, they are not helpful for the classification. 23

6.1. Visualization of a handcrafted predicate perfectly separating the data. . . 25
6.2. In the current curve-fitting implementation, a two-dimensional dataset (a) is mapped to a three-dimensional space where the $z \in \{-1, 1\}$ is determined by the label. The old approach then fits a function to the new dataset. Our approach instead tries to separate the data like the gray surface does in (b). 26
6.3. Examples with redundant features. In (a) x_2 is not needed. In (b) x_1 and x_2 individually look redundant, but only one may be removed. Also, removing x_1 is preferred over removing x_2 29
6.4. Two different splits with their respective entropy values. While split B has a better entropy value and is preferred in machine learing, we want to use split A first when building a perfect classifier. 32
6.5. The decision trees needed for perfect classification with the splits from Figure 6.4. 32

7.1. The decision trees for the `cruise` example generated by our data-driven approach. 37
7.2. Performance comparison of different predicate types. Based on the decision tree size using linear predicates, we compare how many nodes the decision trees with axis-aligned splits and quadratic polynomials have. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV). . . . 40
7.3. Performance comparison of the min-label entropy (MLE) and the low priority heuristic. Based on the decision tree size using quadratic polynomials as predicates with entropy and priority 1.0, we compare how the heuristics change the tree size. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV). 42

8.1. Two different splits with their respective min-label entropy (MLE) and standard entropy values. Split A leads to a more balanced tree and has a better entropy value but split B is preferred by the min-label entropy as one label is completely separated. 44

A.1. Calculating the margin size d by projecting onto the direction $\vec{w}$ 60
A.2. Finding a minimum with an active constraint. If $\vec{\nabla} f$ and $\vec{\nabla} c$ are not parallel, we find a descending step (shown in orange). 62

List of Tables

7.1. The decision tree sizes (number of nodes) while using different sets of predicates for the cruise_250 dataset. The column "Sum?" describes if we use Step 3 of our generation approach from Section 5.1. 36
7.2. The number of nodes of the generated decision trees using axis-aligned splits, linear splits, and the proposed quadratic polynomial splits with priority 0.1 and 1.0. Each row displays the result using the entropy impurity measure at the top and using min-label entropy at the bottom. TO means time out after 3 hours. As a comparison, we show the number of states of the underlying controller and the minimum size a decision tree needs to have. The full table is in Appendix E 39
7.3. Cumulated statistics over all **28** benchmarks. We compare the best linear strategy with entropy impurity with the best of our heuristics. 40

B.1. The parameters used for generating the controllers of the cruise model and the resulting sizes measured in number of states and number of state-action pairs. 65

E.1. Benchmark results for the cyber-physical system case studies. 72
E.2. Benchmark results for case studies from the quantitative verification benchmark set (part 1). 73
E.3. Benchmark results for case studies from the quantitative verification benchmark set (part 2). 74

Bibliography

[Akm19] S. M. Akmese. "Generating Richer Predicates for Decision Trees". Bachelor's thesis. Technical University of Munich, 2019.

[Arl94] S. Arlinghaus. *Practical Handbook of Curve Fitting*. Taylor & Francis, 1994. ISBN: 9780849301438.

[Ash+19a] P. Ashok, T. Brázdil, K. Chatterjee, J. Křetínský, C. H. Lampert, and V. Toman. "Strategy Representation by Decision Trees with Linear Classifiers". In: *Quantitative Evaluation of Systems*. Ed. by D. Parker and V. Wolf. Cham: Springer International Publishing, 2019, pp. 109–128. ISBN: 978-3-030-30281-8. DOI: 10.1007/978-3-030-30281-8_7.

[Ash+20] P. Ashok, M. Jackermeier, P. Jagtap, J. Křetnský, M. Weininger, and M. Zamani. "DtControl: Decision Tree Learning Algorithms for Controller Representation". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. HSCC '20. Sydney, New South Wales, Australia: Association for Computing Machinery, 2020. ISBN: 9781450370189. DOI: 10.1145/3365365.3382220.

[Ash+21] P. Ashok, M. Jackermeier, J. Kretnský, C. Weinhuber, M. Weininger, and M. Yadav. "dtControl 2.0: Explainable Strategy Representation via Decision Tree Learning Steered by Experts". In: Lecture Notes in Computer Science 12652 (2021). Ed. by J. F. Groote and K. G. Larsen, pp. 326–345. DOI: 10.1007/978-3-030-72013-1_17.

[Ash+19b] P. Ashok, J. Kretnský, 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.

[SHB00] R. St-Aubin, J. Hoey, and C. Boutilier. "APRICODD: Approximate Policy Construction Using Decision Diagrams". In: *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, 2000, pp. 1089–1095. URL: https://proceedings.neurips.cc/paper/2000/hash/201d7288b4c18a679e48b31c72c30ded-Abstract.html.

[Bah+97] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 171–206. DOI: 10.1023/A:1008699807402.

[BBC16] BBC. *Hackers caused power cut in western Ukraine*. Jan 12, 2016. URL: https://www.bbc.com/news/technology-35297464.

[BB98] K. P. Bennett and J. A. Blue. "A support vector machine approach to decision trees". In: *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*. Vol. 3. 1998, 2396–2401 vol.3. DOI: 10.1109/IJCNN.1998.687237.

[Bil16] J. Billington. *Nest not working: Smart thermostat bug plunges customers into cold*. Jan 14, 2016. URL: https://www.ibtimes.co.uk/google-owned-nest-thermostat-plunges-customers-into-cold-after-software-glitch-153797.

[BW96] B. Bollig and I. Wegener. "Improving the Variable Ordering of OBDDs Is NP-Complete". In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122.

[Brá+15] T. Brázdil, K. Chatterjee, M. Chmelik, A. Fellner, and J. Kretnský. "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.

[Bry86] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[CPS20] N. Chan, E. Polgreen, and S. A. Seshia. "Gradient Descent over Metagrammars for Syntax-Guided Synthesis". In: *CoRR* abs/2007.06677 (2020). arXiv: 2007.06677. URL: https://arxiv.org/abs/2007.06677.

[CL11] C.-C. Chang and C.-J. Lin. "LIBSVM: A library for support vector machines". In: *ACM Trans. Intell. Syst. Technol.* 2.3 (2011), 27:1–27:27. DOI: 10.1145/1961189.1961199.

[CHL08] K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. "Coordinate Descent Method for Large-scale L2-loss Linear Support Vector Machines". In: *J. Mach. Learn. Res.* 9 (2008), pp. 1369–1398. URL: https://dl.acm.org/citation.cfm?id=1442778.

[Cha+10] Y.-W. Chang, C.-J. Hsieh, K.-W. Chang, M. Ringgaard, and C.-J. Lin. "Training and Testing Low-degree Polynomial Data Mappings via Linear SVM". In: *J. Mach. Learn. Res.* 11 (2010), pp. 1471–1490. URL: http://portal.acm.org/citation.cfm?id=1859899.

[CE07] I. T. Christou and S. Efremidis. "An Evolving Oblique Decision Tree Ensemble Architecture for Continuous Learning Applications". In: *Artificial Intelligence and Innovations 2007: from Theory to Applications, Proceedings of the 4th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI 2007), 19-21 September 2007, Peania, Athens, Greece*. Ed. by C. Boukis, A. Pnevmatikakis, and L. Polymenakos. Vol. 247. IFIP. Springer, 2007, pp. 3–11. DOI: 10.1007/978-0-387-74161-1_1.

[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.

[DS02] D. DeCoste and B. Schölkopf. "Training Invariant Support Vector Machines". In: *Mach. Learn.* 46.1-3 (2002), pp. 161–190. DOI: 10.1023/A:1012454411458.

[Deh+17] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by R. Majumdar and V. Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 592–600. DOI: 10.1007/978-3-319-63390-9_31.

[Fan+08] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. "LIBLINEAR: A Library for Large Linear Classification". In: *J. Mach. Learn. Res.* 9 (2008), pp. 1871–1874. URL: https://dl.acm.org/citation.cfm?id=1442794.

[Har+19] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters. "The Quantitative Verification Benchmark Set". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by T. Vojnar and L. Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 344–350. DOI: 10.1007/978-3-030-17462-0_20.

[HTF09] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. ISBN: 9780387848570. DOI: 10.1007/978-0-387-84858-7.

[HKO19] E. Hemberg, J. Kelly, and U.-M. O'Reilly. "On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 1039–1046. ISBN: 9781450361118. DOI: 10.1145/3321707.3321865.

[Hsi+08] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. "A dual coordinate descent method for large-scale linear SVM". In: *Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008*. Ed. by W. W. Cohen, A. McCallum, and S. T. Roweis. Vol. 307. ACM International Conference Proceeding Series. ACM, 2008, pp. 408–415. DOI: 10.1145/1390156.1390208.

[Isi13] C. Isidore. *Toyota settles acceleration case after $3 million jury verdict*. Oct 25, 2013. URL: https://money.cnn.com/2013/10/25/news/companies/toyota-crash-verdict/.

[IS96] A. Ittner and M. Schlosser. "Non-Linear Decision Trees - NDT". In: *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*. Ed. by L. Saitta. Morgan Kaufmann, 1996, pp. 252–257.

[Jac20] M. Jackermeier. "dtControl: Decision Tree Learning for Explainable Controller Representation". Bachelor's thesis. Technical University of Munich, 2020.

[Jag+20] P. Jagtap, F. Abdi, M. Rungger, M. Zamani, and M. Caccamo. "Software Fault Tolerance for Cyber-Physical Systems via Full System Restart". In: *ACM Trans. Cyber Phys. Syst.* 4.4 (2020), 47:1–47:20. URL: https://dl.acm.org/doi/10.1145/3407183.

[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.

[Jün21] F. Jüngermann. *Learning Algebraic Predicates for Explainable Controllers: Artifacts*. May 2021. DOI: 10.5281/zenodo.4746131.

[KZ19] M. Khaled and M. Zamani. "pFaces: an acceleration ecosystem for symbolic control". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*. Ed. by N. Ozay and P. Prabhakar. ACM, 2019, pp. 252–257. DOI: 10.1145/3302504.3311798.

[KT51] H. Kuhn and A. Tucker. "Nonlinear Programming". In: *Second Berkeley Symposium on Mathematical Statistics and Probability*. 1951, pp. 481–492.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[LMT15a] 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.

[LMT15b] K. G. Larsen, M. Mikučionis, and J. H. Taankvist. *Safe and Optimal Cruise Control: Website*. Sep 16, 2015. URL: https://people.cs.aau.dk/~marius/stratego/cruise.html.

[Lev44] K. Levenberg. "A method for the solution of certain non-linear problems in least squares". In: *Quarterly of applied mathematics* 2.2 (1944), pp. 164–168.

[LWK07] C.-J. Lin, R. C. Weng, and S. S. Keerthi. "Trust region Newton methods for large-scale logistic regression". In: ACM International Conference Proceeding Series 227 (2007). Ed. by Z. Ghahramani, pp. 561–568. DOI: 10.1145/1273496.1273567.

[Man02] O. L. Mangasarian. "A finite newton method for classification". In: *Optim. Methods Softw.* 17.5 (2002), pp. 913–929. DOI: 10.1080/1055678021000028375.

[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.

[Mit+97] T. M. Mitchell et al. *Machine learning*. McGraw-hill New York, 1997.

[Mor+20] K. Morton, W. T. Hallahan, E. Shum, R. Piskac, and M. Santolucito. "Grammar Filtering for Syntax-Guided Synthesis". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 1611–1618. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5522.

[MKS94] S. K. Murthy, S. Kasif, and S. Salzberg. "A System for Induction of Oblique Decision Trees". In: *J. Artif. Intell. Res.* 2 (1994), pp. 1–32. DOI: 10.1613/jair.63.

[NW00] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2000. ISBN: 9780387987934.

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=2078195.

[Pra+04] S. S. Pradhan, W. H. Ward, K. Hacioglu, J. H. Martin, and D. Jurafsky. "Shallow Semantic Parsing using Support Vector Machines". In: *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2004, Boston, Massachusetts, USA, May 2-7, 2004*. Ed. by J. Hirschberg, S. T. Dumais, D. Marcu, and S. Roukos. The Association for Computational Linguistics, 2004, pp. 233–240. URL: https://www.aclweb.org/anthology/N04-1030/.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

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

[RWR15] M. Rungger, A. Weber, and G. Reissig. "State space grids for low complexity abstractions". In: *54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*. IEEE, 2015, pp. 6139–6146. DOI: 10.1109/CDC.2015.7403185.

[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.

[Sha09] A. Shashua. *Introduction to Machine Learning: Class Notes 67577*. 2009. arXiv: 0904.3664 [cs.LG].

[SZ19] A. Swikir and M. Zamani. "Compositional Synthesis of Symbolic Models for Networks of Switched Systems". In: *IEEE Control. Syst. Lett.* 3.4 (2019), pp. 1056–1061. DOI: 10.1109/LCSYS.2019.2920766.

[Vap00] V. N. Vapnik. *The Nature of Statistical Learning Theory, Second Edition*. Statistics for Engineering and Information Science. Springer, 2000. ISBN: 978-0-387-98780-4.

[Wei20] C. Weinhuber. "Learning Domain-Specific Predicates in Decision Trees for Explainable Controller Representation". Bachelor's thesis. Technical University of Munich, 2020.

[Win10] P. Winston. *6.034 Artificial Intelligence, Lecture 16*. Massachusetts Institute of Technology: MIT OpenCourseWare, Fall 2010. URL: https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

[ZVJ18] I. S. Zapreev, C. Verdier, and M. M. Jr. "Optimal Symbolic Controllers Determinization for BDD storage". In: IFAC-PapersOnLine 51.16 (2018). Ed. by A. Abate, A. Girard, and M. Heemels, pp. 1–6. DOI: `10.1016/j.ifacol.2018.08.001`.

A. Support Vector Machine Details

In this chapter, we have a deeper look at support vector machines. In Section A.1 we describe how we can formulate the search for the maximum-margin hyperplane as a constrained optimization problem. Then, in the following Section A.2 we analyze how we can reformulate such a constrained problem more nicely and arrive at the *dual* formulation that is later used for the kernel trick. In Section A.3 we generalize the support vector machine to handle cases where we cannot perfectly separate the data by introducing a loss function we want to minimize. There, we introduce the formulation of the problem that is used by most support vector machine libraries.

A.1. Minimization Problem

We now want to find the *maximum-margin hyperplane*. For that, lets suppose we have a dataset $\vec{x}_i \in \mathbb{R}^M$ with corresponding positive and negative labels $y_i \in \{-1, +1\}$ for $i \in \{1, \ldots, N\}$ that can be perfectly separated by a hyperplane.

We know that for points on the plane $\vec{w} \cdot \vec{x} - b = 0$. For the data points closest to the plane (the support vectors) we want $|\vec{w} \cdot \vec{x}_s - b| = 1$ for future convenience. We can always achieve this normalization by scaling $\vec{w}$. By using our labels $y_i \in \{-1, +1\}$, we can rewrite it to:

$$y_s(\vec{w} \cdot \vec{x}_s - b) = 1 \tag{A.1}$$

For all other data points, this expression has to be even larger. So in general, we have the condition

$$y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1 \tag{A.2}$$

for all i.

Now, how large is the margin between the the positive and negative samples? As described in [Win10], we can calculate the distance d by looking at the connecting vector from a negative support vector $\vec{x}_n$ to a positive support vector $\vec{x}_p$ and projecting it onto the normal direction $\vec{w}$ (see Figure A.1):

$$\begin{aligned} d &= (\vec{x}_p - \vec{x}_n)\frac{\vec{w}}{|\vec{w}|} \\ &= ((1+b) - (b-1))\frac{1}{|\vec{w}|} \\ &= \frac{2}{|\vec{w}|} \end{aligned} \tag{A.3}$$

where we used Equation A.1 to evaluate $\vec{w} \cdot \vec{x}_p$ and $\vec{w} \cdot \vec{x}_n$ as we know $y_p = 1$ and $y_n = -1$.

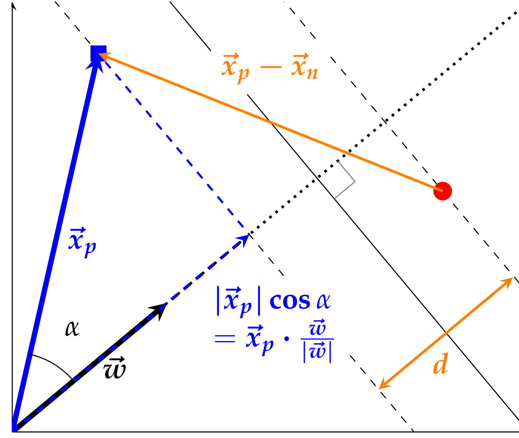


Figure A.1.: Calculating the margin size d by projecting onto the direction $\vec{w}$

So, to maximize the margin distance d, we need to minimize $|\vec{w}|$ while adhering to Equation A.2.

A.2. Lagrangian Function and Dual Problem

To solve this constrained minimization problem, recall that $|\vec{w}| = \sqrt{\sum_j w_j^2}$. As the square root function is strictly monotonically increasing, we can instead minimize the radicand. So we formulate the equivalent constrained quadratic optimization problem [HTF09, Section 4.5.2]:

$$\min \frac{1}{2}|\vec{w}|^2 \tag{A.4a}$$

$$\text{subject to} \quad y_i(\vec{x}_i \cdot \vec{w} - b) \geq 1, \quad i = 1, \ldots, N \tag{A.4b}$$

How do we solve such a constrained problem? Of course, there are tools like [Fan+08] that use iterative methods such as *coordinate descent* [CHL08] for problems formulated like this, or more specifically for the *soft margin* formulation of this problem that we encounter in section A.3. Still, we want to spend some time analyzing the specific problem to find an *dual* formulation crucial for the success of support vector machines.

Ignoring the constraints for a second, we note that the function to be minimized $f(\vec{w}, b) := \frac{1}{2}|\vec{w}|^2$ is convex. We find the global minimum by setting $\vec{\nabla} f = \vec{0}$. In our example, the unconstrained minimum is at $\vec{w} = \vec{0}$ which clearly does not satisfy our constraints. Now, consider one linear constraint $c_i(\vec{w}, b) := y_i(\vec{x}_i \cdot \vec{w} - b) - 1$ with $c_i(\vec{w}, b) \geq 0$.

Definition 5. Similar to [NW00, Chapter 12.1], we define call a constraint $c_i(\vec{w}, b)$ *active* for $(\vec{w}^*, b^*)$, if $c_i(\vec{w}^*, b^*) = 0$. Otherwise, if $c_i(\vec{w}^*, b^*) > 0$, we call c_i *inactive*.

We can then distinguish those two cases when looking for an optimal solution $(\vec{w}^*, b^*)$ [NW00, Chapter 12.1]:

- **The constraint is inactive** ($c_i(\vec{w}^*, b^*) > 0$): In this case,

  $$\vec{\nabla} f(\vec{w}^*, b^*) = \vec{0} \tag{A.5}$$

  must hold. Otherwise, we could move in the opposite direction of the gradient without violating the linear constraint to find a smaller solution $\vec{w}^* - \alpha \vec{\nabla} f(\vec{w}^*, b^*)$ for a sufficiently small $\alpha > 0$.

- **The constraint is active** ($c_i(\vec{w}^*, b^*) = 0$): In this case, we may not be able to freely move in the direction of the gradient. So we can have a local minimum where the gradient is not null. However, we know that the gradient must not have a component in the direction parallel to our constraint, otherwise we could again move in that direction. This means, the gradient of f and the gradient of c_i (the normal vector) must be parallel at position $(\vec{w}^*, b^*)$:

  $$\vec{\nabla} f = \lambda_i \vec{\nabla} c_i, \quad \text{for some } \lambda_i \geq 0. \tag{A.6}$$

  To see why λ_i must be nonnegative, we observe that $\vec{\nabla} c_i$ points towards positions with $c_i > 0$ which satisfy the constraint. If we want to have a minimum at the border, $\vec{\nabla} f$ must point in the same direction, describing that f will increase in that direction (see Figure A.2).

  Also note that equation A.5 is equivalent to A.6 with $\lambda_i = 0$.

Lagrangian Function For now, we assume c_i is active in the solution. We can then define the *Lagrangian function*:

$$\mathcal{L}(\vec{w}, b, \lambda_i) = f(\vec{w}, b) - \lambda_i c_i(\vec{w}, b) \tag{A.7}$$

The main idea is that we encode our active contrains into the Lagrangian function and thereby reduce the problem to an unconstrained optimization problem. Let us verify this claim by calculating the necessarily conditions for an extremum of $\mathcal{L}$:

- $\partial_{\lambda_i} \mathcal{L} = 0$: this is equivalent to $c_i(\vec{w}, b) = 0$, our active constraint.

- $\partial_{\vec{w}, b} \mathcal{L} = 0$: this is equivalent to $\vec{\nabla} f = \lambda_i \vec{\nabla} c_i$, equation A.6 (we still need to require $\lambda_i \geq 0$).

This means we can find candidates for minima by finding extremums of $\mathcal{L}$ as long as we enforce $\lambda_i \geq 0$ and know which constraints are active. By extending this notion

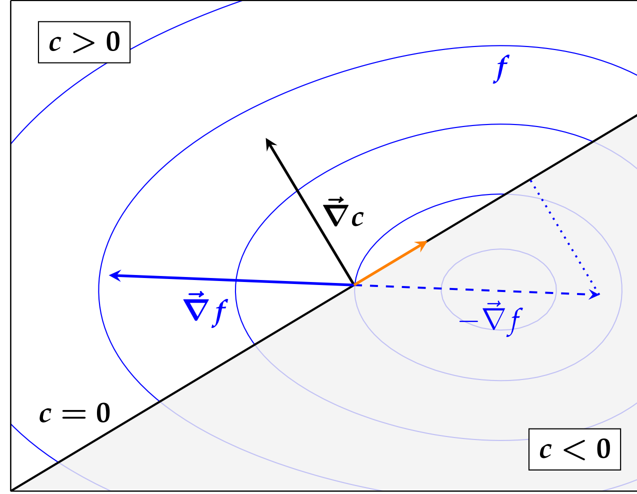


Figure A.2.: Finding a minimum with an active constraint. If $\vec{\nabla} f$ and $\vec{\nabla} c$ are not parallel, we find a descending step (shown in orange).

to multiple constraints as shown in [NW00, Chapter 12], we receive the following Lagrangian equation for our problem:

$$\mathcal{L} = \frac{1}{2}|\vec{w}|^2 - \sum_{i=1}^{N} \lambda_i (y_i(\vec{x}_i \cdot \vec{w} - b) - 1) \tag{A.8}$$

If we again set the partial derivatives to zero, we receive the following insightful relationships:

$$\frac{\partial \mathcal{L}}{\partial \vec{w}} = \vec{w} - \sum_{i=1}^{N} \lambda_i y_i \vec{x}_i = 0 \Rightarrow \vec{w} = \sum_{i=1}^{N} \lambda_i y_i \vec{x}_i \tag{A.9}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{N} \lambda_i y_i = 0 \tag{A.10}$$

Recall that $\lambda_i = 0$ if constraint i is inactive and $\lambda_i \geq 0$ if it is active. This means we can express $\vec{w}$ as a linear combination of that sample points x_i that have active constraints. In fact, these will be our support vectors.

Dual Problem As a last step, we can use relations A.9 and A.10 to substitute $\vec{w}$ in equation A.8:

$$\begin{aligned}\mathcal{L} &= \frac{1}{2}\left(\sum_{i=0}^{N} \lambda_i y_i \vec{x}_i\right)\left(\sum_{j=0}^{N} \lambda_j y_j \vec{x}_j\right) - \left(\sum_{i=1}^{N} \lambda_i y_i \vec{x}_i\right)\left(\sum_{j=1}^{N} \lambda_j y_j \vec{x}_j\right) + b\sum_{i=1}^{N} \lambda_i y_i + \sum_{i=1}^{N} \lambda_i \\ &= \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \vec{x}_i \cdot \vec{x}_j \end{aligned} \tag{A.11}$$

with the additional constraints $\sum_{i=1}^{N} \lambda_i y_i = 0$ (see A.10), $\lambda_i \geq 0$ (see A.6), and either condition i has to be active or $\lambda_i = 0$, which can be expressed as

$$\lambda_i(y_i(\vec{x}_i \cdot \vec{w}^* - b^*) - 1) = 0 \tag{A.12}$$

These equations form the Karush-Kuhn-Tucker conditions [KT51] and are necessary for $(\vec{w}^*, b^*)$ to be a minimum.

Equation A.11 is an alterative way of formulating A.4 and is called the *dual* problem to the *primal* formulation A.4. Interestingly, $\mathcal{L}$ only depends on the dot-products of $\vec{x}_i$. This fact enables the so-called *kernel trick* that we discus in Subsection 3.3.2.

A.3. Soft-Margin

We have seen that working in a higher-dimensional space can make a dataset linearly separable. In general, though, the data might still not be linearly separable in the higher dimension due to noisy or erroneous data. Even if 1% of our data is erroneous, we still want to find a classifier that predicts correct labels for 99% of data points.

Definition 6. We define the *L1 - hinge loss* function ξ for a hyperplane with $\vec{w}^*$ and b^*, and a sample point $\vec{x}_i, y_i$ as

$$\xi(\vec{w}^*,\, b^*,\, \vec{x}_i,\, y_i) = \max(0,\ 1 - y_i(\vec{w}^* \cdot \vec{x}_i - b^*)) \tag{A.13}$$

As we have seen in equation A.2, we can satisfy $y_i(\vec{w}^* \cdot \vec{x}_i - b^*) \geq 1$ in the linearly separable case, so $\xi = 0$. Otherwise, ξ is a measure of how far away x_i is from the correct side. We can now frame our optimization problem (A.4) as the following *unconstrained* problem:

$$\min\ \frac{1}{2}|\vec{w}|^2 + C\sum_{i=1}^{N} \xi(\vec{w}, b, \vec{x}_i, y_i) \tag{A.14}$$

with a constant $C \in \mathbb{R}$. We can use C to balance the size of the margin and the accuracy of the classification.

At first sight, an unconstrained problem looks beneficial. But at the same time, we have lost the smoothness of our function as ξ is not differentiable everywhere. This is why oftentimes an alternative formulation as a smooth constrained problem is used literature [HTF09, Chapter 12.2] as well as when implementing solvers [CL11]. However recently, there has also been work directly using this formulation [Hsi+08].

A variation of the hinge loss function is the *L2 - hinge loss* function $\xi_2 = \xi^2$ used in [Fan+08; CHL08] and hence also utilized in `dtcontrol`. By taking the square, ξ_2 is differentiable, but not twice differentiable [Man02].

B. The Cruise Control Model

B.1. Cruise Control Modifications

The cruise control system is modeled in the tool `UPPAAL Stratego` [Dav+15]. The model was introduced in [LMT15a] and is available to download on the respective website [LMT15b]. We made the following small adjustment just like in previous work [Ash+20; Ash+21]. As the accelerations are $-2, 0$ or 2 and the time step is 1, all velocities occurring in the model are even. However, when a car appears from the far-away state, it can also have an odd number velocity. To keep the even velocities, we changed the source code in lines 242ff. to:

```
242 i:int[minVelocityFront/2, maxVelocityFront/2]
243 2*i &lt;= velocityEgo
244 velocityFront = i*2,
245 distance = maxSensorDistance,
246 rVelocityFront = 2 * i * 1.0,
247 rDistance = 1.0*maxSensorDistance
```

You find the modified model file in [Jün21].

B.2. Cruise Control Parameters

We use other parameters for the `cruise` example than previous work [Ash+20; Ash+21]. Table B.1 contains an overview of the parameters used and the resulting size of the controller measured in number of states and number of state-action pairs. The generated controllers are included in [Jün21].

Table B.1.: The parameters used for generating the controllers of the cruise model and the resulting sizes measured in number of states and number of state-action pairs.

	Parameters			Controller Size	
Name	v_{min}	v_{max}	d_{max}	#states	#s-a pairs
`cruise_prev`	-10	20	200	295,615	886,845
`cruise_250`	-6	20	250	320,523	961,569
`cruise_300`	-10	20	300	500,920	1,502,760

C. Predicate Generation

C.1. Base Identities

These 8 base identities are used when generating new predicates:

$$
\begin{aligned}
a &= \frac{2(d - tv)}{t^2} \\
a &= \frac{v}{t} \\
t &= \frac{-v + \sqrt{2ad + v^2}}{a} \\
t &= -\frac{v + \sqrt{2ad + v^2}}{a} \\
t &= \frac{v}{a} \\
v &= -\frac{at}{2} + \frac{d}{t} \\
v &= at \\
d &= \frac{at^2}{2} + vt
\end{aligned}
$$

C.2. Handcrafted Predicate

The non-simplified predicate deciding whether the ego vehicle can accelerate in the next time step is:

$$
\begin{aligned}
& d_{one} + d_{f1} - d_e + d_{f2} + d_r \geq d_{safe} \\
\Leftrightarrow & \\
& (a_{min} - a_{max})t_1^2/2 + (v_f - v_e)t_1 \\
& + a_{min}((v_{min} - (v_f + a_{min}t_1))/a_{min})^2/2 + (v_f + a_{min}t_1)(v_{min} - (v_f + a_{min}t_1))/a_{min} \\
& - a_{min}((v_{min} - (v_e + a_{max}t_1))/a_{min})^2/2 - (v_e + a_{max}t_1)(v_{min} - (v_e + a_{max}t_1))/a_{min} \\
& + v_{min}\left[(v_{min} - (v_e + a_{max}t_1))/a_{min} - (v_{min} - (v_f + a_{min}t_1))/a_{min}\right] \\
& + d_r \geq d_{safe}
\end{aligned}
$$

D. Predicates From Controller Data

D.1. Predicate Without Prettifying

Before applying the methods described in Section 6.3 and Section 6.4, a predicate for the `cruise` example looks like this (rounded to 6 decimal places):

$$\begin{aligned}
&-1.004058e_{choose}d_r + 0.000121d_r^2 + 4.011296e_{choose}v_e - 0.002316d_rv_e + 0.51353v_e^2 \\
&+ 8.5 \cdot 10^{-5}e_{choose}a_e - 0.000276d_ra_e + 0.002239v_ea_e - 6.4 \cdot 10^{-5}a_e^2 \\
&- 3.007296e_{choose}v_f + 0.001317d_rv_f - 0.012358v_ev_f - 0.001334a_ev_f \\
&- 0.499783v_f^2 - 0.000224e_{choose}a_f + 0.000261d_ra_f - 0.002111v_ea_f \\
&- 6.4 \cdot 10^{-5}a_ea_f + 0.001224v_fa_f + 3.1 \cdot 10^{-5}a_f^2 - 1.004058d_r \\
&+ 4.011296v_e + 8.5 \cdot 10^{-5}a_e - 3.007296v_f - 0.000224a_f \\
&+ 23.107387 \leq 0
\end{aligned}$$

D.2. Predicate Without Rounding

After leaving out unimportant features as described in Section 6.3 but without rounding coefficients as described in Section 6.4, a predicate for the `cruise` example looks like this (rounded to 6 decimal places):

$$\begin{aligned}
&-0.000463d_r^2 + 0.008656d_rv_e - 0.549255v_e^2 - 0.005078d_rv_f \\
&+ 0.046916v_ev_f + 0.496888v_f^2 + 2.043519d_r - 10.25286v_e \\
&+ 6.138132v_f - 39.685041 \leq 0
\end{aligned}$$

D.3. Advanced Numerical Precision Problems

Sometimes, when training the SVM, very large coefficients occur. As long as every datapoint is located on the right side of the hyperplane, the loss function (see Section A.3) does not penalize these large coefficients. However, when evaluating a predicate like

$$2x_1 - 3x_2 + 10^{18}x_3 - 10^{18} \leq 0$$

the floating point precision reaches its limits. So, in the rare case that we observe such a behavior, we apply the following countermeasure.

For every feature i, we add the *control samples* $(\vec{e}_i, 1)$ and $(\vec{e}_i, -1)$ to the dataset, where $\vec{e}_i$ is the unit vector in direction of feature i. Then we re-train the SVM. Note that this way, the data is not linearly separable. The loss function (see Section A.3) penalizes control samples that are far away from the separating hyperplane because either the positive or the negative control sample is located on the wrong side. Hence, the SVM uses small coefficients to keep the control samples reasonably close to the decision function. The `liblinear` tool [Fan+08] that we use in this work also supports different sample weights. Thus, we give the control samples a weight that is three orders of magnitude smaller than the weights for the regular samples so we do not disturb the regular training too much.

If, for some reason, we still have coefficients with an absolute value larger than 10^7 or smaller than 10^{-7} but not 0, we change them to a value inside of this interval. This might change the classification but therefore ensures that we do not run into precision errors after exporting the decision tree and evaluating the predicate on a device with a slightly different floating-point engine.

E. Results

Here we list the benchmark results for all case studies. The structure is the same as described in Subsection 7.2.3 with two slight differences. First, we add the size of the binary decision diagram (BDD) representation. Second, we explicitly list the tree sizes we get with the linear support vector machine, the logistic regression, and the OC1 heuristic splitting strategies. In Table 7.2 in the main matter, we only show the minimum across those.

The BDD sizes for the cyber-physical system cases is the minimum number of nodes from 10 tries. For the quantitative verification case studies, we show the BDD sizes from [Ash+21] which correspond to the minimum across 20 tries.

The benchmarks are split into three tables. Table E.1 contains the cyber-physical system case studies, Table E.2 and Table E.3 contain case studies from the quantitative verification benchmark set [Har+19].

Table E.1.: Benchmark results for the cyber-physical system case studies.

	Comparision				Linear			Quadratic	
Case Study	States	BDD	MinSize	Ax.Al.	LinSVM	LogReg	OC1	Poly	PolyPrio1
cartpole [Jag+20]	271	312	**169**	253	247	199	183	243	189
				263	263	187	261	**169**	**169**
10rooms [JZ17]	26,244	168	**49**	17,297	157	147	4,515	61	61
				17,297	121	107	7,455	**49**	**49**
helicopter [Jag+20]	280,539	**1,348**	475	6,339	5,787	3,769	TO	5,035	3,787
				9,649	9,763	4,637	TO	TO	TO
cruise_250 [LMT15a]	320,523	1,820	9	869	721	557	369	353	37
				1,067	817	657	363	**11**	25
cruise_300 [LMT15a]	500,920	2,229	9	1,157	991	691	467	521	59
				1,343	1,035	881	507	**13**	23
dcdc [RZ16]	593,089	575	5	271	279	139	179	**129**	199
				265	265	173	179	147	273
truck_trailer [KZ19]	1,386,211	**36,169**	1,839	338,283	TO	TO	TO	TO	TO
				366,411	TO	TO	TO	TO	TO
aircraft [RWR15]	2,135,056	**177,332**	31	915,877	916,685	TO	TO	725,011	602,335
				1,015,903	1,013,949	TO	TO	688,577	630,631
traffic_30m [SZ19]	16,639,662	TO	23	12,573	9,631	8,953	TO	TO	TO
				20,895	9,211	**7,099**	TO	TO	TO

Table E.2.: Benchmark results for case studies from the quantitative verification benchmark set (part 1).

	Comparision			Single Feature		Linear			Quadratic	
Case Study	States	BDD	MinSize	Ax.Al.	Categ.	LinSVM	LogReg	OC1	Poly	PolyPrio1
triangle-tireworld.9	48	51	**17**	27	28	25	23	19	25	**17**
				31	31	21	25	23	**17**	**17**
pacman.5	232	330	**37**	53	43	51	49	55	47	**37**
				81	81	71	59	81	**37**	**37**
rectangle-tireworld.11	241	495	481	481	**373**[1]	481	481	481	481	481
				481	481	481	481	481	481	481
philosophers-mdp.3	344	295	59	391	181	381	377	333	315	251
				403	307	375	367	393	251	**223**
firewire_abst.3.rounds	610	295	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
				25	**25**	**25**	**25**	**25**	**25**	**25**
rabin.3	704	303	**23**	111	187	51	43	29	69	27
				175	137	31	29	45	**23**	**23**
ij.10	1,013	436	19	1,291	1,291	907	753	771	897	209
				1,405	1,405	893	735	1,131	**141**	177
zeroconf.1000.4.true.correct_max	1,068	535	**45**	83	83	67	63	49	75	57
				79	79	47	**45**	71	**45**	**45**
blocksworld.5	1,124	3,985	367	1,687	1,308	1,515	1,407	1,583	1,451	891
				1,771	1,649	1,535	1,405	1,719	521	**513**

[1] this is better than the minimum size as it uses non-binary splits

Table E.3.: Benchmark results for case studies from the quantitative verification benchmark set (part 2).

	Comparision			Single Feature		Linear			Quadratic	
Case Study	States	BDD	MinSize	Ax.Al.	Categ.	LinSVM	LogReg	OC1	Poly	PolyPrio1
cdrive.10	1,921	5,134	1,903	2,401	3,122	2,401	2,401	**2,257**	2,401	2,401
				2,089	2,037	TO	TO	TO	TO	TO
consensus.2.disagree	2,064	138	25	67	67	75	69	69	57	51
				105	105	105	93	95	35	**33**
beb.3-4.LineSeized	4,275	913	**57**	65	70	65	65	63	65	59
				85	76	**57**	**57**	89	**57**	**57**
csma.2-4.some_before	7,472	1,059	**65**	103	103	107	105	89	103	79
				185	185	85	**65**	177	**65**	**65**
eajs.2.100.5.ExpUtil	12,627	1,315	65	167	160	173	161	135	133	141
				167	167	167	157	133	133	**125**
elevators.a-11-9	14,742	6,750	129	16,341	16,413	11,243	9,865	13,619	9,779	2,859
				17,809	17,495	11,505	9,955	16,423	2,023	**1,919**
exploding-blocksworld.5	76,741	3,447	149	16,913	8,138	4,503	2,687	5,993	4,511	**829**
				20,273	8,571	5,307	2,845	6,893	TO	TO
echoring.MaxOffline1	104,892	43,165	801	2,101	2,251	1,629	1,625	1,627	2,005	**1,431**
				5,031	TO	TO	TO	TO	TO	TO
wlan_dl.0.80.deadline	189,641	1,541	175	3,369	3,369	2,821	2,563	701	693	667
				3,675	3,675	2,841	2,621	1,049	**523**	TO
pnueli-zuck.5	303,427	**50,128**	173	171,371	171,371	156,165	150,341	125,421	114,979	83,219
				263,955	263,955	221,645	214,801	221,645	95,879	83,951