Low-Rank Approximation For Synthesizing Explainable Controllers

École Polytechnique
CSE303
Instructor: Gleb Pogudin

Alan Picucci
Giacomo Bruno

Due Date: December 12, 2022

1 Introduction

Cyber-physical systems are physical systems controlled by digital devices (like self-driving cars). Machine learning methods are a natural tool for designing controllers for such systems. However, the produced controllers are often not understandable or analyzable by a human, and this often raises safety concerns. A recent paper just published titled *Algebraically Explainable Controllers: Decision Trees and Support Vector Machines Join Forces* [2] tries to combine Decision Trees and SVM to create a new type of controller.

In this project, we will attempt to build upon this paper in order to improve the explainability of their controllers.

1.1 The paper we will build upon

As suggested by the title, the authors of the paper explore two approaches to represent controllers as decision trees. The objective of these approaches is to obtain small decision trees with explainable predicates. The first approach, which aimed to generate algebraic predicates (closed-form expressions) using domain knowledge, was found to be infeasible in practice, and we will not be touching upon this method. On the other hand, the second approach was found to be more successful: using Support Vector Machines to generate the polynomials at each split. To moderate the complexity of the predicates, the degree of the polynomials is restricted to order two. Thus, linear SVM is applied in quadratic space to obtain the best splits.

The results of the latter approach were very promising, as the authors were able to produce very small trees. However, the explainability of the predicates left a lot to be desired, despite the authors' attempts of using various simplifying techniques, such as filtering out irrelevant variables or rounding coefficients.

This paper will thus attempt a new approach to improving the explainability of these predicates: by performing k-rank approximation of the 2nd-degree polynomials.

1.2 k-rank approximation of a quadratic polynomial

In our project we want to simplify the complicated predicates generated by the SVM using low-rank approximation and exploiting the Principal Axis Theorem. The former is a famous approach based on SVD decomposition and it gives us a more concise way of writing a 2nd-degree polynomial. A quadratic form P can be written with its matrix representation:

$$P(\vec{x}) = \sum_{i=0}^{n} \sum_{j=0}^{i} c_{ij} x_i x_j = x^T A x, \text{where } \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{pmatrix} \tag{1}$$

where A is an $(n+1) \times (n+1)$ symmetric matrix of coefficients and obtained by setting $a_{ij} = \frac{1}{2} c_{ij}$ and $a_{ji} = \frac{1}{2} c_{ij}$ if $i \neq j$ and $a_{ii} = c_{ii}$ otherwise. Since A is a real and symmetric matrix, by the spectral theorem $A = FDF^T$, where $F = (f_0, \cdots, f_n)$ is an orthonormal matrix of normalized eigenvectors and D is a diagonal matrix of eigenvalues [1]. This turns out to be very useful since, by the Principal Axis theorem [3], it allows us to represent our Quadratic form P as a sum of squares:

$$P = (F^T x)^T D (F^T x) = \sum_{i=0} \lambda_i (f_i \cdot x)^2 \tag{2}$$

where λ_i is the i-th eigenvalue of the matrix D, f_i is the i-th column vector from the matrix F and x is our vector of variables. This significantly simplifies our Polynomial and allows us to easily remove elements from the sum by using k-rank approximation:

$$P = (\underbrace{\mathbf{F^T}}_{k \times n+1} \underbrace{\mathbf{x}}_{n+1 \times 1})^T \times \underbrace{\mathbf{D}}_{k \times k} \times (\underbrace{\mathbf{F}^{\mathrm{T}}}_{k \times n+1} \underbrace{\mathbf{x}}_{n+1 \times 1}) \tag{3}$$

We are now free to optimize the value of k for every predicate. We will create a list of feasible k's according to some criteria and then choose the k which minimizes the impurity of the split. The function that implements this will be analyzed in the next part of the paper.

1.3 Why we use low-rank approximation

The goal of this project is to obtain more explainable controllers by lowering the rank of our polynomials, but why should this work? The algorithm chooses among different second-degree polynomials and selects the one with the lowest value of impurity. Therefore, our low-rank approximation might be expected to increase the impurity by "removing information". We hope that despite this valid reasoning our k-rank approach will actually regularize these polynomials, avoiding potential overfitting issues that might have derived from the SVM's during training. Furthermore, thanks to the theorem mentioned above we can represent these polynomials in sums of squares which should be easier to read and interpret for a human reader.

2 Into the code

2.1 The original code

The original code builds decision trees by giving them a list of two splitting strategies, one of which is always the axis-aligned, and at each node choosing the split with the best impurity. The axis-aligned is always included since it is the easiest to interpret (it's an inequality in only one variable). Moreover, the author also introduces a "priority" variable for polynomial splitting strategies that takes values between 0 and 1: this is used to compare impurities in a weighted manner so that the impurity of the polynomial split will be evaluated as "real impurity"*"priority". For small values of priority, this ensures that the polynomial split has to be significantly better than the axis-aligned split so that it may be chosen, since it is harder to interpret. In order to implement our low-rank approach, we created a new splitting strategy which we could then feed to the original algorithm, comparing it to the axis-aligned split at each node.

2.2 Implementing our functions

As previously mentioned, we need a function that will allow us to optimize over the parameter k for every generated predicate. We thus break this process down into the following main operations: firstly, having a function which gives us a list of all the feasible k according to the following criteria: the k-th eigenvalue (in decreasing absolute order) must be small enough (smaller than a threshold value m, which is 0.1 by default.) and the ratio between the successive eigenvalues must also be smaller than some threshold z (by default 0.1). The idea is that such values of k will cut irrelevant eigenvalues out, meaning that we won't lose much from this removal. This function is implemented as follows:

```
  def relative_largest_gaps(self,
 m=0.1,z = 0.1):
       # This function now returns a list with all the viable
          values of k.
       L=self.eigenvalues #sorted eigenvalues largest to smallest
       if 0 in L:
           indices = [i for i in range(1, L.index(0)+1) if
              abs(L[i]) < m and abs(L[i] / L[i - 1]) < z]
       else:
           indices = [i for i in range(1, len(L)) if abs(L[i]) < m
              and abs(L[i] / L[i - 1]) < z] + [len(L)]
       return indices
```

Note that we identify the cases in which there is a 0 eigenvalue and treat them differently to avoid dividing by 0. Also, it's clear that if we have null eigenvalues, cutting them off should have no effect. Moreover, we remark that in the above implementation, the maximum k (i.e. no approximation) is always included among the indices. We will eventually modify this function so that we never include the maximum k, except for when no other value of k satisfies the above conditions. By doing so, we can essentially force the algorithm to make approximations, since it will not have the original splits to choose from.

Next, we need to try out each of these potential splits and let the algorithm select the best one based on their impurity. We therefore loop over the possible k's, calculating the impurity of the split to subsequently choose the best one.

```
  for k in indices:
   split=deepcopy(original_split)
   impurity = impurity_measure.calculate_impurity(dataset, split)
   splitData = (impurity, split, label_mask, standardizer, svc)
   splits.append(splitData) #append the possible split
```

Our last contribution to the code is the k rank approx function which for every possible polynomial split computes its low rank approximation for a given k.

```
   def k_rank_approx(self, k, round_eigvals=False, eps=0.0001):
       if round_eigvals: #Possibility to round down small entries
          of the matrix F seiplifying the terms inside the squares
```

```
        self.F=self.F[:,:k]
        self.F[np.abs(self.F) < eps] = 0
    else:
        self.F=self.F[:,:k]
    self.eigenvalues=self.eigenvalues[:k]
    D=np.diag(self.eigenvalues)
    A=(self.F)@D@((self.F).T) #Here we compute the low rank
        approximation.
```

We made several small adjustments to these three functions to reach our desired outputs. For instance, we changed the upper bound for the value of the ratios by experimenting with m and z. Additionally, in the k rank function we included the possibility off adding an epsilon according to which every value of the matrix F below this is assigned zero: this trick simplifies our sums of squares polynomial. Finally, we also added a method that counts how many times each k-approximation was used. All of these small adjustments were crucial for debugging and to analyse how the algorithm computed and chose splits.

2.3 Problems faced

In this section we would like to comment on some of the technical issues we faced along the way. In fact, we had to deal with many problems, mainly due to the complexity and volume of the code that we had to modify. This made implementing new functions quite tricky, since we risked compromising the basic functionality of the program with even slight changes. For example, this occurred when we implemented the function which allowed us to count how many times each k was used for an approximation in the chosen splits. Moreover, we also encountered many bugs while trying to implement the loop over the possible values of k. These were caused by pre-existing functions written by the original author which were not compatible with our new modifications and resulted in some splits not being stored correctly and therefore not being selected as the best split. It's also worth mentioning that the calculated impurities sometimes returned 2^{63}-1, the largest number supported in 64-bit python. This implies that there is probably some part of C code used to calculate the author's new impurity implementation which may cause errors when trying to modify the original code.

3 Results

Now we will present and comment on the results that emerged from our investigation. Note that these results primarily refer to the cruise_250 and cruise_300 databases, which we have used as reference points to compare to the original author's results. The last part of this section will comment on the results found for other data sets. Note that these results can be reproduced through the repository [4] linked in the bibliography.

3.1 Not enforcing low-rank approximation

Before trying to force the algorithm to perform low-rank approximation on the polynomial splits, we gave it the choice of choosing between the approximated polynomials and the original one (as we mentioned earlier in the paper). The goal was to see whether the approximations would ever be preferred to the original polynomials. However, we found that our algorithm generated trees of exactly the same size as the original paper, with virtually every polynomial split having max k. However, there were 4 polynomial splits which were chosen to have k=3, which probably means that both the original splits and the approximated splits with k=3 had a perfect impurity of 0, making the algorithm indifferent. We expected a different result, hoping to verify our initial hypothesis regarding regularization however the algorithm always chose the highest possible rank implying that our hypothesis was probably incorrect. The results are presented in Appendix A.

3.2 Enforcing low-rank approximation

Since our first approach virtually never yielded low-rank splits, we tried to force the algorithm to choose among low-rank polynomials. We implemented this by modifying the relative_largest_gaps function as mentioned earlier: we forced the algorithm to select k's which were strictly smaller than the maximum k whenever such k's were feasible under the criteria imposed. The results can be found in Appendix B and are again inconclusive since most of the trees we obtained have hundreds of additional nodes with respect to the default one. Strangely though in the case of cruise_250 we obtain the same number of nodes, 13, as its former counterpart. We tested several different versions of the splitting strategies, slightly altering the parameters each time. The version available in the Appendix was obtained after several attempts. Unfortunately changing these values didn't really improve the results since the trees often increased in size without improving in explainability. In these cases, the axis-aligned splits were often preferred to the polynomial splits, so the low rank approximation must have significantly increased their impurity.

3.3 Enforcing low-rank approximation on other datasets

In order to further verify our conclusions we tried running our algorithm on different datasets, specifically "dcdc" and "cartpole". The others, such as "airplane" and "helicopter", were too large for us to compute. The two new datasets use a polynomial with max rank of k=3 and the results again were similar to the previous one since we obtained trees which were quite bigger than the original ones. Of course, the original results are available in the original author's paper, while ours can be found in Appendix C.

4 Conclusion

In the end our approach didn't quite work as desired: we expected that these low-rank polynomials would have been more effective at increasing the explainability of the nodes. Instead, we got larger trees and didn't significantly improve their explainability through the sum of squares representation, as exemplified by the sample tree in Appendix B. We note that we used this approach only in cases where max(k)<5 and it may be that in such cases it's not profitable to approximate. In larger k, however, this approach may be more useful and more research should be made into datasets of this type. Still one should attempt to further optimise the current algorithm since it's computationally expensive: running it was quite an extensive process. Another possible reason for the inconclusiveness of the project is that we exploit the greedy approach when choosing the splits and, as known, this process doesn't always finish with the optimal result.

Appendix

A Results on cruise without forcing low-rank approximation

	poly-lowrank	poly-lowrank-minEntropy	poly-lowrankPrio1	poly-lowrankPrio1-minEntropy
cruise_250 #(s,a): 961569 #doc: 320523	nodes: 347 inner nodes: 173 paths: 174 bandwidth: 8 k=1: 0 k=2: 0 k=3: 4 k=4: 68 time: 00:01:01.164 DOT / C	nodes: 11 inner nodes: 5 paths: 6 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:34.015 DOT / C	nodes: 37 inner nodes: 18 paths: 19 bandwidth: 5 k=1: 0 k=2: 0 k=3: 0 k=4: 12 time: 00:00:52.787 DOT / C	nodes: 21 inner nodes: 10 paths: 11 bandwidth: 4 k=1: 0 k=2: 0 k=3: 0 k=4: 6 time: 00:00:41.522 DOT / C
cruise_300 #(s,a): 1502760 #doc: 500920	nodes: 509 inner nodes: 254 paths: 255 bandwidth: 8 k=1: 0 k=2: 0 k=3: 7 k=4: 104 time: 00:01:27.056 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:53.075 DOT / C	nodes: 35 inner nodes: 17 paths: 18 bandwidth: 5 k=1: 0 k=2: 0 k=3: 0 k=4: 12 time: 00:01:16.375 DOT / C	nodes: 19 inner nodes: 9 paths: 10 bandwidth: 4 k=1: 0 k=2: 0 k=3: 0 k=4: 5 time: 00:01:04.883 DOT / C

Figure 1: The tree statistics for different implementations of our splitting strategy on both cruise datasets: the first two use, respectively, entropy and minEntropy (an impurity measure created by the original author) with the "priority" variable set to 0.1. The latter two are equivalent except they have "priority" set to 1.

B Results on cruise enforcing low-rank approximation

```
poly_lowrank_default= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.1, z=0.1, round_eigvals=False) #(Alan+Giacomo): Here w
poly_lowrank_default.priority=0.1 #(Alan+Giacomo): Here we can choose the "priority" variable for our splitting strategy. This variable is us
poly_lowrank_default_rounding= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.1, z=0.1, round_eigvals=True, eps=0.01)
poly_lowrank_default_rounding.priority=0.1
poly_lowrank_permissive= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.5, z=0.2, round_eigvals=False, eps=0.01)
poly_lowrank_permissive.priority=0.1
poly_lowrank_permissive_round= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.5, z=0.2, round_eigvals=True, eps=0.01)
poly_lowrank_permissive_round.priority=0.1
polyPrio1_lowrank_default= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.1, z=0.1, round_eigvals=False)
polyPrio1_lowrank_default.priority=1
polyPrio1_lowrank_default_rounding= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.1, z=0.1, round_eigvals=True, eps=0.01)
polyPrio1_lowrank_default_rounding.priority=1
polyPrio1_lowrank_permissive= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.5, z=0.2, round_eigvals=False, eps=0.01)
polyPrio1_lowrank_permissive.priority=1
polyPrio1_lowrank_permissive_round= LowRankPolynomialClassifierSplittingStrategy(prettify=False, m=0.1, z=0.1, round_eigvals=True, eps=0.01)
polyPrio1_lowrank_permissive_round.priority=1
```

Figure 2: Defining the various splitting strategies that we have tested.

	poly-lowrank_default	poly-lowrank_default-minEntropy	poly-lowrank_default_rounding	poly-lowrank_default_rounding-minEntropy	poly-lowrank_permissive	poly-lowrank_permissive-minEntropy
cruise_250 #(s,a): 961569 #doc: 320523	nodes: 639 inner nodes: 319 paths: 320 bandwidth: 9 k=1: 0 k=2: 0 k=3: 7 k=4: 40 time: 00:01:08.538 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:34.756 DOT / C	nodes: 823 inner nodes: 411 paths: 412 bandwidth: 9 k=1: 0 k=2: 0 k=3: 6 k=4: 5 time: 00:01:01.381 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:31.848 DOT / C	nodes: 825 inner nodes: 412 paths: 413 bandwidth: 9 k=1: 0 k=2: 0 k=3: 9 k=4: 3 time: 00:01:03.613 DOT / C	nodes: 575 inner nodes: 287 paths: 288 bandwidth: 9 k=1: 0 k=2: 2 k=3: 11 k=4: 6 time: 00:01:03.308 DOT / C
cruise_300 #(s,a): 1502760 #doc: 500920	nodes: 887 inner nodes: 443 paths: 444 bandwidth: 9 k=1: 0 k=2: 1 k=3: 15 k=4: 43 time: 00:01:26.306 DOT / C	nodes: 305 inner nodes: 152 paths: 153 bandwidth: 8 k=1: 0 k=2: 0 k=3: 5 k=4: 19 time: 00:01:39.643 DOT / C	nodes: 1055 inner nodes: 527 paths: 528 bandwidth: 10 k=1: 0 k=2: 2 k=3: 9 k=4: 10 time: 00:01:22.363 DOT / C	nodes: 351 inner nodes: 175 paths: 176 bandwidth: 8 k=1: 0 k=2: 0 k=3: 3 k=4: 9 time: 00:01:37.847 DOT / C	nodes: 1063 inner nodes: 531 paths: 532 bandwidth: 10 k=1: 0 k=2: 2 k=3: 18 k=4: 3 time: 00:01:29.741 DOT / C	nodes: 1025 inner nodes: 512 paths: 513 bandwidth: 10 k=1: 0 k=2: 1 k=3: 31 k=4: 4 time: 00:01:51.486 DOT / C

poly-lowrank_permissive_round	poly-lowrank_permissive_round-minEntropy	polyPrio1-lowrank_default	polyPrio1-lowrank_default-minEntropy	polyPrio1-lowrank_default_rounding	polyPrio1-lowrank_default_rounding-minEntropy
nodes: 835 inner nodes: 417 paths: 418 bandwidth: 9 k=1: 0 k=2: 0 k=3: 6 k=4: 2 time: 00:00:56.770 DOT / C	nodes: 635 inner nodes: 317 paths: 318 bandwidth: 9 k=1: 0 k=2: 1 k=3: 10 k=4: 4 time: 00:01:04.884 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:30.657 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:32.305 DOT / C	nodes: 77 inner nodes: 38 paths: 39 bandwidth: 6 k=1: 0 k=2: 1 k=3: 2 k=4: 4 time: 00:00:36.291 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:32.099 DOT / C
nodes: 1113 inner nodes: 556 paths: 557 bandwidth: 10 k=1: 0 k=2: 2 k=3: 12 k=4: 0 time: 00:01:28.420 DOT / C	nodes: 1041 inner nodes: 520 paths: 521 bandwidth: 10 k=1: 0 k=2: 1 k=3: 23 k=4: 4 time: 00:01:48.095 DOT / C	nodes: 507 inner nodes: 253 paths: 254 bandwidth: 8 k=1: 0 k=2: 6 k=3: 35 k=4: 30 time: 00:01:24.152 DOT / C	nodes: 567 inner nodes: 283 paths: 284 bandwidth: 9 k=1: 0 k=2: 9 k=3: 34 k=4: 31 time: 00:01:38.438 DOT / C	nodes: 653 inner nodes: 326 paths: 327 bandwidth: 9 k=1: 0 k=2: 3 k=3: 30 k=4: 15 time: 00:01:23.997 DOT / C	nodes: 599 inner nodes: 299 paths: 300 bandwidth: 9 k=1: 0 k=2: 6 k=3: 16 k=4: 24 time: 00:01:39.409 DOT / C

polyPrio1-lowrank_permissive	polyPrio1-lowrank_permissive-minEntropy	polyPrio1-lowrank_permissive_round	polyPrio1-lowrank_permissive_round-minEntropy
nodes: 591 inner nodes: 295 paths: 296 bandwidth: 9 k=1: 0 k=2: 11 k=3: 42 k=4: 6 time: 00:00:57.940 DOT / C	nodes: 697 inner nodes: 348 paths: 349 bandwidth: 9 k=1: 0 k=2: 12 k=3: 39 k=4: 6 time: 00:01:19.894 DOT / C	nodes: 77 inner nodes: 38 paths: 39 bandwidth: 6 k=1: 0 k=2: 1 k=3: 2 k=4: 4 time: 00:00:40.397 DOT / C	nodes: 13 inner nodes: 6 paths: 7 bandwidth: 3 k=1: 0 k=2: 0 k=3: 0 k=4: 3 time: 00:00:30.737 DOT / C
nodes: 1063 inner nodes: 531 paths: 532 bandwidth: 10 k=1: 0 k=2: 25 k=3: 69 k=4: 6 time: 00:01:49.806 DOT / C	nodes: 1159 inner nodes: 579 paths: 580 bandwidth: 10 k=1: 0 k=2: 23 k=3: 76 k=4: 6 time: 00:03:04.357 DOT / C	nodes: 653 inner nodes: 326 paths: 327 bandwidth: 9 k=1: 0 k=2: 3 k=3: 30 k=4: 15 time: 00:01:24.958 DOT / C	nodes: 599 inner nodes: 299 paths: 300 bandwidth: 9 k=1: 0 k=2: 6 k=3: 16 k=4: 24 time: 00:01:35.487 DOT / C

Figure 3: Results from running the various splitting strategies on both cruise datasets

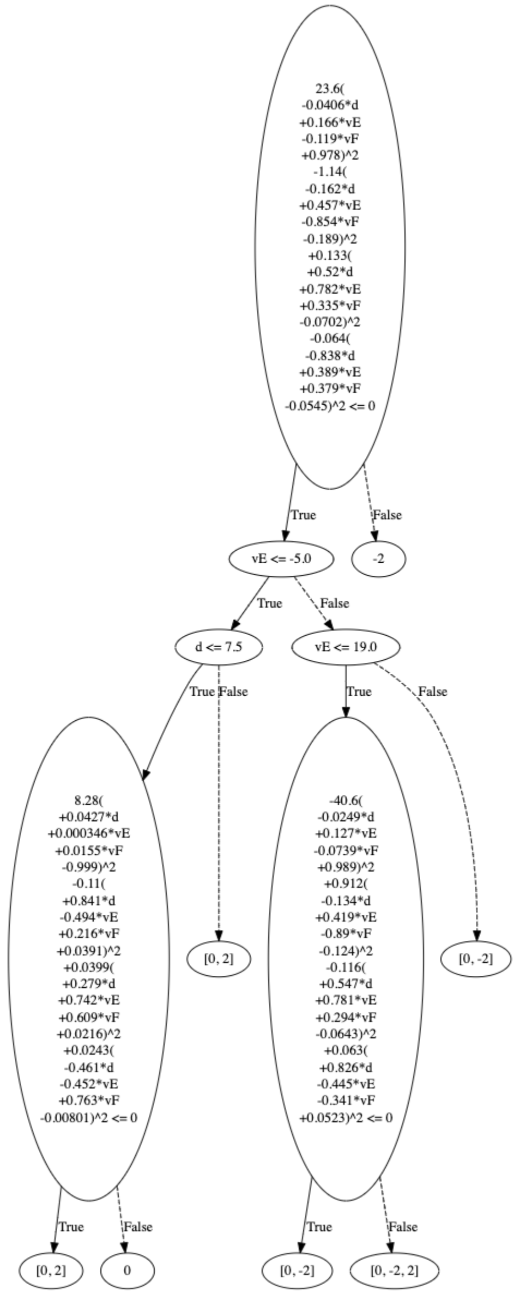

Figure 4: Sample generated tree to showcase the polynomials represented as a sum of squares

C Results on other datasets enforcing low-rank approximation

	poly-lowrank_default	poly-lowrank_default-minEntropy	poly-lowrank_default_rounding	poly-lowrank_default_rounding-minEntropy	poly-lowrank_permissive	poly-lowrank_permissive-minEntropy
dcdc # (s,a): 1186178 #doc: 593089	nodes: 151 inner nodes: 75 paths: 76 bandwidth: 7 k=1: 0 k=2: 1 k=3: 26 k=4: 0 time: 00:03:08.175 DOT / C	nodes: 145 inner nodes: 72 paths: 73 bandwidth: 7 k=1: 0 k=2: 1 k=3: 22 k=4: 0 time: 00:03:14.737 DOT / C	nodes: 229 inner nodes: 114 paths: 115 bandwidth: 7 k=1: 0 k=2: 1 k=3: 10 k=4: 0 time: 00:02:12.384 DOT / C	nodes: 207 inner nodes: 103 paths: 104 bandwidth: 7 k=1: 0 k=2: 1 k=3: 12 k=4: 0 time: 00:03:05.456 DOT / C	nodes: 151 inner nodes: 75 paths: 76 bandwidth: 7 k=1: 0 k=2: 1 k=3: 26 k=4: 0 time: 00:02:07.153 DOT / C	nodes: 145 inner nodes: 72 paths: 73 bandwidth: 7 k=1: 0 k=2: 1 k=3: 22 k=4: 0 time: 00:03:05.091 DOT / C

poly-lowrank_permissive_round-minEntropy	polyPrio1-lowrank_default	polyPrio1-lowrank_default-minEntropy	polyPrio1-lowrank_default_rounding	polyPrio1-lowrank_default_rounding-minEntropy	polyPrio1-lowrank_permissive
nodes: 207 inner nodes: 103 paths: 104 bandwidth: 7 k=1: 0 k=2: 1 k=3: 12 k=4: 0 time: 00:03:01.988 DOT / C	nodes: 173 inner nodes: 86 paths: 87 bandwidth: 7 k=1: 0 k=2: 5 k=3: 33 k=4: 0 time: 00:02:26.912 DOT / C	nodes: 185 inner nodes: 92 paths: 93 bandwidth: 7 k=1: 0 k=2: 5 k=3: 26 k=4: 0 time: 00:04:10.903 DOT / C	nodes: 265 inner nodes: 132 paths: 133 bandwidth: 8 k=1: 0 k=2: 5 k=3: 18 k=4: 0 time: 00:02:16.511 DOT / C	nodes: 229 inner nodes: 114 paths: 115 bandwidth: 7 k=1: 0 k=2: 5 k=3: 19 k=4: 0 time: 00:03:16.615 DOT / C	nodes: 173 inner nodes: 86 paths: 87 bandwidth: 7 k=1: 0 k=2: 5 k=3: 33 k=4: 0 time: 00:02:31.985 DOT / C

polyPrio1-lowrank_permissive-minEntropy	polyPrio1-lowrank_permissive_round	polyPrio1-lowrank_permissive_round-minEntropy
nodes: 153 inner nodes: 76 paths: 77 bandwidth: 7 k=1: 0 k=2: 7 k=3: 26 k=4: 0 time: 00:02:51.257 DOT / C	nodes: 265 inner nodes: 132 paths: 133 bandwidth: 8 k=1: 0 k=2: 5 k=3: 18 k=4: 0 time: 00:02:13.633 DOT / C	nodes: 229 inner nodes: 114 paths: 115 bandwidth: 7 k=1: 0 k=2: 5 k=3: 19 k=4: 0 time: 00:03:08.184 DOT / C

Figure 5: Results from running the various splitting strategies on dcdc dataset

	poly-lowrank_default	poly-lowrank_default-minEntropy	poly-lowrank_default_rounding	poly-lowrank_default_rounding-minEntropy	poly-lowrank_permissive	poly-lowrank_permissive-minEntropy
cartpole `#(s,a)`: 21951 `#doc`: 271	nodes: 243 inner nodes: 121 paths: 122 bandwidth: 7 k=1: 0 k=2: 0 k=3: 5 k=4: 0 time: 00:00:03.653 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 0 k=3: 60 k=4: 0 time: 00:00:20.232 DOT / C	nodes: 243 inner nodes: 121 paths: 122 bandwidth: 7 k=1: 0 k=2: 0 k=3: 5 k=4: 0 time: 00:00:03.013 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 1 k=3: 62 k=4: 0 time: 00:00:14.622 DOT / C	nodes: 243 inner nodes: 121 paths: 122 bandwidth: 7 k=1: 0 k=2: 0 k=3: 5 k=4: 0 time: 00:00:03.158 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 0 k=3: 80 k=4: 0 time: 00:00:16.623 DOT / C

poly-lowrank_permissive_round	poly-lowrank_permissive_round-minEntropy	polyPrio1-lowrank_default	polyPrio1-lowrank_default-minEntropy	polyPrio1-lowrank_default_rounding	polyPrio1-lowrank_default_rounding-minEntropy
nodes: 243 inner nodes: 121 paths: 122 bandwidth: 7 k=1: 0 k=2: 0 k=3: 5 k=4: 0 time: 00:00:03.236 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 0 k=3: 80 k=4: 0 time: 00:00:16.615 DOT / C	nodes: 189 inner nodes: 94 paths: 95 bandwidth: 7 k=1: 0 k=2: 4 k=3: 10 k=4: 0 time: 00:00:03.710 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 0 k=3: 60 k=4: 0 time: 00:00:14.968 DOT / C	nodes: 189 inner nodes: 94 paths: 95 bandwidth: 7 k=1: 0 k=2: 4 k=3: 10 k=4: 0 time: 00:00:03.739 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 1 k=3: 62 k=4: 0 time: 00:00:15.639 DOT / C

polyPrio1-lowrank_permissive	polyPrio1-lowrank_permissive-minEntropy	polyPrio1-lowrank_permissive_round	polyPrio1-lowrank_permissive_round-minEntropy
nodes: 191 inner nodes: 95 paths: 96 bandwidth: 7 k=1: 0 k=2: 5 k=3: 11 k=4: 0 time: 00:00:04.121 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 0 k=3: 80 k=4: 0 time: 00:00:17.051 DOT / C	nodes: 189 inner nodes: 94 paths: 95 bandwidth: 7 k=1: 0 k=2: 4 k=3: 10 k=4: 0 time: 00:00:04.135 DOT / C	nodes: 169 inner nodes: 84 paths: 85 bandwidth: 7 k=1: 0 k=2: 1 k=3: 62 k=4: 0 time: 00:00:14.876 DOT / C

Figure 6: Results from running the various splitting strategies on cartpole

References

[1] Sheldon Axler. *Linear algebra done right*. 3rd ed. Springer, 2015.

[2] Jan Křetínský Florian Jüngermann and Maximilian Weininger. *Algebraically Explainable Controllers: Decision Trees and Support Vector Machines Join Forces*. URL: `https://arxiv.org/pdf/2208.12804.pdf`.

[3] Gilbert Strang. *Introduction to Linear Algebra*. 5th ed. 2016.

[4] *The Repository containing the code used for our project*. `https://github.com/alan-picucci/CSProject/tree/main/florianjuengermann-dtcontrol-thesis-files-1d747f7`.