Vrije Universiteit Amsterdam


Bachelor Thesis

# New clause selection function elements for the E theorem prover

**Author:** Niels Galjaard (2518100)

*supervisor:* Jasmin Blanchette
*2nd reader:* Petar Vukmirovic

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

July 29, 2019

# Contents

# 1 Introduction

## 1.1 Motivation

There is a large variety of research and implementation of automated theorem proving [1] with academic and commercial use, from proving mathematical theorems to proving the correctness of circuits. At times automated provers even find proofs for not yet proven theorems[9], hence there is clear benefit to improving the speed of these provers. Because the work of building provers has already been done, I have decided not to reinvent the wheel and simply go by an existing implementation. Moreover I have decided to use various machine learning algorithms to attempt to achieve this, since the ability to use the required computational resources is far more accessible today, when compared to the time at which these provers were build. Additionally related problems can be efficiently solved using machine-learning as shown by Stanford researchers[12]. Proof search heuristics are therefore a promising subject of research.

## 1.2 Problem statement

The main point of interest for this paper is the speed of provers. Since most provers are already proven to be refutationally complete, what remains is the speed at which these proofs can be found, as is usually the case in computer Science there are two metrics of speed for algorithms, the speed in seconds, and the speed in steps of the program. Because of the degree of inconsistency when measuring speed in seconds, I am for the purpose of this paper focused on the number of steps of the program. Another metric is the number of proofs, since there is a possibility of improving the number of proofs found, by not considering unproductive clauses.

## 1.3 Hypothesis

This paper concerns with the ability to improve the speed of a prover, we are interested in whether or not adapting several existing heuristics with machine learned heuristics increases the speed of a prover, where speed is the number of steps, in this case that is the number of clauses processed by the prover. And whether or not this change in speed results in more proofs found.

## 1.4 Contributions

Similar to almost all theorem provers[1] the E prover is implemented in C, the main bulk of the code of this paper including the code to train models is also in C, because of this a package of C code for training models is included, this ended up not part of the modified E code, however it is used to train several models. see models.c for specifics

**E extensions** For this paper several additions and modifications have been made to the Eprover, these are somewhat inelegant, given the issue with file I/O that is currently prohibiting some features on windows Linux subsystem. on many of the Machine learning models this means that the original model file has not been used and the weights learned have been hard coded into C functions. Moreover the bash scripts also contain some strange constructs as a result of file issues, however several manual checks applied afterwards do show the data and measurements to be correct.

One of the two main sections of this paper is the classification models for priority functions, these are partially trained using the training module of C included in the paper and partially trained on sklearn[**?**] or Keras[**?**] models. In both cases due to file i/o issues they cannot be made to work directly with fopen in C, therefore I have in both cases simply printed the weights and copy pasted the weights into a piece of C code and recompiled.

Despite the issues in training and somewhat strange implementation, the Neural networks and regression models are correct and can with little effort be included into the E prover for testing or further work.

**Training Code and scripts** Another section of the paper restricted due to the file i/o is the printing of clause features. The current version of bash some times discards fprintf calls, and often is incapable of piping to files whenever files exceed a 0.5GB limit. However the way the features are printed to std.out in the current implementation, they can be extracted using grep.

Piping these output files into a larger file I have used very basic pandas[**?**] to organize them into a larger CSV file, which is ultimately used by the training module of C, Keras and sklearn. Once these have been trained the last file is used to evaluate the results of the learned networks.

# 2 Background

## 2.1 First-order Logic

First-order logic can be seen as an extension of propositional logic, introducing predicates and quantifiers. Predicates are functions which map domain variables to truth values. This allows for predication of objects, which have no truth value, hence a formula like $P(x)$, is a well formed formula in first order logic. Quantifiers bind variables. Which describe the value of their respective variables, hence a formula like $\exists x, P(x)$ is a sentence in first order logic with a fixed truth value. As a result we can use all sentences as propositions in rules.

## 2.2 automated theorem proving

Depending on the definition of the Predicates and functions, several statements in mathematics can be proven using first-order logic[10]. Most of the time this is done through a proof of refutation, given a set of axioms $\{A_1, A_2, ...A_n\}$ and

a hypothesis $\{H\}$ assume $\{\neg H, A_1, A_2 ... A_n\}$ and try to show the unsatisfiability of this set. Since this set of statements is rewritten to CNF. If it's possible to derive $\emptyset$ from any of the clauses, this shows that the set of axioms is unsatisfiable and proves $\neg\neg H$

## 2.3 Superposition calculus

Any of the previous FOL literals from the CNF can be rewritten to equations, for example $P(x)$ becomes $P(x) \simeq \top$ and $\neg P(x)$ becomes $P(x) \not\simeq \top$. From these types equations it's possible to use paramodulation, for example given $C_1 \quad C_2$. it's possible to derive $C_3$ like this

$$\frac{C_1 : G[x] \simeq y \vee C_1^{'} \quad C_2 : f(a) \simeq x \vee C_2^{'}}{C_3 : G[f(a)] \simeq y \vee C_1^{'} \vee C_2^{'}}$$

The E prover uses a restricted form of paramodulation, which uses an ordering $\succ$, to constrain the possible from and into clauses. This would make the previous derivation subject to three extra constraints, with more constraints introduced for negative superposition.

$$\frac{C_1 : G[x] \simeq y \vee C_1^{'} \quad C_2 : f(a) \simeq x \vee C_2^{'}}{C_3 : G[f(a)] \simeq y \vee C_1^{'} \vee C_2^{'}}$$

where x $\succ f(a)$
$G[x] \succ y$
$f(a) \simeq x \succ L \in C_1^{'}$

Another important feature of the super position calculus is the use of rewrite rules, which alter or remove clauses from the current set. These rules[11] unlike the paramodulation consume their premise.

$$\frac{C}{\;\;\;} \qquad\qquad \frac{C_1^{'} \vee t \not\simeq t}{C_1^{'}}$$

The rule on the left simply removes clause C from the current set. This rule is used to remove tautologies, which are unproductive when trying to show that the sets are unsatisfiable. The other rules is used to removed contradictory literals, Since $C_1^{'} \vee \bot \equiv C_1^{'}$

## 2.4 The E prover

### 2.4.1 Heuristic types of E

E by default has three different types of interdependent heuristics, which are considered in the following order, priority functions, generic weight functions and literal selection strategies.

**Priority functions** Priority functions split the data set into two categories, those with high and normal priority. The prover considers the clauses for selection in this order, by adding them to priority queues. Higher priority is always considered first. Therefore prioritizing bad clauses severely impacts the prover's performance.

In the E implementation these priority functions are based on features that are based on clause properties, hence the E prover contains priority functions like "PreferGoal".

**generic weight functions** Generic weights functions assign a weight to clauses. In the original implementation of E these are based on symbols and their type in clauses. And are linear combinations of their features. These weights are considered from lowest to highest, they are used to sort the priority queues from the priority functions.

**Literal selection strategies** This heuristic is a function which marks literals for paramodulation, resulting from these markings the prover will perform generating inferences on those literals.

# 3 Machine learning

## 3.1 data generation

The following algorithms have been used to fit models to data, where the binary classifications are made using the clause features, with the class being if the clause was in the proof. For the use of generic weight functions, I have split the data into Goal and nonGoal as a result we train two models on these datasets, moreover we obviously need to innovate some response variable, I have chosen to use a hyper parameter $\zeta$ [1], which is a term that is added to the original clauseweight assigned by E, where the clause was not in the proof. The hyper parameter is subtracted if the clause was in the proof. This modified clause weight data will be used for regression fitting. This returns a model, which should assign a higher score to clauses not in the proof, thereby improving on the original weight.

Since the E prover already provides several methods for heuristics, which assign multiplicative scores to certain types of features, i.e. "Refinedweight" and "Literalweight" [2] it is possible to use these functions to extract features, by setting all but one multiplier to 0 and the remaining multiplier to 1. This extracts the feature for that one multiplier, thereby creating $2^{multipliers}$ features to use. In addition to these I have also included whether or not the clause is a goal clause as a feature. This means that the heuristics created from these features can only leverage that information, which is also available to the code that computes the default heuristic.

Creating the data is done though running the E prover on ALG tptp problems[13] for which it can rapidly find the proofs.[2] then piping out, the clause features

[1] in my case $\zeta = 5$ and $\zeta = 20$

[2] in our case ALG006-1.p ALG007-1.p ALG011-1.p ALG014+1.p ALG017+1.p ALG018+1.p ALG019+1.p ALG029+1.p ALG030+1.p ALG036+1.p ALG039+1.p ALG040+1.p ALG041+1.p ALG069+1.p ALG070+1.p ALG171+1.p ALG172+1.p ALG173+1.p ALG174+1.p ALG175+1.p ALG176+1.p ALG177+1.p ALG178+1.p

to be considered, the heuristic clause evaluation assigned, as well as the proof object, into an intermediate file. Consequently the remaining dataset is split and merged into two CSV files, the clauses in the proof and those not in the proof. These datasets are once again split into train and validation data, these two datasets have been used to fit the algorithms. The partitioning is done to prevent overfitting on the training problems.

## 3.2 Proposed algorithms

### 3.2.1 Support vector Machine

The Support vector machine is a model, which creates a maximum margin decision boundary between N classes, since we use only two classes those in and not in the proof we can use this model as a classifier, however a slight adaptation can change it to a ranking classifier. Therefore in addition to classification it is also an appropriate model to compute a generic clause weight function.

### 3.2.2 Linear Regression

There are several versions of linear regression, since this is a known reliable method we will include it's most standard variants here. We will use the least squares estimate by using the ordinary least squares regression, as well as the constrained version of non negative regression. So we are looking for a $y^{'} = \vec{w} \cdot \vec{x} + b$ has minimal $L$

$L = \frac{1}{N} \sum_{i=1}^{N} (y_i - \vec{w} \cdot \vec{x_i}))^2$ and $L = \frac{1}{N} \sum_{i=1}^{N} (y_i - \vec{w} \cdot \vec{x_i}))^2, where, \vec{w} \in \mathbb{R}_+^n$

### 3.2.3 Logistic Regression

A logistic regression is applicable to binary classification and there is no obvious extension to modeling clause-weights. We will use this prediction model only for determining clause-priorities. For the content of this paper we will use only the logistic regression with $b = e$ or , $\frac{1}{1+e^{\vec{w} \cdot \vec{x}}}$ this has little to no effect on performance, but simply allows us to take advantage of faster computation by the standard implementations of exp() in C.

Some adaptations can be made in testing the model, where the minimal score for proper classification; using raw output vs strict predictions, meaning that we can use the logistic regression model, however slightly changing the decision boundary. To better fit validation data.

### 3.2.4 Neural networks

A neural network can, depending on it's architecture, be equivalent to various other functions[3]. For example a net with no hidden layer and a sigmoid activated output, has the following structure. $\frac{1}{1+e^{\vec{w} \cdot \vec{x}}}$. As can be seen from

ALG179+1.p ALG194+1.p ALG198+1.p ALG200+1.p ALG201+1.p ALG202+1.p ALG203+1.p ALG211+1.p ALG235-1.p ALG340-1.p ALG350-1.p ALG371-1.p ALG372-1.p ALG397-1.p

the previous section this is an equivalent function to a logistic regression. Except here the method of determining weights differs, hence assuming that both weight assignment algorithms do converge to the same hidden distribution, we can view neural networks as a super set of the other methods used in this paper. Because of this fact I have decided to devote most training and testing to neural-networks.

To determine the exact architecture the training data is used to fit the model, where evaluations on the validation data is used to determine it's architecture. A set of conjectured options and combinations are fit on the validation data and these will ultimately be exported to E. This approach can be extended by recreating the data several times after it has been generated by neural network heuristics, however on the current version of Linux subsystem bash this is not feasible [3].

#### 3.2.5 Degenerate Trees (logistic)

A degenerate tree is a decision tree where all decisions have are in a single direction, hence it is effectively a linked list of decision nodes, where every leaf node except the deepest is a negative result. For this ensemble it is critical to achieve as high a recall as possible, while minimizing false positives. from the preceding four models only the logistic regression has an easy extension to this type of behaviour

Using the logit output of the logistic regression that we can change it's threshold line to effectively create 1.0 recall or a close approximation of it, it's possible to use many logistic models, with different true negatives, to ensemble a powerful classifier. This type of ensemble is an adaptation from the well known Viola Jones algorithm. [15]

# 4 Evaluations

for raw data see the appendix

## 4.1 Experimental setup

**Technical details** :
lenovo y570k 80WK005QMH
OS windows 10 / Linux subsystem / gcc
(modified)Eprover 2.3 Gielle
powershell / bash
memory allocation: 6GB

[3] currently bash has problem releasing file locks in many cases, as does the rest of the Linux subsystem for Windows, hence any looping and re-selecting of files and models is extremely difficult

**Feature selection** There are several linear models, SVMs and linear regression, which have not been included in this paper but have simply been used as dummies to detect useful features. The data gathered for this purpose is not from the ALG or any other group of problems used to evaluate the algorithms. It is very much inline with what E does by default. The features that are deemed most useful from this preliminary testing are those which the E prover uses by default. The use of maximal terms, positive terms, etc. Hence the features are simply those elements of a clause which, are subject to multipliers, not subject to multipliers and clause types. Therefore we have features of the type, "count of non-positive non-maximal function symbols". and "clause is goal". Meaning that the original heuristic of Refined weight is simply a weighted sum of the elements of our features vector.

**Data gathering** The data is gathered using the default settings of the E prover, this is done so that we can see this data as the original policy $\pi$. We cannot start from the usual case where we simply take random steps. Because a simple random policy $\pi'$ does not produce enough data, as it rarely ever finds a proof. From the outputs of the E prover on the ALG set, only those for which the default setting of the prover finds, a proof or a satisfiable set, are included in the dataset. This is done to determine which parts of the proof are productive and which are not. Considering a clause which is not in the proof is unproductive and something the heuristic should seek to avoid. Hence after selecting a subset of files the data is split into two large CSV files it is then labeled with class 0, in the proof object and derivation, and class 1, outside of the proof object. This is then further split into training and testing data, where the training data is split further into training data and validation data.

## 4.2 Preliminary testing

**Control group** Preliminary testing does show that all random models have extraordinarily poor performance. These control models achieve almost no proofs, hence we can already conclude that at least a relation is learned by all those algorithms which pass this sanity check.

**Support vector machines** While the SVM for classification performs fine, the ranking SVM based on this is incapable of achieving any results, even when tested on problems in it's original dataset. More detailed testing of this particular model shows that the SVM has negative weights for some of it's features, this results in a situation where the clause set of the prover can contain clauses like $f(y) \simeq x$ and $C^{'} \vee \phi(x)$, which can produce something of the type $C^{'} \vee \phi(f(y))$ , which in turn can be combined with $y \simeq g(x)$ to produce $C^{'} \vee \phi(f(g(x))$. [4] Given that our model assigned negative weights to the maximal literal of a

[4] Depending on the ordering this may or may not eventually terminate. As the ordering determines which clause is maximal And restricts the paramodulation.

clause, it simply generated clauses, with ever larger maximal literals. Ending with larger clauses and a timeout or out of memory crash.

**Logistic Regression** This method performs really well in terms of the validation set, achieving 95% + accuracy on both training and validation data. attempting to change the level of recall and or accuracy by changing the cutoff does not improve performance, hence we set it to 0.5.

**Linear Regression** Like the SVM the oridnary linear regressions get stuck trying to minimize negative features, however there exists a special extension to this type of regressions, where we ensure the results are always positive. Because this forces a positive intercept and coefficients the result can not decrease whenever the features increase. This prevents the crashing behaviour that undermines the normal regression.

**Degenerate trees** None of the trees of logistic regressions seem to really improve on the original, achieving only below 90% validation accuracy. Recall only improves when accuracy decreases. And the maximal performance is achieved using only a single logistic regression, hence I have removed the degenerate tree from the results. As it is literally equivalent to a logistic regression.

**Neural networks** Many architectures of neural networks are used to simulate some of the other algorithms in this paper, logistic and linear regression, however these provide inferior results and have been dropped accordingly. Based on these attempts a large set of possible options has been determined. All networks are a combination the following options,L1 regularization, L2 regularization, dropout, Relu, TanH, Sigmoid, Hingeloss, Cross entopy loss, mean squared error loss and absolute error loss. Using these in addition to several architecture sizes, the only well performing classification networks use Cross entropy, with no regularization other than drop out and a (16,16,1) or (16,1) size, Relu for the hidden layers and sigmoid as last activation.

The performance of these classifiers in preliminary testing was inline with expectations except for some rare cases, This seems to be due to similar reasons for the failure of the ranking SVM, the neural networks develop negative weights thereby becoming stuck and crash due to memory errors. Because of this the previous process has been repeated in Keras, because it allows for constraints on the weights. When, all weights are non negative, and the activation functions are monotonic and have a Range $[0, \infty)$, it follows that the neural-network will have monotonic-like properties, whereby for a function $f : \mathbb{R}^n \to \mathbb{R}$ given $(x_1, x_2, .., x_n)$ and $(x_1^{'}, x_2, ..., x_n)$ and $x_1 < x_1{}'$ it follows that $f(x_1, x_2, .., x_n) \leq f(x_1^{'}, x_2, .., x_n)$ this is also true for all $x \in (x_1, x_2, .., x_n)$ . This monotonic-like property should avoid such behaviour.

Other changes have also been tested with the Keras module, the introduction of a diagonal matrix $\boldsymbol{M}$, whose elements are $\frac{1}{max}$ for their respective features, as explained above the smallest case is 0 for all features, hence the normalizations

$\frac{x_i - min_i}{max_i - min_i}$ are equal to $x_i \cdot \frac{1}{max_i}$. Prepending this $\boldsymbol{M}$ to the model does not positively affect it's performance in either validation or any intermediate test. Likewise prepending $\boldsymbol{M}$ as a learnable layer has no increase in accuracy or recall. Even when all models are tested using different learning rates across special layers etc, there is no significant improvement.

The above architectures have also been considered on the regression networks, The (16,16,1) architecture with relu activation, mean squared error and dropout performs best. I have also attempted a recurrent Network, but in preliminary testing this network fails due to assigning negative weights, it also fails when constraint to strictly positive weights. Moreover an attempt has been made to use the learned regression as the starting point of the data gathering, this is "Regression network v2" in the table.

### 4.3 Testing Results

The following test have been performed on several combinations of these new priority functions and clause weight functions, which also include the default settings of the E prover. When testing several of these combinations, it seems like the (3,1,1)[5] split is determined by testing on the training set of problems. The results obtained are from this exact command: [6].

```
./eprover -D"ex1=Refinedweight(<custom prio function>,2,1,1.5,1.1,1)
ex2=Refinedweight(PreferNonGoals,2,1,1.5,1.1,1.1)
ex3=Refinedweight(PreferGoals, 2,1,1.5,1.1,1)"  --cpu-limit=20
--output-level=0 -H"new=(3*ex1,1*ex2,1*ex3)" -x new $f
```

Where for all classifiers the Clause weight is Refinedweight[7], and for all clause weight functions the following command is used.

```
./eprover -D"ex1=<custom clause weight>(PreferNonGoals)
ex2=<custom clause weight>(PreferGoals)"  --cpu-limit=20
--output-level=0 -H"new=(3*ex1,1*ex2)" -x new $f
```

The previous command gives us the following result, where clauses are determined running a separate command with higher "–output-level", the total clauses in the table are the sum of their respective intersections, clauses under ALG classifier are the result of the intersection of all proven ALG problems, likewise ALG regression, etc.

[5] the 3 can be made slightly larger but this won't improve the performance much

[6] these might have to be modified given the binary "eprover" location and /or operating system / bash version

[7] the default E function

| | ALG test set | | BOO | | COM | |
|---|---|---|---|---|---|---|
| classifier | clauses | proofs | clauses [8] | proofs | clauses | proofs |
| default | 1958649 | 94 | 12017 | 55 | 37706 | 24 |
| classifier SVM | 1746017 | 98 | 11450 | 53 | 15265 | 25 |
| Logistic Classifier | 1755604 | 94 | 15754 | 44 | 15694 | 24 |
| (16,16,1) NN | 1728466 | 98 | 12022 | 55 | 15338 | 24 |
| (16,1) NN | 1728239 | 98 | 12020 | 55 | 15502 | 24 |
| (16,16,1) constrained NN | 1728462 | 98 | 12020 | 55 | 15492 | 24 |
| (16,1) constrained NN | 1729340 | 98 | 12020 | 55 | 15492 | 24 |
| regression | | | | | | |
| default | 696102 | 94 | 8668 | 55 | 3896 | 24 |
| Ranking SVM | None | None | None | None | None | None |
| Linear Regression | None | None | None | None | None | None |
| RNN | None | None | None | None | None | None |
| NonNeg Linear Regression | 239719 | 104 | 11445 | 52 | 7950 | 24 |
| Regression network 5 epochs | 274566 | 99 | 504496 | 39 | 25843 | 24 |
| Regression network v2 | 283375 | 93 | 413596 | 41 | 26296 | 24 |

Table 1: summary table 1

This section is the comparison of proofs to default, where faster, means less clauses processed, equal the exact same, and slower more clauses processed.

| | SVM | logistic | NN1 | NN2 | NN3 | NN4 | noneg | reg ntwrk |
|---|---|---|---|---|---|---|---|---|
| ALG | | | | | | | | |
| faster | 56 | 53 | 51 | 51 | 53 | 53 | 52 | 51 |
| equal | 16 | 6 | 8 | 6 | 6 | 7 | 20 | 17 |
| slower | 19 | 32 | 34 | 34 | 31 | 31 | 6 | 10 |
| BOO | | | | | | | | |
| faster | 15 | 12 | 0 | 0 | 0 | 0 | 9 | 6 |
| equal | 5 | 2 | 38 | 39 | 39 | 39 | 5 | 4 |
| slower | 21 | 27 | 3 | 2 | 2 | 2 | 25 | 29 |
| COM | | | | | | | | |
| faster | 14 | 14 | 12 | 15 | 15 | 15 | 10 | 10 |
| equal | 3 | 2 | 5 | 6 | 7 | 7 | 3 | 2 |
| slower | 7 | 8 | 7 | 3 | 2 | 2 | 7 | 8 |

Table 2: summary table 2

## 4.4 Analysis

The raw steps tables, which include the entire intersection for ALG, are too large and have been included in the appendix. However the above provide a good summary. In other tests, preliminary tests and features selections etc, the behaviour of the classifiers roughly follows that of the COM data set, except for the one case of the BOO dataset which is a unique example, where the neural networks almost perfectly follow the default settings of E. I have yet to uncover a problem set where the classification neural net policies underperform when compared to the default policy. Though these probably exist.

Moreover over a Chi-square test on the BOO results, between NNets, SVM and default cannot reject the null that these are homogeneous populations[9], with respect to the policy strategy. Therefore it is impossible to state that in any case the new provers take more steps or find fewer proofs when given a random sample. Likewise a T test for the classification Heuristics only ever displays a reduction in steps or no rejection of the null, when comparing these proof sets, however it should be noted that none of the problem sets follow a normal distribution, thereby they violate the assumptions of the T test.

Several of the generic weight functions computed are completely dysfunctional and should be discarded. Given expert knowledge for constraints and changes to, class weight, and loss, they can be modified to create desirable behaviour, the problem of negative weights are a good example, but more tests to get at the details of this have to be done.

Moreover the regression models do actually underperform in many cases, when they are evaluated on problem sets, which are not of the same type as the training data.

[9] see appendix for the table

# 5 Related work

## 5.1 E prover

There is a history of machine learning optimizations for the Eprover, however the most used techniques in those papers are store and recall [5]. The closest paper to this one is the use of the LIBLINEAR[6] library SVM,by [8], to construct a prediction function, which also adds the length in symbols to the same weight. I have used this idea in prelimenary testing aswell, but it is not very applicable in my case.[10]

## 5.2 tableaux

There is also research on introducing new machine learned proof search for different calculi, such as connection tableaux [14], however these leverage proof state descriptions that don't exist in the E prover.

## 5.3 probabilistic sat solving

Recently Stanford University published paper on sat solving using a neural network to predict it's satisfiability [12] "When it guesses sat, we can almost always decode the satisfying assignment it has found from its activations." [4], within polynomial time. It is a closely related subject, however it critically differs in that it is not focused on proving. Other neural net based approaches do exist for various provers[7], however these lack some of the specifications required to re-implement and compare them.

# 6 Conclusion

**summary** It is clear from the results that we can improve the search heuristic in steps, real time and scope at the same time, however this improvement seems to mostly generalize to other problems within the same set. Moreover despite the fact that the adding and subtracting of the hyper-parameter to the existing heuristic makes this heuristics a nonlinear function, with respect to it's features, the linear regression that fits on the least squares estimate is still the best performing heuristic. Moreover there is a sampling bias that might have crept into the prover as it loses performance on very small proofs, but offsets this by gains made in performance on the larger proofs. This happens in the training set(see ALG194+1.p ALG200+1.p) as well as the test set.

[10] It is both fundamentally different from what I do here, and I am unable to find the exact library used, since they reference a library that has implemented L2-SVM after the publishing date of the paper.

**future work** This paper can be summarized as finding a policy $\pi'$ from $\pi$, which is the basis of reinforcement learning algorithms. Given that it is possible to automate all steps of this paper, we can setup the following program:

```
Train, Test, Validation = Shuffle_and_split(tptp_problems)
for(training_length):
    sample <- Sample(train)
    for(problem in sample):
        features.append( proof(problem) )
    test_policy <- learn(policy,features)
    if( check(test_policy,Validation) ):
        policy <- test_policy
    logg(features)
    features <- empty;
done
```

Since the original update of the policy increases the amount of proofs found and reduces the number of steps of the program, without any clear cost. I would expect this program to create an incrementally better prover every iteration, up to some maximum. This could however require a major modification of the E prover, such as merging the clause picking heuristics into a single function mapping clauses, as well as requiring an extremely large amount of time and computational resources, several moths of running time is a conservative estimate of the requirements. Given that every update can take well up to a day to complete.

Another very interesting topic would be the retraining of a small network combined with L1 regularization, this results into a sparse graph, where it would be possible to understand the underlying logic of the decision procedure.

# References


[1] `https://theoremprover-museum.github.io/`. 2019-06-09.

[2] *E 2.3 User Manual preliminary version.*

[3] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[4] et.al. Daniel Selsam, Stanford University. Neuro Sat, February 2018.

[5] Jörg Denzinger, Matthias Fuchs, Christoph Goller, and Stephan Schulz. Learning from previous proof experience: A survey. 1999.

[6] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[7] Christoph Goller. A connectionist control component for the theorem prover setheo. In *In ECAI-94 Workshop on Combining Symbolic and Connectionist Processing*, page pages, 1994.

[8] Jan Jakubv and Josef Urban. Enigma: efficient learning-based inference guiding machine. In *International Conference on Intelligent Computer Mathematics*, pages 292–302. Springer, 2017.

[9] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[10] Alan =Robinson and Andrei Voronkov. *handbook of, automated reasoning, volume I.* elsevier, 2006.

[11] Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3):111–126, 2002.

[12] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[13] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[14] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. Malecop machine learning connection prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277. Springer, 2011.

[15] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

# 7 Appendix

## 7.1 code

The start of this loop is identical to the one that prints the clause features, the numbers that mutliply fmmp,fmmn etc are the weights learned(in this case sklearn non negative linear regression), they cannot be implemented by having the custom Heuristic accept a file description as the current bash version might insert a "\r" character which will make it fail to load.

```
double nnl_goal(void* data,Clause_p clause){
  Eqn_p handle;
  CustomParam_p local = data;
  ClauseCondMarkMaximalTerms(local->ocb, clause);
  double fmmp = 0;
  double fmmn = 0;
  double fmnp = 0;
  double fmnn = 0;
  double fnmp = 0;
  double fnmn = 0;
  double fnnp = 0;
  double fnnn = 0;

  double vmmp = 0;
  double vmmn = 0;
  double vmnp = 0;
  double vmnn = 0;
  double vnmp = 0;
  double vnmn = 0;
  double vnnp = 0;
  double vnnn = 0;

  for(handle = clause->literals;handle;handle=handle->next){
    //res += LiteralWeight(handle, max_term_multiplier, max_literal_multiplier, pos_multiplier, vweight, fweight, 1, false);
    //                                 m  m  p  v  f
    fmmp += LiteralWeight(handle, 1, 1, 1, 0, 1, 1, false);
    fmmn += LiteralWeight(handle, 1, 1, 0, 0, 1, 1, false);
    fmnp += LiteralWeight(handle, 1, 0, 1, 0, 1, 1, false);
    fmnn += LiteralWeight(handle, 1, 0, 0, 0, 1, 1, false);
    fnmp += LiteralWeight(handle, 0, 1, 1, 0, 1, 1, false);
    fnmn += LiteralWeight(handle, 0, 1, 0, 0, 1, 1, false);
    fnnp += LiteralWeight(handle, 0, 0, 1, 0, 1, 1, false);
    fnnn += LiteralWeight(handle, 0, 0, 0, 0, 1, 1, false);

    vmmp += LiteralWeight(handle, 1, 1, 1, 1, 0, 1, false);
    vmmn += LiteralWeight(handle, 1, 1, 0, 1, 0, 1, false);
    vmnp += LiteralWeight(handle, 1, 0, 1, 1, 0, 1, false);
    vmnn += LiteralWeight(handle, 1, 0, 0, 1, 0, 1, false);
    vnmp += LiteralWeight(handle, 0, 1, 1, 1, 0, 1, false);
    vnmn += LiteralWeight(handle, 0, 1, 0, 1, 0, 1, false);
    vnnp += LiteralWeight(handle, 0, 0, 1, 1, 0, 1, false);
    vnnn += LiteralWeight(handle, 0, 0, 0, 1, 0, 1, false);


  }
  return fmmp * 7.552057 + fmmn * 0.000000 + fmnp* 4.691503 + fmnn * 0.000000 +
        fnmp * 7.972812 + fnmn * 0.000000 + fnnp * 2.291421 + fnnn * 0.000000 +
        vmmp * 6.258440 + vmmn * 0.000000 + vmnp * 1.012416 + vmnn * 0.000000 +
        vnmp * 2.633825 + vnmn * 0.000000 + vnnp * 0.000000 + vnnn * 9.338033 + 166.5520588210747;
}
```

The following section is the implementation of some neural network functionality, it is copied from the training file, and is part of models.h. It simply defined the necessary functions from the training module. functions like "BackwardPass" and or "train" are not part of the E modifications and thereby not part of the E code. the blunt use of "NN* network1();" is due to previously mentioned issue with file i/o.

```
#ifndef NN_MODELS
#define NN_MODELS

typedef float (*act) (float x);
typedef float (*dist) (float x, float y);
typedef struct l
{
  float *inputs;   //store this layers input for dynamic computation of gradients
  float *output;   //store the output to allow for dynamic computation of gradients
  float *weights;  //store the weights of a network
  float *bias;     //store the bias of the layer
  float *error;    //store this layers error of gradient descent
  float *delta;    //store this layers derivative on it's output
  int in;           //in size for weight retrieval
  int out;          //out size for weight retrieval
  act S;            //activation function
  act SD;           //dynamic activation
  struct l *next;   //nextlayer NULL --> this layer is the output layer
  struct l *prev;   //prevlayer NULL --> this layer is the input layer
} layer;

typedef struct NetN
{
  layer *initial;   //inputlayer of the network
  float alpha;     //learning rate
  float epsilon;   //dropout rate --increase in later versions--
  float lambda;    //weight decay rate/ lambda in loss function
  dist loss;        //loss function
  dist lossDeriv;   //loss function derivative
  dist acc;         //accuracy function
  float (*regfunc)(struct NetN* network);
} NN;

float Relu (float x);
float Sigmoid (float x);
float* getWeight(int i, int j, layer * l);
void pred (float *in, layer * l);
float forward (float *in, NN* network);

NN* network1();
NN* network2();
NN* network3();
NN* network4();
NN* network5();
NN* network6();
NN* network7();
NN* network8();
NN* network9();
NN* network10();
#endif
```

## 7.2 tables

| ALG_problem | default | SVM | logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| ALG002-1 | 3352 | 11705 | 12780 | 5100 | 5100 | 5095 | 5095 |
| ALG007-1 | 448 | 441 | 998 | 448 | 448 | 448 | 448 |
| ALG011-1 | 44 | 33 | 33 | 33 | 35 | 33 | 33 |
| ALG012-1 | 3724 | 17236 | 6106 | 6106 | 6110 | 6106 | 6106 |
| ALG014+1 | 20 | 20 | 32 | 32 | 32 | 32 | 32 |
| ALG016+1 | 3115 | 4115 | 4130 | 4098 | 4098 | 4098 | 4098 |
| ALG017+1 | 25 | 24 | 37 | 37 | 37 | 37 | 37 |
| ALG018+1 | 32 | 24 | 24 | 24 | 28 | 24 | 24 |
| ALG019+1 | 32 | 24 | 24 | 24 | 28 | 24 | 24 |
| ALG020+1 | 302 | 279 | 283 | 267 | 267 | 267 | 267 |
| ALG021+1 | 424 | 424 | 424 | 425 | 425 | 425 | 425 |
| ALG022+1 | 483 | 480 | 480 | 481 | 481 | 481 | 481 |
| ALG023+1 | 337098 | 230003 | 229596 | 230305 | 230288 | 230250 | 230266 |
| ALG025+1 | 1904 | 925 | 600 | 595 | 595 | 595 | 595 |
| ALG026+1 | 37902 | 54101 | 53123 | 53198 | 53198 | 53198 | 53198 |
| ALG028+1 | 13583 | 3780 | 1390 | 1390 | 1390 | 1390 | 1390 |
| ALG029+1 | 64 | 58 | 58 | 58 | 58 | 58 | 58 |
| ALG030+1 | 64 | 58 | 58 | 58 | 58 | 58 | 58 |
| ALG031+1 | 847 | 720 | 720 | 684 | 684 | 684 | 684 |
| ALG032+1 | 802 | 802 | 802 | 802 | 802 | 802 | 802 |
| ALG033+1 | 815 | 815 | 829 | 815 | 815 | 815 | 815 |
| ALG036+1 | 20 | 18 | 18 | 18 | 18 | 18 | 18 |
| ALG037+1 | 17530 | 16684 | 16689 | 16694 | 16694 | 16694 | 16694 |
| ALG038+1 | 481166 | 479853 | 479823 | 479818 | 479818 | 479822 | 479824 |
| ALG039+1 | 20 | 20 | 24 | 24 | 24 | 24 | 24 |
| ALG040+1 | 32 | 24 | 24 | 24 | 28 | 24 | 24 |
| ALG041+1 | 32 | 24 | 24 | 24 | 28 | 24 | 24 |
| ALG042+1 | 302 | 279 | 283 | 267 | 267 | 267 | 267 |
| ALG043+1 | 1016 | 1016 | 1019 | 1019 | 1019 | 1019 | 1019 |
| ALG044+1 | 1075 | 1072 | 1075 | 1079 | 1079 | 1075 | 1075 |
| ALG045+1 | 9038 | 10888 | 10889 | 10889 | 10889 | 10889 | 10889 |
| ALG054+1 | 147 | 152 | 162 | 163 | 163 | 163 | 163 |
| ALG055+1 | 38667 | 13847 | 15422 | 13839 | 13839 | 13839 | 13839 |
| ALG058+1 | 8027 | 6008 | 7506 | 6004 | 6004 | 6004 | 6004 |
| ALG062+1 | 719 | 1267 | 1267 | 1267 | 1267 | 1267 | 1267 |
| ALG063+1 | 1765 | 1563 | 1762 | 1572 | 1572 | 1568 | 1569 |
| ALG067+1 | 1145 | 1482 | 972 | 1484 | 1484 | 1482 | 1484 |
| ALG069+1 | 36 | 21 | 21 | 21 | 21 | 21 | 21 |
| ALG070+1 | 48 | 43 | 46 | 43 | 43 | 43 | 43 |
| ALG071+1 | 5068 | 16217 | 10631 | 8185 | 8185 | 8185 | 8185 |
| ALG073+1 | 306 | 606 | 137 | 313 | 313 | 313 | 313 |
| ALG075+1 | 487 | 421 | 431 | 406 | 406 | 406 | 406 |

| ALG_problem | default | SVM | logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| ALG076+1 | 503 | 437 | 447 | 422 | 422 | 422 | 422 |
| ALG077+1 | 502 | 436 | 446 | 421 | 421 | 421 | 421 |
| ALG078+1 | 506 | 440 | 450 | 425 | 425 | 425 | 425 |
| ALG079+1 | 519 | 453 | 463 | 438 | 438 | 438 | 438 |
| ALG080+1 | 825 | 759 | 769 | 744 | 744 | 744 | 744 |
| ALG081+1 | 489 | 423 | 433 | 408 | 408 | 408 | 408 |
| ALG082+1 | 490 | 424 | 434 | 409 | 409 | 409 | 409 |
| ALG083+1 | 501 | 435 | 445 | 420 | 420 | 420 | 420 |
| ALG084+1 | 546 | 480 | 490 | 465 | 465 | 465 | 465 |
| ALG085+1 | 489 | 423 | 433 | 408 | 408 | 408 | 408 |
| ALG086+1 | 491 | 425 | 435 | 410 | 410 | 410 | 410 |
| ALG087+1 | 486 | 420 | 430 | 405 | 405 | 405 | 405 |
| ALG088+1 | 485 | 419 | 429 | 404 | 404 | 404 | 404 |
| ALG089+1 | 490 | 424 | 434 | 409 | 409 | 409 | 409 |
| ALG090+1 | 1039 | 1039 | 1040 | 1064 | 1064 | 1040 | 1040 |
| ALG091+1 | 1156 | 1156 | 1163 | 1161 | 1161 | 1161 | 1161 |
| ALG092+1 | 1029 | 1029 | 1030 | 1054 | 1054 | 1030 | 1034 |
| ALG093+1 | 1798 | 1798 | 1873 | 1803 | 1803 | 1803 | 1803 |
| ALG094+1 | 1017 | 1017 | 1022 | 1046 | 1046 | 1022 | 1025 |
| ALG095+1 | 1017 | 1016 | 1021 | 1046 | 1046 | 1021 | 1021 |
| ALG105+1 | 39404 | 40676 | 40947 | 40515 | 40515 | 40515 | 40515 |
| ALG107+1 | 191635 | 167768 | 168172 | 167721 | 167747 | 168135 | 168205 |
| ALG110+1 | 12995 | 17145 | 17130 | 17181 | 17181 | 17181 | 17181 |
| ALG111+1 | 2648 | 1019 | 1039 | 1033 | 1033 | 1033 | 1033 |
| ALG113+1 | 1213 | 787 | 760 | 778 | 778 | 779 | 779 |
| ALG114+1 | 1423 | 62062 | 62093 | 62073 | 62073 | 62062 | 62062 |
| ALG115+1 | 7093 | 6436 | 6837 | 6870 | 6905 | 6888 | 6888 |
| ALG117+1 | 1442 | 51363 | 50926 | 51747 | 51747 | 50914 | 50914 |
| ALG118+1 | 19265 | 18519 | 18789 | 18948 | 18940 | 18971 | 18977 |
| ALG121+1 | 1101 | 1093 | 1184 | 1112 | 1112 | 1112 | 1112 |
| ALG123+1 | 5859 | 5179 | 5122 | 5056 | 5056 | 5044 | 5054 |
| ALG125+1 | 92400 | 91177 | 91384 | 91487 | 91487 | 91482 | 91482 |
| ALG138+1 | 274 | 133 | 137 | 137 | 137 | 137 | 137 |
| ALG166+1 | 40477 | 37844 | 38994 | 38044 | 38044 | 38409 | 38643 |
| ALG167+1 | 349293 | 211041 | 222874 | 210594 | 209589 | 210329 | 210844 |
| ALG168+1 | 1346 | 1700 | 1706 | 1728 | 1728 | 1727 | 1727 |
| ALG169+1 | 1307 | 1659 | 1688 | 1699 | 1699 | 1699 | 1699 |
| ALG170+1 | 17861 | 17487 | 17894 | 17384 | 17384 | 17484 | 17484 |
| ALG171+1 | 42 | 144 | 144 | 144 | 144 | 144 | 144 |
| ALG172+1 | 57 | 171 | 171 | 171 | 171 | 171 | 171 |
| ALG173+1 | 40 | 109 | 109 | 109 | 109 | 109 | 109 |
| ALG174+1 | 67 | 179 | 179 | 179 | 179 | 179 | 179 |
| ALG175+1 | 58 | 160 | 163 | 163 | 163 | 163 | 163 |
| ALG176+1 | 84 | 112 | 66 | 67 | 67 | 67 | 67 |
| ALG177+1 | 80 | 62 | 62 | 62 | 62 | 62 | 62 |

| ALG_problem | default | SVM | logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| ALG178+1 | 128 | 122 | 122 | 122 | 122 | 122 | 122 |
| ALG179+1 | 128 | 122 | 122 | 122 | 122 | 122 | 122 |
| ALG180+1 | 468 | 395 | 404 | 379 | 379 | 379 | 379 |
| ALG181+1 | 417 | 351 | 361 | 336 | 336 | 336 | 336 |
| ALG182+1 | 417 | 351 | 361 | 336 | 336 | 336 | 336 |
| ALG183+1 | 417 | 351 | 361 | 336 | 336 | 336 | 336 |
| ALG184+1 | 417 | 351 | 361 | 336 | 336 | 336 | 336 |
| ALG185+1 | 1623 | 1487 | 1487 | 1462 | 1462 | 1462 | 1462 |
| ALG186+1 | 1010 | 1000 | 1007 | 1034 | 1034 | 1006 | 1006 |
| ALG187+1 | 1010 | 1001 | 1008 | 1035 | 1035 | 1007 | 1007 |
| ALG188+1 | 978 | 978 | 980 | 1001 | 1001 | 980 | 980 |
| ALG189+1 | 978 | 978 | 980 | 1001 | 1001 | 980 | 980 |
| ALG192+1 | 24127 | 19651 | 19663 | 19663 | 19663 | 19663 | 19663 |
| ALG193+1 | 163826 | 113007 | 113054 | 113069 | 113084 | 113089 | 113054 |
| ALG194+1 | 3108 | 932 | 932 | 932 | 932 | 932 | 932 |
| ALG198+1 | 32 | 18 | 28 | 28 | 28 | 28 | 28 |
| ALG199+1 | 112 | 2662 | 2676 | 2676 | 2676 | 2676 | 2676 |
| ALG200+1 | 1776 | 410 | 405 | 405 | 405 | 405 | 405 |
| ALG201+1 | 16 | 13 | 13 | 13 | 13 | 13 | 13 |
| ALG202+1 | 28 | 26 | 26 | 26 | 26 | 26 | 26 |
| ALG203+1 | 48 | 43 | 45 | 45 | 45 | 45 | 45 |
| ALG204+1 | 930 | 724 | 731 | 682 | 682 | 682 | 682 |
| ALG205+1 | 3077 | 2864 | 2870 | 2821 | 2821 | 2821 | 2821 |
| ALG206+1 | 1391 | 1185 | 1195 | 1143 | 1143 | 1143 | 1143 |
| ALG207+1 | 3690 | 3691 | 3719 | 3723 | 3723 | 3697 | 3697 |
| ALG208+1 | 3679 | 3679 | 3713 | 3729 | 3729 | 3684 | 3684 |
| ALG209+1 | 3690 | 3690 | 3718 | 3701 | 3701 | 3701 | 3701 |
| ALG210+2 | 74 | 109 | 129 | 66 | 66 | 66 | 66 |
| ALG211+1 | 28 | 20 | 20 | 20 | 20 | 20 | 20 |
| ALG227+1 | 1506 | 1338 | 1338 | 1328 | 1333 | 1338 | 1338 |
| ALG235-1 | 38 | 38 | 43 | 38 | 38 | 38 | 38 |
| ALG236-1 | 186 | 185 | 510 | 186 | 186 | 186 | 186 |
| ALG299-1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| ALG300-1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| ALG302-1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| ALG350-1 | 136 | 64 | 330 | 149 | 145 | 145 | 145 |
| ALG371-1 | 886 | 1296 | 2880 | 1213 | 1192 | 1267 | 1267 |
| ALG397-1 | 240 | 236 | 236 | 121 | 121 | 233 | 233 |

Table 3: classifiers: steps per ALG problem

| ALG_problem | default | nonneg | NNreg |
|---|---|---|---|
| ALG006-1 | 296 | 3012 | 36587 |
| ALG007-1 | 448 | 1553 | 23394 |

| ALG_problem | default | nonneg | NNreg |
|---|---|---|---|
| ALG011-1 | 44 | 31 | 31 |
| ALG014+1 | 20 | 30 | 30 |
| ALG016+1 | 3115 | 3097 | 3999 |
| ALG017+1 | 25 | 35 | 35 |
| ALG018+1 | 32 | 44 | 36 |
| ALG019+1 | 32 | 36 | 36 |
| ALG020+1 | 302 | 241 | 226 |
| ALG021+1 | 424 | 424 | 424 |
| ALG022+1 | 483 | 480 | 480 |
| ALG025+1 | 1904 | 1904 | 1904 |
| ALG026+1 | 37902 | 14051 | 7560 |
| ALG028+1 | 13583 | 8948 | 8788 |
| ALG029+1 | 64 | 56 | 64 |
| ALG030+1 | 64 | 56 | 64 |
| ALG031+1 | 847 | 626 | 551 |
| ALG032+1 | 802 | 802 | 802 |
| ALG033+1 | 815 | 815 | 815 |
| ALG036+1 | 20 | 20 | 20 |
| ALG037+1 | 17530 | 16406 | 16371 |
| ALG039+1 | 20 | 20 | 20 |
| ALG040+1 | 32 | 44 | 36 |
| ALG041+1 | 32 | 36 | 36 |
| ALG042+1 | 302 | 241 | 226 |
| ALG043+1 | 1016 | 1016 | 1016 |
| ALG044+1 | 1075 | 1072 | 1072 |
| ALG045+1 | 9038 | 3558 | 3033 |
| ALG054+1 | 147 | 133 | 133 |
| ALG055+1 | 38667 | 1727 | 1738 |
| ALG058+1 | 8027 | 2441 | 2877 |
| ALG062+1 | 719 | 427 | 474 |
| ALG063+1 | 1765 | 1101 | 1078 |
| ALG067+1 | 1145 | 733 | 803 |
| ALG069+1 | 36 | 32 | 32 |
| ALG070+1 | 48 | 40 | 40 |
| ALG075+1 | 487 | 348 | 330 |
| ALG076+1 | 503 | 365 | 346 |
| ALG077+1 | 502 | 363 | 345 |
| ALG078+1 | 506 | 365 | 349 |
| ALG079+1 | 519 | 381 | 362 |
| ALG080+1 | 825 | 688 | 662 |
| ALG081+1 | 489 | 352 | 332 |
| ALG082+1 | 490 | 354 | 333 |
| ALG083+1 | 501 | 356 | 344 |
| ALG084+1 | 546 | 408 | 389 |
| ALG085+1 | 489 | 352 | 332 |

| ALG_problem | default | nonneg | NNreg |
|---|---|---|---|
| ALG086+1 | 491 | 351 | 334 |
| ALG087+1 | 486 | 349 | 329 |
| ALG088+1 | 485 | 347 | 328 |
| ALG089+1 | 490 | 349 | 333 |
| ALG090+1 | 1039 | 1039 | 1039 |
| ALG091+1 | 1156 | 1156 | 1156 |
| ALG092+1 | 1029 | 1034 | 1034 |
| ALG093+1 | 1798 | 1798 | 1821 |
| ALG094+1 | 1017 | 1017 | 1017 |
| ALG095+1 | 1017 | 1017 | 1017 |
| ALG105+1 | 39404 | 22700 | 19193 |
| ALG107+1 | 191445 | 7386 | 2397 |
| ALG110+1 | 12995 | 8617 | 8511 |
| ALG111+1 | 2648 | 2362 | 2375 |
| ALG113+1 | 1213 | 1155 | 1127 |
| ALG114+1 | 1423 | 17382 | 18103 |
| ALG115+1 | 7093 | 5158 | 5093 |
| ALG117+1 | 1442 | 14430 | 15265 |
| ALG121+1 | 1101 | 1120 | 1173 |
| ALG138+1 | 274 | 523 | 523 |
| ALG166+1 | 40477 | 35949 | 33309 |
| ALG168+1 | 1346 | 1290 | 1270 |
| ALG169+1 | 1307 | 1278 | 1290 |
| ALG170+1 | 17861 | 11391 | 9395 |
| ALG171+1 | 42 | 35 | 35 |
| ALG172+1 | 57 | 49 | 49 |
| ALG173+1 | 40 | 40 | 40 |
| ALG174+1 | 67 | 59 | 59 |
| ALG175+1 | 58 | 58 | 58 |
| ALG176+1 | 84 | 40 | 40 |
| ALG177+1 | 80 | 40 | 40 |
| ALG178+1 | 128 | 112 | 100 |
| ALG179+1 | 128 | 112 | 100 |
| ALG180+1 | 468 | 333 | 313 |
| ALG181+1 | 417 | 284 | 260 |
| ALG182+1 | 417 | 284 | 260 |
| ALG183+1 | 417 | 284 | 260 |
| ALG184+1 | 417 | 284 | 260 |
| ALG185+1 | 1623 | 1443 | 1416 |
| ALG186+1 | 1010 | 1010 | 1010 |
| ALG187+1 | 1010 | 1010 | 1010 |
| ALG188+1 | 978 | 978 | 978 |
| ALG189+1 | 978 | 978 | 978 |
| ALG194+1 | 3108 | 1356 | 1356 |
| ALG198+1 | 32 | 32 | 32 |

| ALG_problem | default | nonneg | NNreg |
|---|---|---|---|
| ALG199+1 | 112 | 113 | 113 |
| ALG200+1 | 1776 | 1296 | 1332 |
| ALG201+1 | 16 | 16 | 16 |
| ALG202+1 | 28 | 28 | 28 |
| ALG203+1 | 48 | 48 | 52 |
| ALG204+1 | 930 | 512 | 478 |
| ALG205+1 | 3077 | 2811 | 2302 |
| ALG206+1 | 1391 | 971 | 904 |
| ALG207+1 | 3690 | 3690 | 3717 |
| ALG208+1 | 3679 | 3679 | 3725 |
| ALG209+1 | 3690 | 3690 | 3690 |
| ALG211+1 | 28 | 24 | 28 |
| ALG227+1 | 1506 | 689 | 481 |
| ALG235-1 | 38 | 115 | 420 |
| ALG299-1 | 3 | 3 | 3 |
| ALG300-1 | 3 | 3 | 3 |
| ALG302-1 | 2 | 2 | 2 |
| ALG340-1 | 544 | 2504 | 334 |
| ALG341-1 | 186505 | 320 | 328 |
| ALG350-1 | 136 | 93 | 60 |
| ALG371-1 | 886 | 779 | 1008 |
| ALG372-1 | 298 | 475 | 339 |
| ALG397-1 | 240 | 219 | 162 |

Table 4: regressor steps per ALG problem

| BOO_problem | default | SVM | Logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| BOO001-1 | 13 | 20 | 15 | 13 | 13 | 13 | 13 |
| BOO002-1 | 132 | 254 | 44 | 132 | 132 | 132 | 132 |
| BOO002-2 | 131 | 254 | 44 | 131 | 131 | 131 | 131 |
| BOO003-2 | 69 | 50 | 104 | 69 | 69 | 69 | 69 |
| BOO003-4 | 69 | 40 | 24 | 69 | 69 | 69 | 69 |
| BOO004-2 | 17 | 36 | 97 | 18 | 18 | 18 | 18 |
| BOO004-4 | 18 | 25 | 55 | 18 | 18 | 18 | 18 |
| BOO005-2 | 19 | 28 | 130 | 19 | 19 | 19 | 19 |
| BOO005-4 | 19 | 27 | 31 | 19 | 19 | 19 | 19 |
| BOO006-2 | 70 | 47 | 131 | 70 | 70 | 70 | 70 |
| BOO006-4 | 70 | 41 | 26 | 70 | 70 | 70 | 70 |
| BOO007-4 | 1332 | 2177 | 6564 | 1332 | 1332 | 1332 | 1332 |
| BOO009-2 | 71 | 47 | 127 | 71 | 71 | 71 | 71 |
| BOO009-4 | 71 | 42 | 47 | 71 | 71 | 71 | 71 |
| BOO010-2 | 70 | 80 | 144 | 70 | 70 | 70 | 70 |
| BOO010-4 | 70 | 50 | 60 | 70 | 70 | 70 | 70 |
| BOO011-1 | 17 | 17 | 17 | 19 | 17 | 17 | 17 |

| BOO_problem | default | SVM | Logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| BOO011-2 | 6 | 6 | 9 | 6 | 6 | 6 | 6 |
| BOO011-4 | 7 | 10 | 11 | 7 | 7 | 7 | 7 |
| BOO012-2 | 126 | 37 | 125 | 126 | 126 | 126 | 126 |
| BOO012-4 | 126 | 72 | 54 | 126 | 126 | 126 | 126 |
| BOO013-2 | 245 | 118 | 161 | 245 | 245 | 245 | 245 |
| BOO013-4 | 185 | 71 | 334 | 185 | 185 | 185 | 185 |
| BOO014-2 | 2170 | 800 | 4345 | 2170 | 2170 | 2170 | 2170 |
| BOO016-2 | 68 | 94 | 131 | 68 | 68 | 68 | 68 |
| BOO017-2 | 72 | 65 | 145 | 72 | 72 | 72 | 72 |
| BOO018-4 | 11 | 15 | 15 | 11 | 11 | 11 | 11 |
| BOO021-1 | 78 | 29 | 89 | 78 | 78 | 78 | 78 |
| BOO026-1 | 251 | 305 | 304 | 251 | 251 | 251 | 251 |
| BOO027-1 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| BOO028-1 | 4345 | 4577 | 926 | 4345 | 4345 | 4345 | 4345 |
| BOO029-1 | 350 | 226 | 422 | 350 | 350 | 350 | 350 |
| BOO034-1 | 28 | 40 | 35 | 30 | 30 | 30 | 30 |
| BOO068-1 | 527 | 533 | 93 | 527 | 527 | 527 | 527 |
| BOO069-1 | 57 | 57 | 60 | 57 | 57 | 57 | 57 |
| BOO070-1 | 522 | 529 | 91 | 522 | 522 | 522 | 522 |
| BOO071-1 | 55 | 55 | 58 | 55 | 55 | 55 | 55 |
| BOO072-1 | 133 | 145 | 145 | 133 | 133 | 133 | 133 |
| BOO074-1 | 153 | 165 | 165 | 153 | 153 | 153 | 153 |
| BOO075-1 | 237 | 259 | 369 | 237 | 237 | 237 | 237 |

Table 5: classifiers: steps per BOO problem

| BOO_problem | default | nonneg | NNreg |
|---|---|---|---|
| BOO001-1 | 13 | 24 | 14 |
| BOO003-2 | 69 | 200 | 27296 |
| BOO003-4 | 69 | 200 | 27296 |
| BOO004-2 | 17 | 18 | 20 |
| BOO004-4 | 18 | 18 | 20 |
| BOO005-2 | 19 | 20 | 21 |
| BOO005-4 | 19 | 18 | 21 |
| BOO006-2 | 70 | 198 | 27469 |
| BOO006-4 | 70 | 198 | 27469 |
| BOO009-2 | 71 | 205 | 27373 |
| BOO009-4 | 71 | 205 | 27373 |
| BOO010-2 | 70 | 198 | 27397 |
| BOO010-4 | 70 | 198 | 27397 |
| BOO011-1 | 17 | 17 | 17 |
| BOO011-2 | 6 | 8 | 6 |
| BOO011-4 | 7 | 10 | 7 |
| BOO012-2 | 126 | 259 | 28141 |

| BOO_problem | default | nonneg | NNreg |
|---|---|---|---|
| BOO012-4 | 126 | 259 | 28141 |
| BOO013-2 | 245 | 1270 | 29727 |
| BOO013-4 | 185 | 1034 | 29079 |
| BOO014-2 | 2170 | 2106 | 33673 |
| BOO015-2 | 2152 | 2158 | 39881 |
| BOO016-2 | 68 | 202 | 27536 |
| BOO017-2 | 72 | 206 | 27401 |
| BOO018-4 | 11 | 12 | 11 |
| BOO021-1 | 78 | 67 | 28276 |
| BOO024-1 | 222 | 137 | 5640 |
| BOO025-1 | 217 | 153 | 5657 |
| BOO026-1 | 251 | 349 | 433 |
| BOO027-1 | 7 | 7 | 9 |
| BOO029-1 | 350 | 489 | 776 |
| BOO034-1 | 28 | 157 | 208 |
| BOO068-1 | 527 | 112 | 54 |
| BOO069-1 | 57 | 57 | 48 |
| BOO070-1 | 522 | 113 | 56 |
| BOO071-1 | 55 | 55 | 47 |
| BOO072-1 | 133 | 87 | 83 |
| BOO074-1 | 153 | 96 | 101 |
| BOO075-1 | 237 | 325 | 322 |

| COM_problem | default | SVM | Logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| COM001-1.p | 716 | 31 | 202 | 197 | 197 | 197 | 197 |
| COM001_1.p | 716 | 31 | 202 | 197 | 197 | 197 | 197 |
| COM002-1.p | 15332 | 6040 | 5733 | 5733 | 5733 | 5733 | 5733 |
| COM002_2.p | 60 | 40 | 45 | 65 | 36 | 36 | 36 |
| COM002_1.p | 15324 | 6038 | 5740 | 5733 | 5733 | 5733 | 5733 |
| COM002_2.p | 456 | 95 | 241 | 226 | 186 | 186 | 186 |
| COM003-2.p | 72 | 51 | 51 | 51 | 55 | 51 | 51 |
| COM003_1.p | 741 | 825 | 807 | 489 | 732 | 732 | 732 |
| COM004-1.p | 16 | 13 | 13 | 24 | 13 | 13 | 13 |
| COM007+1.p | 2680 | 516 | 1021 | 1016 | 1016 | 1021 | 1021 |
| COM007+2.p | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| COM009-2.p | 6 | 5 | 5 | 5 | 5 | 5 | 5 |
| COM010-2.p | 5 | 6 | 6 | 5 | 5 | 5 | 5 |
| COM011-2.p | 474 | 515 | 515 | 515 | 515 | 515 | 515 |
| COM012+1.p | 54 | 54 | 54 | 56 | 56 | 54 | 54 |
| COM012+3.p | 63 | 63 | 66 | 63 | 63 | 63 | 63 |
| COM013+4.p | 208 | 202 | 207 | 180 | 180 | 180 | 180 |
| COM016+4.p | 57 | 49 | 66 | 67 | 68 | 59 | 59 |
| COM018+1.p | 42 | 41 | 41 | 37 | 41 | 41 | 41 |
| COM018+4.p | 69 | 84 | 79 | 69 | 69 | 69 | 69 |

| COM_problem | default | SVM | Logistic | NN1 | NN2 | NN3 | NN4 |
|---|---|---|---|---|---|---|---|
| COM021+4.p | 152 | 156 | 153 | 152 | 152 | 152 | 152 |
| COM022+1.p | 121 | 129 | 134 | 124 | 121 | 121 | 121 |
| COM022+4.p | 153 | 80 | 139 | 154 | 149 | 149 | 149 |
| COM023+1.p | 156 | 168 | 141 | 147 | 147 | 147 | 149 |

Table 7: classiefiers: steps per COM problem

| COM_problem | default | nonneg | NNreg |
|---|---|---|---|
| COM001-1 | 716 | 2600 | 9540 |
| COM001_1 | 716 | 2600 | 9540 |
| COM002-2 | 60 | 40 | 36 |
| COM002_2 | 456 | 360 | 220 |
| COM003-2 | 72 | 52 | 56 |
| COM003_1 | 741 | 1164 | 5268 |
| COM004-1 | 16 | 44 | 40 |
| COM007+2 | 33 | 27 | 27 |
| COM009-2 | 6 | 6 | 5 |
| COM010-2 | 5 | 5 | 5 |
| COM012+1 | 54 | 54 | 54 |
| COM012+3 | 63 | 58 | 75 |
| COM013+4 | 208 | 228 | 332 |
| COM016+4 | 57 | 36 | 41 |
| COM018+1 | 42 | 41 | 41 |
| COM018+4 | 69 | 99 | 121 |
| COM021+4 | 152 | 125 | 79 |
| COM022+1 | 121 | 123 | 122 |
| COM022+4 | 153 | 138 | 105 |
| COM023+1 | 156 | 140 | 136 |