Learning Precedences from Simple Symbol Features[*]

Filip Bártek and Martin Suda

Czech Technical University in Prague, Czech Republic

Abstract

A simplification ordering, typically specified by a symbol precedence, is one of the key parameters of the superposition calculus, contributing to shaping the search space navigated by a saturation-based automated theorem prover. Thus the choice of a precedence can have a great impact on the prover's performance. In this work, we design a system for proposing symbol precedences that should lead to solving a problem quickly. The system relies on machine learning to extract this information from past successful and unsuccessful runs of a theorem prover over a set of problems and randomly sampled precedences. It uses a small set of simple human-engineered symbol features as the sole basis for discriminating the symbols. This allows for a direct comparison with precedence generation schemes designed by prover developers.

1 Introduction

Modern saturation-based automated theorem provers (ATPs) such as E [18], SPASS [22] or Vampire [8] use the superposition calculus [10] as their underlying inference system. Superposition is built around the paramodulation inference [16] crucially constrained by simplification ordering on terms and literals, which is supplied as a parameter of the calculus. Both of the two main classes of simplification orderings used in practice, i.e., the Knuth-Bendix Ordering [7] and the Lexicographic Path Ordering [6], are mainly determined by a *symbol precedence*, a (partial) ordering on the signature symbols.[1]

While the superposition calculus is known [1] to be refutationally complete for any simplification ordering, the choice of the precedence may have a significant impact on how long it takes to solve a given problem. In a well-known example, prioritizing in the precedence the predicates introduced during the Tseitin transformation of an input formula [21] exposes the corresponding literals to resolution inference during early stages of the proof search, with the effect of essentially undoing the transformation and thus threatening with an exponential blow-up that the transformation is designed to prevent [14]. ATPs typically offer a few heuristic schemes for generating the symbol precedences. For example, the successful `invfreq` scheme in E [17] orders the symbols by the number of occurrences in the input problem, prioritizing symbols that occur the least often for early inferences. Experiments with random precedences have shown that the existing schemes often fail to come close to the optimum precedence [13], revealing there is a large potential for further improvements.

In this work, we design a system that, when presented with a First-Order Logic (FOL) problem, proposes a symbol precedence that will likely lead to solving the problem quickly. The system relies on the techniques of supervised machine learning and extracts such theorem-proving knowledge from successful (and unsuccessful) runs of the Vampire theorem prover [8]

[*]Supported by the ERC Consolidator grant AI4REASON no. 649043 under the EU-H2020 programme, the Czech Science Foundation project 20-06390Y and the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37.

[1]KBO is further parameterized by symbol weights, but our reference implementation in Vampire [8] uses for efficiency reasons only weights equal to one [9] and so we do not consider this parameter here.

when run over a variety of FOL problems equipped with randomly sampled symbol precedences. We assume that by learning to solve already solvable problems quickly, the acquired knowledge will generalize and help solving problems previously out of reach. As a first step in a more ambitious project, we focus here on representing the symbols in a problem by a fixed set of simple human-engineered features (such as the number of occurrences used by `invfreq` scheme mentioned above)[2] and, to simplify the experimental setup, we restrict our attention to learning precedences for predicate symbols only.[3]

Learning to predict good precedences poses several interesting challenges that we address in this work. First, it is not immediately clear how to characterize a precedence, a permutation of a finite set of symbols, by a real-valued feature vector to serve as an input for a learning algorithm. Additionally, to be able to generalize across problems we need to do it in a way which does not presuppose a fixed signature size. There is also a complication, when sampling different problems, that some problems may be easy to solve for almost every precedence and others hard. In theorem proving, running times typically vary considerably. Finally, even with a regression model ready to predict the prover's performance under a particular precedence π, we still need to solve the task of finding an optimum precedence π^* according to this model, which cannot be simply solved by enumerating all the permutations and running the prediction for each due to their huge number.

Our way of addressing the above sketched challenges lies in using *pairwise symbol preferences* to characterize a precedence, normalizing the target prover run times on a per problem basis, and in the use of "second-order" learning of the preferences for symbols abstracted by their features. These concepts are introduced in Section 3 and later formalized in Section 4. Section 5 presents the results of our experimental evaluation of the proposed technique over the TPTP [19] benchmark. We start our exposition by fixing the notation and basic concepts in Section 2.

2 Preliminaries

We assume that the reader is familiar with basic concepts used in first order logic (FOL) theorem proving. We use this section to recall and formalize the key notions relevant for our work.

Problem A *(first-order) problem* is a pair $P = (\Sigma, Cl)$, where $\Sigma = (s_1, s_2, \ldots, s_n)$ is a list of (predicate and function) symbols called the *signature*, and Cl is a set of first-order clauses built over the symbols of Σ.

The problem is either given directly by the user or could be the result of clausifying a general FOL formula φ, in which case we know which of the symbols were introduced during the clausification (namely during Tseitin transformation and skolemization; see e.g. Nonnengart and Weidenbach [11]) and which occurred in the conjecture (if it was present).

Precedence Given a problem $P = (\Sigma, Cl)$ with $\Sigma = (s_1, s_2, \ldots, s_n)$, a precedence π_P is a permutation, i.e. a bijective mapping, of the set of indices $\{1, \ldots, n\}$. A precedence π_P *determines* a (total) ordering on Σ as follows: $s_{\pi_P(1)} < s_{\pi_P(2)} < \ldots < s_{\pi_P(n)}$.

Simplification Orderings are orderings on terms used to parameterize the superposition calculus [10] employed by modern saturation-based theorem provers. The two classes of simplification orderings most commonly used in practice, the Knuth-Bendix Orderings [7] and the

[2]Automatic feature extraction using neural networks is planned for future work.
[3]Our theoretical considerations, however, apply equally to learning function symbol precedences.

Lexicographic Path Orderings [6], are both defined in terms of a user-supplied (possibly partial) ordering $<$ on the given problem's signature Σ. In this work, we assume that the theorem prover uses a simplification ordering from one of these two classes relying on the ordering on Σ determined by a precedence π_P to construct such a simplification ordering.

Performance measure A saturation-based ATP *solves* a problem $P = (\Sigma, Cl)$ (under a particular fixed strategy and a determined symbol precedence π_P) by either

- deriving from Cl a contradiction in the form of the empty clause, in which case P is shown *unsatisfiable*, or
- finitely saturating the set of clauses Cl without deriving the contradiction, in which case P is shown *satisfiable*.[4]

In both cases, we take the number of iterations of the employed saturation algorithm (see, e.g., Riazanov and Voronkov [15] for an overview) as a measure of the effort that the ATP took to solve the problem. We refer to this measure as the *abstract solving time* and denote it $ast(P, \pi_P)$.[5]

In practice, an ATP can also run out of resources, typically out of the allocated time. In that case, the abstract solving time is *undefined*: $ast(P, \pi_P) = \bot$. While it may happen that running an ATP with the same problem and symbol precedence two times yields a different result each time (namely succeeding one time and failing another time), such cases are rare and we ensure they do not interfere with the learning process by caching the results.

Order matrix Given a permutation π of the set of indices $\{1, \ldots, n\}$, the order matrix $O(\pi)$ is a binary matrix of size $n \times n$ defined in the following manner:

$$O(\pi)_{i,j} = \llbracket \pi^{-1}(i) < \pi^{-1}(j) \rrbracket ,$$

where we use $\llbracket P \rrbracket$ to denote the Iverson bracket [4] applied to a proposition P, evaluating to 1 if P is true, and 0 otherwise. In other words, for a symbol precedence π_P, $O(\pi_P)_{i,j} = 1$ if the precedence π_P orders the symbol s_i before the symbol s_j, and $O(\pi_P)_{i,j} = 0$ otherwise.

Flattened matrix Given a matrix M of size $n \times n$, $\overrightarrow{M}$ is the vector of length n^2 obtained by flattening M:

$$\overrightarrow{M}_{(i-1)n+j} = M_{i,j}$$

for every $i, j \in \{1, \ldots, n\}$. For our use the exact way of mapping the matrix elements to the vector indices is not important. We mostly just need a vector representation of the data contained in the given matrix to have access to the dot product operation.

Linear regression is an approach to modeling the relationship between scalar *target* values $y_i \in \mathbb{R}$ and one or more *input* variables $\mathbf{x}_i = (x_i^1, \ldots, x_i^k)$, $i = 1, \ldots, n$. The relationship is modeled using a linear predictor function:

$$\hat{y}_i = \mathbf{x}_i \cdot \mathbf{w} + b,$$

[4] We assume a *refutationally complete* calculus and saturation strategy.

[5] The advantage of using abstract solving time is that it does not depend on the hardware used for the computation.

whose unknown model parameters $\mathbf{w} \in \mathbb{R}^k$ and $b \in \mathbb{R}$ are estimated from the data. We call the vector $\mathbf{w}$ the *coefficients* of the model and b the *intercept*. Most commonly, the parameters are picked to minimize the so-called mean squared error:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2,$$

but other norms are also possible [3].

Basic assumptions For the discussion that follows, we assume a fixed ATP that uses the superposition calculus with a simplification ordering parameterized by a symbol precedence. While the practical experiments described in Section 5 use the ATP Vampire [8], the model architecture does not assume a particular ATP and is compatible with any superposition-based ATP such as E [18], SPASS [22] or Vampire. Within the prover a particular saturation strategy is fixed including a time limit. If the prover runs out of time before solving a problem, we record $ast(P, \pi_P) = \perp$.

3 General considerations and overview

The aim of this work is to design a system that learns to suggest good symbol precedences to an ATP from observations of the ATP's performance on a class $\mathcal{P}_{train}$ of problems with randomly sampled precedences. Given a problem $P = (\Sigma, Cl)$ with $|\,\Sigma\,| = n$, we consider a precedence π_P good, if it leads to a low $ast(P, \pi_P)$ among the $n!$ possible precedences for P. Note that for problems with a signature with more than a few symbols, repeatedly running the prover with random precedences represents an effectively infinite source of training data.

Ideally, we would like to learn general theorem proving knowledge, not too dependent on $\mathcal{P}_{train}$, which could be later explained and compared to precedence generation schemes manually designed by the prover developers. Let us quickly recall one such scheme, already mentioned in the introduction, called `invfreq` in E [18]. The prover's manual [17] explains:

> Sort symbols by frequency (frequently occurring symbols are smaller).

What is common to basically all manually designed schemes, is that they pick a certain scalar property of symbols (here it is the symbol frequency, i.e. the number of occurrences of the symbol in the given problem) and obtain a precedence by *sorting* the symbols using that property.

Decomposing We might want our system to also learn a certain property of symbols and use sorting to generate and suggest a precedence. However, it is not clear how to "extract" such property from the observed data, since we only have access to the target values for full precedences. Our idea for "decomposing" these values into pieces that somehow relate to individual symbols (and can thus be "transferred" across problems) is to take a detour using symbol pairs: we assume that the performance of the ATP on P given π_P, i.e. our measure $ast(P, \pi_P)$, can be predicted from a sum of individual contributions corresponding to facts of the form

$$\pi_P \text{ orders the symbol } s_i \text{ before the symbol } s_j.$$

This is in line with how a prover developer could reason about a precedence generating scheme: Even when it is not clear how good or bad a symbol is in absolute terms, one might have an intuition that a symbol from a certain class should preferably come before a symbol from another class in the precedence (e.g., symbols introduced during clausification should typically be smaller than others) and assign some weight to this piece of intuition.

In Section 4.2 we formalize this idea using the notion of *preference matrix* and show how, for each problem in isolation, such preference matrix can be obtained in the form of coefficients learned by linear regression.

Learning across problems Symbol preferences learned on a particular problem are inherently tied to that problem and do not immediately carry over to other problems. The main reason for this is that symbols themselves only appear in the context of a particular problem.[6] That is why we resort to representing symbols by their *features* (cf. Section 4.3.2) when aggregating the learned preferences across different problems. This is in more detail explained further below and, formally, in Section 4.3.

We also strive to ensure that the preference values across problems have possibly the same magnitude. Note that $ast(P, \pi_P)$ may vary a lot for a fixed problem P but all the more so across problems. To obtain commensurable values, we normalize (see Section 4.1) the prover performance data on a per problem basis before learning the preferences. Normalization also deals with supplying a concrete value to those runs which did not finish, i.e. have $ast(P, \pi_P) = \bot$.

"Second-order" regression Once the symbols are abstracted by their feature vectors, we can collect symbol preferences from all the tested problems and turn this collection into another regression task. Note that at this moment, the preferences, which were obtained as the coefficients learned by linear regression, themselves become the regression target. Thus, in a certain sense, we now do second-order learning. It should be stressed though, that while the learning of the preferences *requires* a linear regression model by design, this second-order regression does not need to be linear and more sophisticated models can be experimented with.

The details of this step are given in Section 4.3.

Preference prediction and optimization Once the second-order model has been learned, we can predict preferences for any pair of symbols based on their feature vectors and thus also predict, given a problem P, how many steps will a prover require to solve it using a particular precedence π_P. (For this second step, we reverse the idea of decomposition: we sum up those predicted preferences that correspond to pairs of symbols s_i, s_j such that π_P orders the symbol s_i before the symbol s_j – see Section 4.4 for details).

Having access to an estimate of performance for each precedence π_P, the final step is to look for a precedence π_P^* that ideally minimizes the predicted performance measure over all the $n!$ possible precedences on P's signature. Since finding the true optimum could be computationally hard, we resort to using an approximation algorithm by Cohen et al. [2].

The algorithm is recalled in Section 4.4.2.

4 Architecture

4.1 Values of precedences

We define the base cost value $cost_{base}(P, \pi_P)$ of precedence π_P on problem P according to the outcome of the proof search configured to use this precedence:

[6]On certain benchmarks, such as those coming from translations of mathematical libraries [5], symbols maintain identity and meaning across individual problems. However, since our goal in this work is to learn general theorem proving knowledge, we do not use the assumption of aligned signatures.

- If the proof terminates successfully, $\mathit{cost}_{\mathit{base}}(\pi_P)$ is the number of iterations of the saturation loop started during the proof search: $\mathit{cost}_{\mathit{base}}(\pi_P) = \mathit{ast}(P, \pi_P)$.
- If the proof search fails (meaning that $\mathit{ast}(P, \pi_P) = \perp$), then $\mathit{cost}_{\mathit{base}}(\pi_P)$ is the maximum number of saturation loop iterations encountered in successful training proof searches on this problem: $\mathit{cost}_{\mathit{base}}(\pi_P) = \max_{\pi'_P \in \Pi_P^+} \mathit{ast}(P, \pi'_P)$, where Π_P^+ is the set of all training precedences on problem P that yield a successful proof search.

We further normalize the cost values by the following operations:

1. Logarithmic scaling: For each solvable problem, running proof search with uniformly random predicate precedences reveals a distribution of abstract solving times on successful executions. Examining these distributions for various problems suggests that they are usually approximately log-normal. To make further scaling by standardization reasonable, we first transform the base costs by taking their logarithm.
2. Standardization: Independently for each problem, we apply an affine transformation so that the resulting cost values have the mean 0 and standard deviation 1. This ensures that the values are comparable across problems.

Let $\mathit{cost}_{\mathit{std}}(\pi_P)$ denote the resulting cost value of precedence π_P after the scaling and standardization.

4.2 Problem preference matrix learning

Given a problem P with n symbols, a *preference matrix* W_P is any matrix over $\mathbb{R}$ of size $n \times n$. We define the *proxy cost of precedence* π_P *under preference* W_P to be the sum of the preference values $W_{P_{i,j}}$ of all symbol pairs s_i, s_j ordered by π_P such that s_i comes before s_j:

$$\mathit{cost}_{\mathit{proxy}}(\pi_P, W_P) = \sum_{i,j} [\![\pi_P^{-1}(i) < \pi_P^{-1}(j)]\!] \, W_{P_{i,j}} = \overrightarrow{O(\pi_P)} \cdot \overrightarrow{W_P}$$

where $\overrightarrow{O(\pi_P)} \cdot \overrightarrow{W_P}$ is the dot product of the flattened matrices $O(\pi_P)$ and W_P.

For any given problem we can uniformly sample precedences π_P to form the training set $T = \{(\pi_P^1, \mathit{cost}_{\mathit{std}}(\pi_P^1)), (\pi_P^2, \mathit{cost}_{\mathit{std}}(\pi_P^2)), \ldots, (\pi_P^m, \mathit{cost}_{\mathit{std}}(\pi_P^m))\}$. Having such training set allows us to find a vector $\overrightarrow{W_P}$ that minimizes the mean square error

$$\frac{1}{m} \sum_{(\pi_P, \mathit{cost}_{\mathit{std}}(\pi_P)) \in T} (\mathit{cost}_{\mathit{proxy}}(\pi_P, W_P) - \mathit{cost}_{\mathit{std}}(\pi_P))^2$$

by linear regression.

Minimizing the mean square error directly may lead to overfitting to the training set, especially in problems whose signature is relatively large in comparison to the size of the training set. To improve generalization, we use the Lasso regression algorithm [20] instead of standard linear regression. We use cross-validation to set the value of the regularization hyperparameter.[7]

Another reason to use the Lasso algorithm is that it performs regularization by imposing a penalty on coefficients with large absolute value, effectively shrinking the coefficients that correspond to symbol pairs whose mutual order does not affect the $\mathit{cost}_{\mathit{std}}(\pi_P)$. We can use

[7] See the model `LassoCV` in the machine learning library scikit-learn [12].

this property to interpret the absolute value of preference value as a measure of the importance of a given symbol pair.

In the following sections we assume that the preference matrix W_P we find by Lasso regression yields $cost_{proxy}$ that approximates $cost_{std}$ well.

4.3 General preference matrix learning

We proceed to cast the task of finding a good preference matrix W_P for an arbitrary problem as a regression on feature vector representations of symbol pairs. To accomplish this we need to be able to represent each pair of symbols by a feature vector and to know target preference values for pairs of symbols in a training problem set.

4.3.1 Target preference values

For each problem P in the training problem set, we find a problem preference matrix by the method outlined in Section 4.2. The target value of an arbitrary pair of symbols s_i, s_j in P is $W_{P i,j}$.

4.3.2 Symbol pair embedding

We represent each symbol by a numeric *feature vector* that consists of the following components: symbol arity, the number of symbol occurrences in the problem, the number of clauses in the problem that contain at least one occurrence of the symbol, an indicator of occurrence in a conjecture clause, an indicator of occurrence in a unit clause, and an indicator of being introduced during clausification. This choice of symbol features is motivated by the fact that they are readily available in Vampire and that they suffice as a basis for common precedence generation schemes, such as the `invfreq` scheme. We denote the feature vector corresponding to symbol s as $fv(s)$.

We represent a pair of symbols s, t by the concatenation of their feature vectors $[fv(s), fv(t)]$.

4.3.3 Training data

The general preference regressor is trained on samples of the following structure:

- the input: $[fv(s_i), fv(s_j)]$ – the embedding of a symbol pair s_i, s_j in problem P,
- the target: $W_{P i,j}$ – an element of the preference matrix we learned for problem P corresponding to the symbol pair (s_i, s_j).

We sample problem P from the training problem set with uniform probability.

Thanks to how W_P is constructed (see Section 4.2), preference values close to 0 are associated with symbol pairs whose mutual order has little effect on the outcome of the proof search. To focus the training on the symbol pairs whose order does matter, we weight the samples by the absolute value of the target. More precisely, given a problem P, the probability of sampling the symbol pair i, j is proportional to the absolute target value $|W_{P i,j}|$. Experiments have shown that using sample weighting improves the performance of the resulting model (see Section 5.2).

We denote the trained model as M and its prediction of the preference value of the symbol pair s_i, s_j as $M([fv(s_i), fv(s_j)])$.

4.4 Precedence construction

When presented a new problem $P = (\Sigma, Cl)$, we propose a symbol precedence by taking the following steps:

1. Estimate a preference matrix $\widehat{W_P}$.
2. Construct a precedence $\widehat{\pi_P}$ that approximately minimizes $cost_{proxy}(\widehat{\pi_P}, \widehat{W_P})$.

4.4.1 Preference matrix construction

To construct a preference matrix $\widehat{W_P}$ for a new problem P, we evaluate the general preference regressor on the feature vectors of all symbol pairs in P. More specifically,

$$\widehat{W_P}_{i,j} = M([fv(s_i), fv(s_j)])$$

for all $s_i, s_j \in \Sigma$.

At this moment, one can use $\widehat{W_P}$ to estimate the cost of an arbitrary symbol precedence.

4.4.2 Precedence construction from preference matrix

The remaining task is, given a preference matrix $\widehat{W_P}$, to find a precedence $\widehat{\pi_P}$ that minimizes $cost_{proxy}(\widehat{\pi_P}, \widehat{W_P})$. Since this task is NP-hard in general [2], we rely in this work on a greedy 2-approximation algorithm proposed by Cohen et al. [2]. The rest of this section provides a brief description of the algorithm.

The algorithm maintains a partially constructed symbol precedence $\mathbf{p} \in \mathbb{N}^*$ (a finite sequence over $\mathbb{N}$; initially empty), a set of available symbols $\Sigma_{avail} \subseteq \Sigma$ (initially the whole Σ) and a *potential value* for each of the symbols $c : \Sigma_{avail} \to \mathbb{R}$. The potential value of a symbol corresponds to the relative increase in proxy cost associated with selecting the symbol as the next to append to the partial precedence:

$$c(s_i) = \sum_{s_j \in \Sigma_{avail}} \widehat{W_P}_{i,j} - \sum_{s_j \in \Sigma_{avail}} \widehat{W_P}_{j,i}$$

In each iteration, a symbol s_i with the smallest potential is selected from Σ_{avail}. This symbol is removed from Σ_{avail} and its index i is appended to the partial precedence $\mathbf{p}$. The potentials of the remaining symbols in Σ_{avail} are updated. This process is repeated until all symbols have been selected, yielding the final $\mathbf{p}$ as $\widehat{\pi_P}$.

5 Evaluation

5.1 Setup

Since the simplification orderings under consideration (LPO and KBO) never use the symbol precedence to compare a predicate symbol with a function symbol, we can break down the symbol precedence into a predicate precedence and a function precedence. In this paper, we restrict our attention to predicate precedences, leaving function symbols to be ordered by the `invfreq` scheme. A more thorough evaluation of both predicate and function precedences and their interaction is left for future work.

We use problems from the TPTP library v7.2.0 [19] for the evaluation. Let $\mathcal{P}_{train}$ be the set of all FOL and CNF problems in TPTP with at most 200 predicate symbols such that at least 1 out of 24 random predicate precedences leads to a successful proof search ($|\mathcal{P}_{train}| = 8217$). Let $\mathcal{P}_{test}$ be the set of all FOL and CNF problems in TPTP with at most 1024 predicate symbols ($|\mathcal{P}_{test}| = 15751$). In each of 5 evaluation iterations (splits), we sample 1000 training problems from $\mathcal{P}_{train}$ and 1000 test problems from $\mathcal{P}_{test}$ uniformly in a way that ensures that the sets do not overlap. We repeat the evaluation 5 times to evaluate the stability of the training.

On each training problem we run Vampire with 100 uniformly random predicate precedences and a strategy fixed up to the predicate precedence.[8] We limit the time to 10 seconds per execution which is in our experience with Vampire sufficient to exhibit interesting behavior. Note that we use a customized version of Vampire to extract a symbol table from each of the problems.[9]

After we fit a preference matrix on each of the training problems (see Section 4.2), we create a batch of 10^6 symbol pair feature vectors with target values to train the general preference regressor (see Section 4.3). We evaluate the trained model by running Vampire on the test problem set with predicate precedences proposed by the trained model, counting the number of successfully solved problems.

A collection of scripts created for the experimental evaluation can be found in the Git repository at `https://github.com/filipbartek/vampire-ml/tree/75c693f3`. The script map-reduce/paar2020/run.sh can be used to perform the measurements presented below.

5.2 Experimental results

We trained two types of general preference regressors (see Section 4.3):

- Elastic-Net – a linear regression model with L1 and L2 norm regularization; see `ElasticNetCV` in Pedregosa et al. [12]
- Gradient Boosting regressor – see `GradientBoostingRegressor` in Pedregosa et al. [12]

We compared the performance of the regressors with three baseline precedence generation schemes – random precedence, best of 10 random precedences and the `invfreq` scheme. Table 1 shows the results of evaluation on 1000 problems for 5 random choices of training and test problem set (splits).

Case	Successes out of 1000 per split						
	0	1	2	3	4	Mean	Std
Best of 10 random	514	499	509	511	476	501.8	13.85
`invfreq`	494	472	480	481	452	475.8	13.83
Elastic-Net	484	471	479	470	454	471.6	10.21
Gradient Boosting	473	462	475	475	439	464.8	13.78
Elastic-Net without sample weighting	475	454	465	459	453	461.2	8.11
Random	454	455	457	456	430	450.4	10.25

Table 1: Experiment results

The case "Elastic-Net without sample weighting" shows the effect of sampling the symbol pairs uniformly. Inspection of the trained feature coefficients reveals that the fitting ends up

[8]Time limit: 10 seconds, memory limit: 8192 MB, literal comparison mode: predicate, function symbol precedence: `invfreq`, saturation algorithm: discount, age-weight ratio: 1:10, AVATAR: disabled.

[9]`https://github.com/filipbartek/vampire/tree/926154f2`

with an all-zero feature weight vector on splits 1 and 2, signifying a complete failure to learn on these training sets.

Using Elastic-Net for general preference prediction on average nearly matches the performance of Vampire with the `invfreq` precedence scheme. While Elastic-Net performs significantly better than a random precedence generator, it still performs significantly worse than a generator that, given a problem, tries 10 random precedences and chooses the best of these. This suggests that there is space for improvement, possibly with a more sophisticated, non-linear model. Plugging in a Gradient Boosting regressor does not show immediate improvement so more elaborate feature extraction may be necessary.

5.3 Feature coefficients

Since Elastic-Net is a linear regression model, we can easily inspect the coefficients it assigns to the input features (see Section 4.3.2). In each of the five splits, the final coefficients of the three indicator features (namely the indicators of presence in a conjecture clause, presence in a unit clause and being introduced during clausification) are 0. Table 2 shows the fitted non-zero coefficients of the remaining features. The coefficients were scaled so that their absolute values sum up to 1. Note that scaling the coefficients by a constant does not affect the precedence constructed using the greedy algorithm presented in Section 4.4.2.

Training set	Left symbol			Right symbol		
	Arity	Frequency	Unit frequency	Arity	Frequency	Unit frequency
0		−0.01		−0.98	0.01	
1		−0.48			0.08	0.44
2			−0.64		0.36	
3	0.88	−0.03	0.01		−0.03	0.05
4		−0.62			0.30	0.07
$\mathcal{P}_{train}$			−0.57		0.43	

Table 2: Elastic-Net feature coefficients after fitting on each of the 5 training sets of 1000 problems and on the whole $\mathcal{P}_{train}$. “Frequency” is the number of occurrences of the symbol in the problem. “Unit frequency” is the number of clauses in the problem that contain at least one occurrence of the symbol.

It is worth pointing out that the regressor fitted on the whole $\mathcal{P}_{train}$ and on the training sets 1, 2 and 4 assigns a high preference value to symbol pairs (s, t) such that t has a higher frequency and unit frequency than s. Since unit frequency is positively correlated with frequency, minimizing $cost_{proxy}$ using this fitted regressor is consistent with the `invfreq` precedence generating scheme (ordering the symbols by frequency in descending order). Similarly, the model fitted on training sets 0 and 3 corresponds to ordering the symbols by arity in ascending order.

6 Conclusion

This paper is, to the best of our knowledge, a first attempt to use machine learning for proposing symbol precedences for an ATP. This appears to be a potentially highly rewarding task with an access to effectively unlimited amount of training data generated on demand. Nevertheless, the journey from evaluating the prover on random precedences to proposing a good precedence when presented with a new problem is not straightforward and several conceptual gaps need to be bridged to connect these two tasks algorithmically.

In this paper, we proposed a connection using the concept of pairwise symbol preferences that, as we have shown, can be learned as the coefficients of a linear regression model for which an order matrix provides the features of a precedence understood as a permutation. In a second stage, in which symbols are abstracted by their features, the preferences themselves become regression targets.

In our initial experiments reported in this paper, the performance of our system does not yet reach that of the human-designed heuristic `invfreq`. We believe, however, that further improvements are possible by using a more advanced regression model for the second stage and/or by further hyper-parameter tuning (e.g. of the Gradient Boosting model). Ultimately, we expect to gain the most by using a richer set of symbol features, ideally automatically extracted from the problems using graph neural networks [23].

References

[1] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994. doi: 10.1093/logcom/4.3.217. URL https://doi.org/10.1093/logcom/4.3.217.

[2] William W. Cohen, Robert E. Schapire, and Yoram Singer. Learning to order things. *CoRR*, abs/1105.5464, 2011. URL http://arxiv.org/abs/1105.5464.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[4] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., USA, 1962. ISBN 0471430145.

[5] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55 (3):245–256, 2015. doi: 10.1007/s10817-015-9330-8. URL http://dx.doi.org/10.1007/s10817-015-9330-8.

[6] Samuel N. Kamin and Jacques Lévy. Two generalizations of the recursive path ordering. 1980.

[7] D. E. Knuth and P. B. Bendix. *Simple Word Problems in Universal Algebras*, pages 342–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_23. URL https://doi.org/10.1007/978-3-642-81955-1_23.

[8] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8. URL http://www.vprover.org/cav2013.pdf.

[9] Laura Kovács, Georg Moser, and Andrei Voronkov. On transfinite knuth-bendix orders. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2011. ISBN 978-3-642-22437-9. doi: 10.1007/978-3-642-22438-6_29. URL https://doi.org/10.1007/978-3-642-22438-6_29.

[10] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. Elsevier and MIT Press, 2001. doi: 10.1016/b978-044450813-3/50009-6. URL https://doi.org/10.1016/b978-044450813-3/50009-6.

[11] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 335–367. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. doi: 10.1016/b978-044450813-3/50008-4. URL https://doi.org/10.1016/b978-044450813-3/50008-4.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] Giles Reger and Martin Suda. Measuring progress to predict success: Can a good proof strategy be evolved? In *AITP 2017*, 2017.

[14] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016. doi: 10.29007/dzfz. URL https://easychair.org/publications/paper/XncX.

[15] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003. doi: 10.1016/S0747-7171(03)00040-3. URL https://doi.org/10.1016/S0747-7171(03)00040-3.

[16] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 298–313, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_19. URL https://doi.org/10.1007/978-3-642-81955-1_19.

[17] Stephan Schulz. E 2.4 User Manual. http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.4/eprover.pdf (accessed May 2020), 2019.

[18] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pacal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

[19] 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.

[20] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178.

[21] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_28. URL https://doi.org/10.1007/978-3-642-81955-1_28.

[22] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009. ISBN 978-3-642-02958-5. doi: 10.1007/978-3-642-02959-2_10. URL https://doi.org/10.1007/978-3-642-02959-2_10.

[23] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks, 2019.