Learning Rules Explaining Interactive Theorem Proving Tactic Prediction

Liao Zhang[1 [0000−0002−4574−8843]], David M. Cerna[2 [0000−0002−6352−603X]], and Cezary Kaliszyk[3,1 [0000−0002−8273−6059]]

[1] University of Innsbruck, Innsbruck, Austria
Liao.Zhang@student.uibk.ac.at
[2] Czech Academy of Sciences, Prague, Czechia
dcerna@cas.cs.cz
[3] University of Melbourne
cezary.kaliszyk@unimelb.edu.au

Abstract. Formally verifying the correctness of mathematical proofs is more accessible than ever, however, the learning curve remains steep for many of the state-of-the-art interactive theorem provers (ITP). Deriving the most appropriate subsequent proof step, and reasoning about it, given the multitude of possibilities, remains a daunting task for novice users. To improve the situation, several investigations have developed machine learning based guidance for *tactic* selection. Such approaches struggle to learn non-trivial relationships between the chosen tactic and the structure of the proof state and represent them as symbolic expressions.

To address these issues we (i) We represent the problem as an *Inductive Logic Programming (ILP)* task, (ii) Using the ILP representation we enriched the feature space by encoding additional, computationally expensive properties as *background knowledge* predicates, (iii) We use this enriched feature space to learn rules explaining when a tactic is applicable to a given proof state, (iv) We use the learned rules to filter the output of an existing tactic selection approach and empirically show improvement over the non-filtering approaches.

Keywords: Inductive logic programming · Interactive theorem proving.

1 Introduction

Interactive Theorem Provers (ITP), such as Coq [27], Lean [20], and Isabelle [22], are powerful tools that combine human instruction with computer verification to construct formal mathematical proofs, providing a reliable means of certification and ensuring safety in critical applications.

These systems operate as follows: the user specifies a goal to prove, *the initial proof state*. Then the user specifies *tactics* (an operation transforming a proof state into proof states). Certain tactics close proof states. The proof is complete if there are no remaining open proof states, i.e., the goal has been proved.

Given the complexity of ITP systems, a fully automated approach to proving user specified goals is intractable. Numerous investigations have instead focused on providing the user with guidance through tactic suggestion.

The methods used in practice by ITP users are statistical machine learning methods such as k-nearest neighbors (k-NN) and naive Bayes [9]. These methods take a goal g, select a goal g' most similar goal to g, and rank the particular tactics relevant for solving g' based on their likelihood of solving g.

Neural network and LLM-based approaches addressing the task include: CoqGym [29] trains tree neural networks to automatically construct proofs for *Coq*. Thor [14] combines LLMs and external symbolic solvers to search for proofs for Isabelle. LLMs are also applied to synthesising training data to enhance the performance of theorem proving [28]. Despite showing slight improvement in performance during machine learning evaluations, in practice these methods require long training for each new theory, which makes them less useful for day to day proof development.

Additionally, they lack interpretability. When a user receives predictions, they may want to know why a particular tactic was chosen over another tactic to better understand what actions they should take in the future.

Furthermore, guidance based on statistical learning approaches often requires propositionalisation of features, calculated based on the structure of the *abstract syntax tree (AST)* of a proof state [31], e.g., *there is a path between nodes X and Y in tree T*. For complex and precise features, pre-computation is prohibitively expensive.

Moreover, logical inference is significantly influenced by the small error margins present in the statistical inferencing mechanisms of LLMs and similar models. Thus, predictions based on chained logical inferences will quickly suffer a loss of predicative accuracy [17].

In contrast to pre-computed features, we represent such features as logic programs and compute them only when needed for learning. For example, we define logic programs for the existence of two particular nodes on a path (of arbitrary length) from the root of the tree as $(above(AST, X, Y))$. Below, we present a learned rule for the simplification tactic which states that the tactic is applicable to a proof state when the goal node of the proof state contains a constant above two constructs (also in the goal) which differ.

```
tac(A,"simpl") :-
  goal_node(const,A,B,C), goal_node(construct,A,D,E),
  goal_above(A,B,D), goal_node(construct,A,F,E),dif(F,D),
  goal_above(A,B,F).
```

The rules, as presented above, are learned using inductive logic programming (ILP), in particular, *Aleph* [26]. In addition to providing rules explaining tactic prediction, we use the resulting rules to filter the output of k-NN, in particular, the classifier presented in [3,9] (Tactician and TacticToe). Essentially, we want to determine whether $ps, r \vDash p_t$ where ps is a logic program representing the proof state, r is a learned rule for the tactic t, and p_t is the head predicate of

r denoting that t should be applied to ps. Thus, given the list of recommended tactics by a k-NN classifier, we can further filter this list using the learned rules. Our hypothesis is that features of proof state defined through logic programs can be used to learn rules which can be used to filter the output of a k-NN model to improve accuracy.

In addition to improved performance, our approach produces rules to explain the predictions. Consider again the aforementioned rule of `simpl` that specifies that the goal may be simplified if it contains a constant above two constructors with different positions. Here, the constructor and the constant denote the datatypes of Coq's terms. The same variable E confirms that the two constructors must correspond to the same identifier in Coq. This rule may suit the Coq structure $S\ x - S\ y$ which denotes $(1 + x) - (1 + y)$. It can be simplified to $x - y$. S denotes a constructor, and $-$ denotes a constant. The first argument of `goal_node` is a constant that is constrained by us via mode declarations [26].

We use the ILP system Aleph [26] together with a user-defined cost function to evaluate the learned rules on the Coq standard library. We chose Aleph because it has empirically good results [5]. We refrain from using modern ILP approaches such as *Popper* [6] as the underlying ASP solvers have difficulty generating models when many variables are required and high-arity definitions are included in the background. We develop representation predicates (`goal_node`) to efficiently denote the nodes of the AST. We also develop feature predicates (`goal_above`) which denote the properties of the AST calculated based on the representation predicates. The motivation for developing feature predicates is that propositionalization of it would significantly enlarge the representation making it impractical to use. Our experiments confirm that feature predicates can learn more precise rules (rules with higher F-1 scores [25]) compared to representation predicates. Additionally, the experiments demonstrate that the combination of ILP and k-NN can improve the accuracy of tactic suggestions in Tactician, the main tactic prediction system for Coq.

Our contributions can be summarized as follows:

- We express the task of predicting the best tactic to apply to the given proof state as an ILP task.
- Using the ILP representation we enriched the feature space by encoding additional, computationally expensive features as *background knowledge* predicates, allowing us to avoid grounding the features which are computationally expensive.
- We use this enriched feature space to learn rules explaining when a tactic is applicable to a given proof state, and filter the output of an existing tactic selection approaches using these rules.
- Finally, we empirically show improvement over the non-filtering approaches.

This is the first time an investigation has considered ILP as a tool for improving tactic suggestion methods for ITPs.

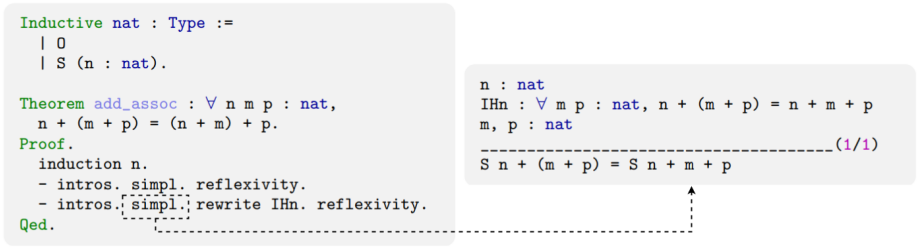

Fig. 1. A Coq proof of the associative property of addition and the proof state before `simpl`.

2 Background

2.1 Theorem Proving in Coq

Coq is one of the most popular proof assistants and has been widely used for building trustworthy software [18] and verifying the correctness of mathematical proofs [10]. Coq tactics are proof state transformations that provide a high-level combination of underlying logical inferences.

To illustrate how theorems are formalized in Coq, we present a simple example in Figure 1. Here, we want to prove the associative property of addition. The natural numbers in Coq are defined by two constructors `O` and `S`. `O` denotes 0, and `S n` denotes $n + 1$. Here, the initial proof state is the same as the statement of the theorem. We first apply induction on `n` and obtain two cases corresponding to the two constructors. In the first case, `n` equals 0. After some simplifications, we can prove $0 = 0$ by the tactic `reflexivity`. The second case is a bit more complicated, and we need to apply the induction hypothesis `IHn` to finish the proof. Figure 1 also presents a concrete example of a proof state. A proof state consists of a *goal* to prove and several *hypotheses*. The goal is below the dashed line. `IHn`, `n`, `m`, and `p` are the names of hypotheses. A proof state is often represented as a sequent $E \vdash g$ where E and g denote the hypotheses and the goal, respectively.

2.2 k-NN adaptations to theorem proving

Several machine learning algorithms have been adapted to theorem proving tasks. In most cases, simpler algorithms adapted to formal reasoning tasks perform better than deep learning based methods in practice. For this reason, modified k-NN (explained in [2]) is the main algorithm for TacticToe and Tactician. Even if evaluations with deep learning or large tree-based classifiers have shown some theoretical improvements, the simpler algorithms training for the particular theories developed by users, give a larger practical advantage to the users. As such, we focus on the modified k-NN in this work. Standard k-NN starts by calculating the distance between a new proof state and all known proof states in

the database. The distance is measured by the similarity between the features of the proof states, usually using tree walks in the AST of the proof state [16]. The dependencies of such selected neighbours, with additional scaling by their distances, inclusion of the neighbours themselves, and further modifications exhibit commendable empirical performance [24], and are therefore the default algorithm both in Tactictoe and Tactician.

3 Background Knowledge

To utilize ILP, we need to appropriately define the background knowledge. We start this section by encoding the nodes of AST as representation predicates. Then we propose new feature predicates that will allow leveraging the power of ILP. We finally add predicate anonymization, already very useful in automated reasoning systems, to representation and feature predicates.

3.1 Representation Predicates

Every node in the AST of the proof state is converted to a fact. There are two categories of nodes: identifiers of existing objects and constructors of Coq's datatype. A node in the goal is converted to $goal\_node(name, nat, goal\_idx)$. The argument *name* refers to the value of the node. A unique natural number is assigned to every proof state to identify it. The argument $goal\_idx$ uses a sequence of natural numbers to specify the position of the node in the goal. A node in a hypothesis is converted to a fact $hyp\_node(name, nat, hyp\_name, hyp\_idx)$. Compared to a $goal\_idx$, a hyp_idx starts with the name of a hypothesis so that two hyp_idx from different hypotheses have different prefixes. The $goal\_node$ and hyp_node predicates are called *representation predicates* in this work.

3.2 Feature Predicates

We also define two categories of *feature predicates* which represent the properties of AST based on the representation predicates.

Positional Predicates represents the relative relationships between nodes' positions. The predicate `goal_left(Goal_idx1, Goal_idx2)` and `goal_above(Nat, Goal_idx1, Goal_idx2)` respectively checks whether the node is left (above) to another node in the goal. They are inspired by the horizontal features and vertical features used in previous works [4,31]. Similarly, we define `hyp_left(Hyp_idx1, Hyp_idx2)` and `hyp_above(Nat, Hyp_idx1, Hyp_idx2)`. Previous works have confirmed the usefulness of using the occurrence numbers of features in feature characterization, which inspires us to develop the predicate `dif(Goal_idx1, Goal_idx2)`. It denotes that the same node multiply occurs in different positions in the goal.

Equational Predicates check the equality between two terms. The predicate `eq_goal_term(Nat, Goal_idx1, Goal_idx2)` checks that the two subterms in the goal are the same. The root nodes of the two subterms are located in the positions `Goal_idx1` and `Goal_idx2`, respectively. It pertains to `reflexivity` which proves a goal of the equation if the equality holds after some normalization. Thus, it can prove $x = x$ and inspires us to develop `eq_goal_term`. The predicate `eq_goal_hyp_term(Nat, Goal_idx, Hyp_idx)` is inspired by a number of tactics that check the equality between the goal and the hypotheses, such as `assumption`, `apply`, and `auto`. For instance, `assumption` proves a goal if it equals a hypothesis. Assume a proof state $H_1 : Q\ x, H_2 : P\ x \to Q\ x \vdash Q\ x$ which can be proved by `assumption`. The predicate `eq_goal_hyp_term` checks the equality between the goal and $Q\ x$ in a hypothesis. The predicate `is_hyp_root(Nat, Hyp_idx)` ensures the node is the root of a hypothesis. Thus, it can show the equality only holds between the goal and H_1 instead of H_2. With a reason akin to that of `is_hyp_root`, we define `is_goal_root(Nat, Goal_idx)`. The equality between two terms in different hypotheses is checked by `eq_hyp_term(Nat, Hyp_idx1, Hyp_idx2)`. It is useful for tactics that can apply hypotheses several times, e.g., `auto`. Assume a proof state $H_1 : P\ x, H_2 : P\ x \to Q\ x \vdash Q\ x$. First, `auto` applies H_2 to the goal and changes the goal to $P\ x$. Then, it applies H_1 to prove the new goal. The description of the operation requires to show that H_1 equals to the premise of H_2.

3.3 Anonymous Predicates

We also substitute identifiers with more abstract descriptions to facilitate the generalization ability of ILP. The substitution is similar to that in ENIGMA anonymous [13]. The predicates that accept original nodes and abstract nodes as their first arguments are called *original predicates* and *anonymous predicates*, respectively. We substitute identifiers with their categories, consisting of inductive types, constants, constructors, and variables. Besides the abstract nodes, we also include the original nodes as arguments in `goal_node` and `hyp_node`. We need them because when checking the equality, we want to compare the original nodes. Afterward, the anonymous predicates of nodes change to $goal\_node(anonym\_name, nat, goal\_idx, origin\_name)$ and $hyp\_node(anonym\_name, nat, hyp\_name, hyp\_idx, origin\_name)$. Some basic identifiers are not substituted, which consist of `logic_false`, `logic_true`, `and`, `or`, `iff`, `not`, `eq`, `bool_true`, and `bool_false`. There are both logic and boolean values of true and false because Coq can represent objects in logic or programs. Concerning the constructors of Coq's datatypes, we only retain four important constructors: `rel`, `prod`, `lambda`, and `evar`.

4 Method

Figure 2 presents an overview of our learning framework. During the training, we first perform orthogonalization, a technique introduced in TacticToe, to clean

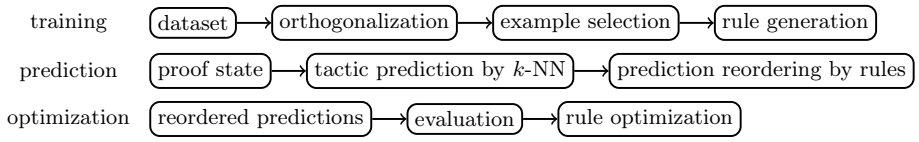

Fig. 2. An overview of the procedures of the learning framework.

the dataset. Then, we select examples and apply ILP to generate rules. To make predictions, first, k-NN predicts a sequence of likely helpful tactics. Afterward, the rules are used as a filter to reorder the predictions. The optimization procedure denotes removing some low-quality rules. This is achieved by evaluating the reordered predictions in the validation dataset and removing the low-quality ones. In the next subsections, we describe these parts.

4.1 Orthogonalization

In some cases, different tactics could transform the same proof state in the same way. This raises ambiguity and makes learning difficult. Orthogonalization is used to reduce such ambiguity. In the orthogonalization, we only focus on four very popular tactics in the Coq standard library: `assumption`, `reflexivity`, `trivial`, and `auto`. We denote the sets of proof states which can be closed by `assumption`, `reflexivity`, `trivial`, and `auto` as AS, R, T, and AT, respectively. There exist the relations $AS \subsetneq T$, $R \subsetneq T$, and $T \subsetneq AT$. For each proof state ps to which the tactic t is applied, the above four automation tactics are sequentially tried. If ps can be finished by the automation tactic t', we replace t by t'. If none of the four tactics can finish the proof state, the original t is preserved. The orthogonalization procedure is simpler than in TacticToe, which orthogonalizes all tactics. This is because our current predicates can only capture a part of the usage of tactics. We leave full orthogonalization as future work.

4.2 Example Selection

Choosing appropriate training examples is crucial for learning reasonable rules. For a specific tactic *tac*, the proof states to which it is applied are regarded as the positive examples. The proof states to which the tactics different from *tac* are applied are regarded as the negative examples. We experimentally determine the number of positive and negative examples for learning rules. We develop a clustering mechanism to split positive examples into roughly equal-sized clusters. We experimentally evaluate the combinations of different numbers of negative examples and different numbers of positive examples.

We choose an implementation of a constrained k-means algorithm [19] to split positive examples into clusters of roughly the same size. The original k-means algorithm [11] can only split examples into a certain number of clusters. In contrast, constrained k-means can also specify the lower bound and the upper

Algorithm 1 Preselection Reorder

Input: a sequence of tactics $tac_{1..50}$ preselected by k-NN for a proof state
Output: a sequence of tactics which is a reorder of the preselection
$goods \leftarrow [\,]$
$bads \leftarrow [\,]$
for all $i \in \{1..50\}$ **do**
  if tac_i is accepted by learned rules **then**
    append tac_i to the end of $goods$
  else
    append tac_i to the end of $bads$
  end if
end for
$reorder \leftarrow$ the sequence of appending $bads$ to the end of $goods$
return $reorder$

bound of the size of the clusters, which is important to give good sizes of training examples for each ILP learning task.

We apply k-NN to discover negative examples for each positive example. As this pre-processing step is not theorem-proving specific, we use the general k-NN from the scikit-learn library [23]. We use the same features as Tactician [31]. For each positive example, k-NN calculates the distance between it and every negative example in the training data. Then, we rank the negative examples in an ascending order of distance.

4.3 Training and Prediction

For each tactic, we use Aleph to generate ILP rules for each cluster of positive examples and its associated negative examples. Afterwards, all the rules are merged together, and duplicated rules are removed. Finally, we remove the rules of tactics that are logically subsumed by other rules of the same tactic.

Algorithm 1 illustrates the procedures of making predictions. We use the state-of-the-art k-NN in Tactician. The features are the same as those used in Section 4.2. Assume a pair of a proof state and a tactic (ps, tac). To make predictions, first, we use k-NN to preselect a sequence of likely tactics $tac_{1..50}$. For each tac_i, we use the learned rules to determine whether to accept it. During the evaluation, the prediction tac_i is expected (unexpected) if tac_i is equal (unequal) to tac. If the rules accept (reject) a tactic, the prediction is a declared positive (negative). If the rules reject a tac_i equal to tac, we regard the prediction made by the rules as a false negative (FN). Based on the expected tactics and acceptances, we also obtain true positives (TPs), true negatives (TNs), and false positives (FPs).

4.4 Rule Optimization

The idea of rule optimization is to remove some low-quality rules to increase the overall performance of rules. For the evaluation of all rules, we chose the F-1

score as the metric, defined as $\frac{2TP}{2TP+FP+FN}$, because it is a standard metric for evaluating imbalanced data. As an illustration of the imbalance, given a pair of a proof state and a tactic *tac*, rules make predictions for 50 preselected tactics. However, at most one is the same as *tac*. If a rule is overly general, which means that the number of FPs introduced by it is much larger than the number of TPs introduced by it, removing it will increase the overall F-1 score.

Although a large number of negative examples prevents generating overly general rules, using them may not produce the best rules for two reasons. First, our background can merely capture a portion of the usage of the tactics; thus, a significantly large number of negative examples cannot produce perfect rules but may produce overly specific rules. Second, some negative examples in our dataset are actually false negatives. A mathematician may be able to choose the next step from a couple of tactics that make different proof transformations. Orthogonalization in Section 4.1 can only partially remove such overlaps between tactics, thereby decreasing the number of false negatives. It is computationally prohibitive to perform full orthogonalization of our data. Observe that, our experiments still show an increase in accuracy in light of the noisy data.

Our approach allows us to learn many rules explaining a particular tactic. Over the training set, some of these rules capture the usage of the given tactic better than others. Before, moving to testing on unseen data, we prune the learn rules and keep only those that performed well on the validation set.

To determine which rules to include, we evaluate the quality of each rule in the validation dataset and remove those with low qualities. The left rules are used for the evaluation on the test dataset. We measure the quality of each rule and remove it if its quality is below a certain threshold. We set different thresholds and choose the threshold leading to the highest F-1 score via experiments. For the metric of the quality of a single rule, we use *precision*, defined as $\frac{TP}{TP+FP}$. Here, FP and TP are produced by a single rule. Precision is a good metric because if a rule is too general, its precision will be low and we will be able to remove it to improve the overall F-1 score.

5 Experiments

We conducted the experiments on the Coq standard library, a standard dataset for evaluating machine learning for Coq [7]. It consists of 41 theories and 151,678 proof states in total. The code for this paper is available at https://github.com/Zhang-Liao/ilp_coq.

Most parameters of Aleph were left as default besides three parameters. We set the maximal length of a clause to 1,000, the upper bound of proof depth to 1,000, and the largest number of nodes to be explored during the search to 30,000. We define a cost function similar to the default cost function because, by default, Aleph cannot learn with no negative examples or only one positive example. The user-defined cost function was only used when there were no negative examples or exactly one positive example. We set a timeout of ten minutes for learning.

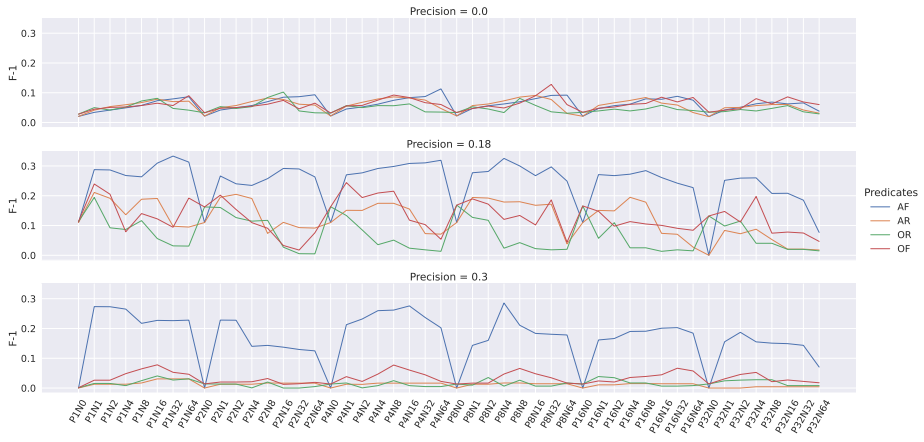

Fig. 3. F-1 scores of different parameters when *qualt* is set to 0, 0.18, or 0.30. *AF*, *AR*, *OF*, and *OR* denote the anonymous feature predicates, the anonymous representation predicates, the original feature predicates, and the original representation predicates, respectively. In the x-axis caption P and N denote *pos* and *neg*, respectively.

We conducted the experiments in the transfer-theory setting, which means different Coq theories are used for training, validation, and testing. We use this setting because it simulates a practical application scenario of ILP. Mathematicians develop new theories based on the definitions and proven theorems in the developed theories. To be practically beneficial, ILP should also learn rules from training theories, and the learned rules should help make tactic suggestions for theories that do not depend on the training theories.

The training theory should be carefully chosen before conducting experiments. The theory `Structures` was chosen for training because it has a balanced distribution of various tactics.

To be consistent with the transfer-theory setting, the testing theories should not depend on `Structures`. From the Coq standard library, we chose all theories which do not depend on `Structures` for testing including `rtauto`, `FSets`, `Wellfounded`, `funind`, `btauto`, `nsatz`, and `MSets`. Afterward, from all the theories that do not depend on the testing theories, we randomly chose five theories: `PArith`, `Relations`, `Bool`, `Logic`, and `Lists`, merged as the validation dataset.

5.1 Parameter Optimization

In Section 4, we introduced three additional hyper-parameters beyond those already present in Aleph. They are the size of the cluster of positive examples (*pos*), the number of negative examples of each positive example (*neg*), and the quality-theshold (*qualt*) below which the rule should be removed.

We evaluated the F-1 scores of different predicate categories with different parameters. There are four predicate categories AF, AR, OF, and OR, respec-

tively denoting the anonymous feature predicates, the anonymous representation predicates, the original feature predicates, and the original representation predicates. AF and OF contain both representation predicates and feature predicates, while AF and AR only contain anonymous predicates. We chose pos between 0 and 32. For neg, we chose it between 0 and 64. For all the combinations of pos and neg, rules were generated. Afterward, the learned rules were evaluated in the validation dataset. Finally, we calculated the F-1 scores with different values of $qualt$. The range of $qualt$ was set between 0 and 0.30, with intervals of 0.6.

Figure 3 depicts the F-1 scores when $qualt = \{0, 0.18, 0.30\}$. The significance of $qualt$ is evident. When $qualt = 0$, the best F-1 scores of all predicate categories hover around 0.10. The best scores become significantly higher than 0.10 when $qualt = 0.18$. The low F-1 scores of $qualt = 0$ are caused by some overly general rules which are discussed in Section 4.4. An example of such an overly general rule is provided below, showing the necessity of employing an appropriate $qualt$.

```
tac(A,"reflexivity") :- goal_node(coq_Init_Logic_eq,A,B,C).
```

The above rule denotes that `reflexivity` is appropriate whenever there is an equal sign in the goal. It is too general and irrelevant to the usage of `reflexivity` as explained in Section 3.2. With $qualt = 0.30$, the best F-1 scores decrease again. The decline is attributed to the fact that most rules remained by a very high $qualt$ are excessively specific, thereby producing a limited number of TPs.

Afterward, we analyze the results obtained with $qualt = 0.18$ since the F-1 scores are notably higher than those with $qualt = \{0, 0.30\}$. None of the predicate categories obtains the highest F-1 score with $pos = 32$. We assume the reason is that an overly large pos may gather many irrelevant positive examples, which causes difficulties in choosing negative examples. If two positive examples significantly differ, an appropriate negative example for one of them may be inappropriate for the other. A large value of neg generally decreases the F-1 scores of all predicates except for AF. A possible explanation is that too many negative examples cause AR, OF, and OR to learn overly specific rules. Due to the expressivity of AF, it can still learn some reasonable rules.

Table 1 displays the optimal parameters of all predicate categories. The generalization does not work well for OF and OR, but already with AF its F-1 score peak necessitates a large neg, indicating its superior ability to distinguish positive examples from negative examples and to learn precise rules. Perhaps due to the reason that our background knowledge is incapable of perfectly capturing the usage of tactics, AF also uses $pos = 1$. A small pos allows AF to learn many rules for diverse situations. AR requires $pos = 16$ to achieve its peak F-1 score, possibly due to its limitation of representing AST in a highly generalized manner.

5.2 Testing

According to parameter optimization, we choose the rules with the best parameter and test the performance in the test dataset.

Table 1. The best parameters of each predicate category.

Parameter	AF	AR	OF	OR
Precision	0.18	0.12	0.18	0.12
Positive	1	16	4	1
Negative	32	1	1	1

Table 2. The F-1 scores in the test dataset.

Theory	AF	AR	OF	OR
rtauto	**0.564**	0.401	0.502	0.440
FSets	**0.266**	0.125	0.193	0.144
Wellfounded	**0.229**	0.049	0.134	0.135
funind	**0.545**	0.0	0.0	0.0
btauto	**0.339**	0.125	0.162	0.122
nsatz	**0.164**	0.070	0.163	0.116
MSets	**0.272**	0.084	0.143	0.095

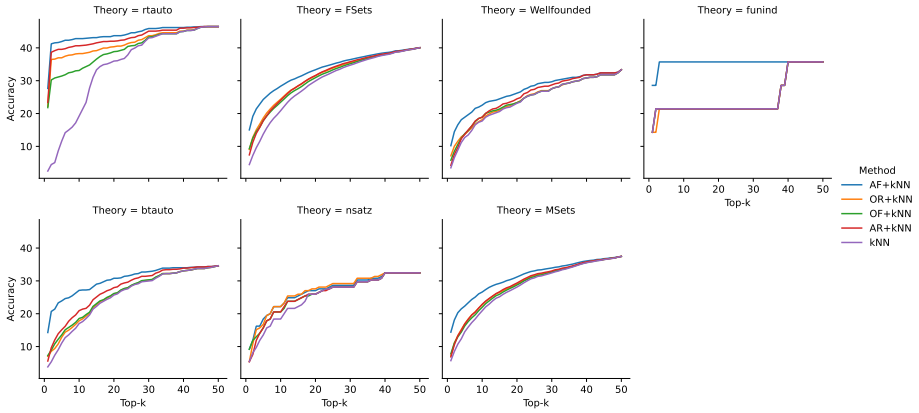

Fig. 4. Top-k accuracies in the test theories. It denotes how often the label is predicted in the first k predictions. The symbol $+$ denotes using the rules learned by a certain predicate category to reorder the preselections.

Table 2 shows the F-1 scores in the test dataset. Using a background knowledge consisting of AF predicate definitions during training results in rules which perform best during testing. This owes to that AF can learn precise rules to characterize the usage of tactics. In comparison, the rules learned by AR are too general, and the rules learned by OF and OR are too specific. In all theories, except those consisting of only a few proof states (`funind` has only 14 proof states), training with OF, OR, and AR background knowledge results in rules that perform well on the test data F-1 scores.

We also evaluated whether the combination of ILP and k-NN can improve the accuracy of k-NN. The algorithm of reordering is explained in Section 4.3. Figure 4 shows the results of the top-k accuracies in different theories. In all the theories, the combination of ILP and k-NN increases the accuracies of k-NN.

6 Case Studies and Limitations

To illustrate that we indeed learn precise rules, besides the example of `simpl` presented in Section 1, we present three more examples in this section. The rule of `trivial` suits the goal $A \to B = B$. First, `trivial` introduces A as a hypothesis, changing the proof state to $H : A \vdash B = B$. Next, `trivial` can automatically prove $B = B$. The rule of `auto` aligns the proof state of the structure $H : B \vdash A \vee B$. The tactic `auto` decomposes the disjunction, and the goal changes to either proving A or proving B. Then, it proves B with the hypothesis. In contrast, `trivial` cannot decompose the disjunction. The rule of `intuition` suits the goal $A \leftrightarrow A$ which cannot be proved by `auto`. In comparison, `intuition` can perform stronger automation than `auto` and can prove it.

```
tac(A,"trivial") :-
  goal_node(prod,A,B,C),goal_node(const,A,D,E), goal_above(A,B,D),
  goal_node(const,A,F,E),goal_above(A,B,F),eq_goal_term(A,F,D).
tac(A,"auto") :-
  goal_node(coq_Init_Logic_or,A,B,C),goal_node(const,A,D,E),
  goal_above(A,B,D),hyp_node(const,A,F,G,E),eq_goal_hyp_term(A,D,G).
tac(A,"intuition") :-
  goal_node(coq_Init_Logic_iff,A,B,C),goal_node(const,A,D,E),
  goal_above(A,B,D),goal_node(const,A,F,E),eq_goal_term(A,F,D)
```

Albeit we can learn several reasonable rules, many tactics are difficult to describe. There are several reasons for the difficulties. First, our current work cannot generalize tactics with different arguments. For instance, assume there are two tactics `apply H1` and `apply H2` where `H1` and `H2` are names of hypotheses. They are regarded as different tactics but may have the same behavior. Second, the usage of some tactics such as `induction` is inherently complicated [21]. Third, the same mathematical theorem can proved in various ways which leads to many overlaps between the usage of tactics.

7 Related Work

There are several tasks of machine learning for theorem proving. *Premise selection* is probably the most well-discovered task. It studies the question of how to predict possibly useful lemmas for a given theorem. Quite a lot of classical learning methods [1,8] and neural networks [12] have been applied to premise selection. The most relevant task to our work is learning-based formal theory proving. Researchers have investigated both employing machine learning to learn from human-written proofs [9] and guide some sophisticated software to automatically construct proofs [15].

8 Conclusion and Future Work

We have developed the first application of ILP to interactive theorem proving. For this, we have developed new feature predicates, able to dynamically calculate

features based on the representation of AST of the proof state. We proposed a method for using ILP effectively for tactic prediction. We experimentally evaluated the rules learned by ILP and compared them to practically used prediction mechanisms in ITPs. The experiments confirm that the method gives explainable tactic predictions. Our work shows the potential of applications of ILP to improve ITP tactic suggestion methods.

Several improvements are possible. We would like to use our work with stronger ILP systems, such as Popper. However, given that our background knowledge includes predicates with high arity and our method builds large rules with many variables, the underlying ASP (SAT) solver used by Popper struggles with the generation of models. Improvements to our encoding and recent work on improving the performance of Popper can make this research direction viable in the near future.

Next, it is interesting to use ILP to capture the relations between arguments of tactics and the objects to which the arguments refer. Finally, we plan to investigate the application of ILP to other ITP tasks [30].

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. Journal of automated reasoning **52**, 191–213 (2014)
2. Blaauwbroek, L., Urban, J., Geuvers, H.: Tactic learning and proving for the coq proof assistant. arXiv preprint arXiv:2003.09140 (2020)
3. Blaauwbroek, L., Urban, J., Geuvers, H.: The tactician: A seamless, interactive tactic learner and prover for coq. In: CICM, July 26–31, 2020, Proceedings. pp. 271–277. Springer (2020)
4. Chvalovský, K., Jakubův, J., Suda, M., Urban, J.: Enigma-ng: efficient neural and gradient-boosted inference guidance for e. In: CADE 27: August 27–30, 2019, Proceedings 27. pp. 197–215. Springer (2019)
5. Cropper, A., Dumančić, S.: Inductive logic programming at 30: a new introduction. Journal of Artificial Intelligence Research **74**, 765–850 (2022)
6. Cropper, A., Morel, R.: Learning programs by learning from failures. Machine Learning **110**, 801–856 (2021)
7. Czajka, Ł., Kaliszyk, C.: Hammer for coq: Automation for dependent type theory. Journal of automated reasoning **61**, 423–453 (2018)
8. Gauthier, T., Kaliszyk, C.: Premise selection and external provers for hol4. In: CPP. pp. 49–57 (2015)
9. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: Tactictoe: learning to prove with tactics. Journal of Automated Reasoning **65**, 257–286 (2021)
10. Gonthier, G., et al.: Formal proof–the four-color theorem. Notices of the AMS **55**(11), 1382–1393 (2008)
11. Hartigan, J.A., Wong, M.A.: Algorithm as 136: A k-means clustering algorithm. Journal of the royal statistical society. series c (applied statistics) **28**(1), 100–108 (1979)
12. Irving, G., Szegedy, C., Alemi, A.A., Eén, N., Chollet, F., Urban, J.: Deepmath-deep sequence models for premise selection. NIPS **29** (2016)

13. Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., Urban, J.: Enigma anonymous: Symbol-independent inference guiding machine (system description). In: IJCAR. pp. 448–463. Springer (2020)
14. Jiang, A.Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygóźdź, T., Miłoś, P., Wu, Y., Jamnik, M.: Thor: Wielding hammers to integrate language models and automated theorem provers. NIPS **35**, 8360–8373 (2022)
15. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. Advances in Neural Information Processing Systems **31** (2018)
16. Kaliszyk, C., Urban, J., Vyskočil, J.: Efficient semantic features for automated reasoning over large theories. In: Yang, Q., Wooldridge, M. (eds.) IJCAI. pp. 3084–3090. AAAI Press (2015)
17. LeCun, Y.: Do large language models need sensory grounding for meaning and understanding. In: Workshop on Philosophy of Deep Learning (2023)
18. Leroy, X.: The CompCert C verified compiler: Documentation and user's manual. Ph.D. thesis, Inria (2021)
19. Levy-Kramer, J.: k-means-constrained (Apr 2018), https://github.com/joshlk/k-means-constrained
20. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Automated Deduction-CADE-25, August 1-7, 2015, Proceedings 25. pp. 378–388. Springer (2015)
21. Nagashima, Y.: Lifter: language to encode induction heuristics for isabelle/hol. In: Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, December 1–4, 2019, Proceedings 17. pp. 266–287. Springer (2019)
22. Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)
23. Pedregosa, F., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
24. Rute, J., et al.: Graph2tac: Learning hierarchical representations of math concepts in theorem proving. arXiv preprint arXiv:2401.02949 (2024)
25. Sasaki, Y., et al.: The truth of the f-measure. Teach tutor mater **1**(5), 1–5 (2007)
26. Srinivasan, A.: The aleph manual (2001)
27. The Coq Development Team: Coq reference manual 8.11.1 (2020), https://coq.github.io/doc/v8.11/refman/index.html
28. Xin, H., et al.: Lego-prover: Neural theorem proving with growing libraries. arXiv preprint arXiv:2310.00656 (2023)
29. Yang, K., Deng, J.: Learning to prove theorems via interacting with proof assistants. In: ICML. pp. 6984–6994. PMLR (2019)
30. Zhang, L., Blaauwbroek, L., Kaliszyk, C., Urban, J.: Learning proof transformations and its applications in interactive theorem proving. In: International Symposium on Frontiers of Combining Systems. pp. 236–254. Springer Nature Switzerland Cham (2023)
31. Zhang, L., Blaauwbroek, L., Piotrowski, B., Kaliszyk, C., Urban, J.: Online machine learning techniques for coq: A comparison. In: International Conference on Intelligent Computer Mathematics. pp. 67–83. Springer (2021)