PaMpeR: A Proof Method Recommendation System for Isabelle/HOL

Yutaka Nagashima[1,2] and Yilun He[3]

[1] CIIRC, CTU
[2] The University of Innsbruck
[3] The University of Sydney

Abstract. Deciding which sub-tool to use for a given proof state requires expertise specific to each ITP. To mitigate this problem, we present `PaMpeR`, a proof method recommendation system for Isabelle/HOL. Given a proof state, `PaMpeR` recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. `PaMpeR` generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that `PaMpeR` correctly predicts experienced users' proof methods invocation especially when it comes to special purpose proof methods.

1 Introduction

Do you know when to use the `induct_tac` method instead of the `induct` method in Isabelle/HOL [1]? Did you know the proof method called `intro_classes`? What about `uint_arith`? If you are an experienced Isabelle user, your answer is "*Sure.*" But if you are new to Isabelle, your answer might be "*No. Do I have to know these Isabelle specific details?*"

Modern ITPs are equipped with many sub-tools, such as proof methods and tactics. For example, Isabelle/HOL comes with dozens of proof methods. These sub-tools provide useful automation for interactive theorem proving; however, it still requires proof assistant specific expertise to pick up the right proof method to discharge a given proof goal.

This paper presents our novel approach to proof method recommendation and its implementation, `PaMpeR` (available on GitHub [3]). Our research hypothesis is that:

> *it is possible to advise which proof methods are useful to a given proof state, based only on the meta-information about the state and information in the standard library. Furthermore, we can extract advice by applying machine learning algorithms to existing large proof corpora.*

The paper is organized as follows: Section 2 explains the basics of Isabelle/HOL and provides the overview of `PaMpeR`. Section 3 expounds how `PaMpeR` transforms the complex data structures representing proof states to simple data structures that are easier to handle for machine learning algorithms. Section 4 shows how our machine learning algorithm constructs regression trees from these simple data structures. Section 5 demonstrates how users can elicit recommendations from `PaMpeR`. Section 6 presents our preliminary evaluation of `PaMpeR` to assess the accuracy of `PaMpeR`'s recommendations. Section 7 justifies our design decisions for `PaMpeR`. Section 8 discusses the strengths and limitations of the current implementation and the design of a proof search tool based on `PaMpeR`. Section 9 compares our work with other attempts of applying machine learning to interactive theorem proving.

2 Background and Overview of PaMpeR

2.1 Background

Isabelle/HOL is an interactive theorem prover, mostly written in Standard ML. The consistency of Isabelle/HOL is carefully protected by isolating its logical kernel using the module system of Standard ML. *Isabelle/Isar* [14] (*Isar* for short) is a proof language used in Isabelle/HOL. Isar provides a human-friendly interface to specify and discharge proof obligations. Isabelle users discharge proof obligations by applying *proof methods*, which are the Isar syntactic layer of LCF-style tactics.

Each proof obligation in Isabelle/HOL is stored within a *proof state*, which also contains locally bound theorems for proof methods (*chained facts*) and the background *proof context* of the proof obligation, which includes local assumptions, auxiliary definitions, and lemmas proved prior to the the current step. Proof methods are in general sensitive not only to proof obligations but also to their chained facts and background proof contexts: they behave differently based on information stored in proof state. Therefore, when users decide which proof method to apply to a proof obligation, they often have to take other information in the proof state into consideration.

Isabelle comes with many Isar keywords to define new types and constants, such as `datatype`, `codatatype`, `primrec`, `primcorec`, `inductive`, and `definition`. For example, the `fun` command is used for general recursive definitions.

These keywords not only let users define new types or constants, but they also automatically derive auxiliary lemmas relevant to the defined objects behind the user-interface and register them in the background proof context where each keyword is used. For example, Nipkow *et al.* defined a function, `sep`, using the `fun` keyword in an old Isabelle tutorial [1] as following:

```
fun sep :: "'a  'a list  'a list" where
"sep a [ ]      = [ ]" |
"sep a [x]      = [x]" |
"sep a (x#y#zs) = x # a # sep a (y#zs)"
```

Intuitively, this function inserts the first argument between any two elements in the second argument. Following this definition, Isabelle automatically derives the following auxiliary lemma, `sep.induct`, and registers it in the background proof context as well as other four automatically derived lemmas:

```
sep.induct: (!!a. ?P a []) ==> (!!a x. ?P a [x]) ==> (!!a x y zs. ?P a (y # zs)
  ==> ?P a (x # y # zs)) ==> ?P ?a0.0 ?a1.0
```

where variables prefixed with `?` are schematic variables and `!!` is the meta-logic universal quantifier. Isabelle also attaches unique names to these automatically derived lemmas following certain naming conventions hard-coded in Isabelle's source code. In this example, the full name of this lemma is `fun0.sep.induct`, which is a concatenation of the theory name (`fun0`), the delimiter (`.`), the name of the constant defined (`sep`), followed by a hard-coded postfix (`.induct`), which represents the kind of this derived lemma.

When users want to prove conjectures about `sep`, they can specify their conjectures using Isar keywords such as `lemma` and `theorem`. The Isar commands, `apply` and `by`, allow users to apply proof methods to these proof obligations. In the above example, Nipkow *et al.* proved the following lemma about `map` and `sep` using the automatically derived auxiliary lemma, `sep.induct`, as an argument to the proof method `induct_tac` as following:

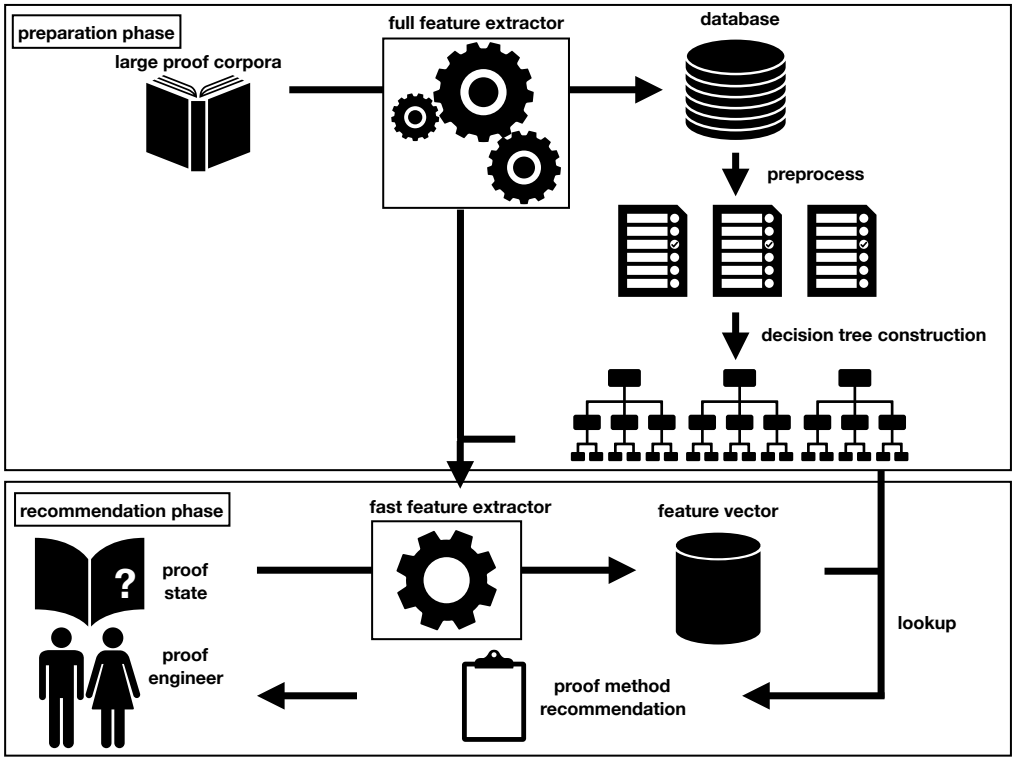

Fig. 1. Proof attempt with PaMpeR.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
apply(induct_tac x xs rule: sep.induct) apply simp_all done
```

where `simp_all` is a proof method that executes simplification to all sub-goals and `done` is another Isar command used to conclude a proof attempt.

Isabelle provides a plethora of proof methods, which serves as ammunitions when used by experienced Isabelle users; However, new Isabelle users sometimes spend hours or days trying to prove goals using proof methods sub-optimal to their problems without knowing Isabelle has specialized methods that are optimized for their goals.

2.2 Overview of PaMpeR

Figure 1 illustrates the overview of PaMpeR. The system consists of two phases: the upper half of the figure shows PaMpeR's preparation phase, and the lower half shows its recommendation phase.

In the preparation phase, PaMpeR's feature extractor converts the proof states in existing proof corpora such as the Archive of Formal Proofs (AFP) [9] into a database. This database describes which proof methods have been applied to what kind of proof state, while abstracting proof states as arrays of boolean values. This abstraction is a many-to-one mapping: it may map multiple distinct

proof states into to the same array of boolean values. Therefore, each array represents a group of proof states sharing certain properties.

PaMpeR first preprocesses this database and generates a database for each proof method. Then, PaMpeR applies a regression algorithm to each database and creates a regression tree for each proof method. This regression algorithm attempts to discover combinations of features useful to recommend which proof method to apply. Each tree corresponds to a certain proof method, and each node in a tree corresponds to a group of proof states, and the value tagged to each leaf node shows how likely it is that the method represented by the tree is applied to these proof states according to the proof corpora used as training sample.

For the recommendation phase, PaMpeR offers three commands, `which_method`, `why_method`, and `rank_method`. The `which_method` command first abstracts the state into a vector of boolean values using PaMpeR's feature extractor. Then, PaMpeR looks up the regression trees and presents its recommendations in Isabelle/jEdit's output panel. If you wonder why PaMpeR recommends certain methods, for example `auto`, to your proof state, type `why_method auto`. Then, PaMpeR tells you why it recommended `auto` to the proof state in jEdit's output panel. If you are curious how PaMpeR ranks a certain method, let us say `intro_classes`, type `rank_method intro_classes`. This command shows `intro_classes`'s rank given by PaMpeR in comparison to other proof methods. In the following, we describe these steps in detail.

3 Processing Large Proof Corpora

The key component of PaMpeR is its feature extractor: the extractor converts proof obligations, chained facts, and proof contexts into arrays of boolean values by applying assertions to them.

3.1 Representing a Proof State as an Array of Boolean Values

Currently we employ 60 assertions manually written in Isabelle's implementation language, Standard ML, based on our expertise in Isabelle/HOL. List 1 shows selected assertions we used in PaMpeR. Most of these assertions fall into two categories: assertions about proof obligations themselves, and assertions about the relation between proof obligations and information stored in the corresponding proof context.

Note that PaMpeR's assertions do not directly rely on any user-defined constants because PaMpeR's developers cannot access concrete definitions of user-defined constants when developing PaMpeR. For example, we can check if the first proof obligation has a constant defined in the `Set.thy` file in Isabelle/HOL, but we cannot check if that sub-goal has a constant defined in the proof script that some user developed after we released PaMpeR.

However, by investigating how Isabelle/HOL works, we implemented assertions that can check the meta-information of proof goal even without knowing their concrete specifications when developing PaMpeR. For example, the lemma presented in Section 2.1 has a function, `sep`, which was defined with the `fun` keyword. PaMpeR's feature extractor checks if the underlying proof context contains a lemma of name `sep.elims`. If the context has such a lemma, PaMpeR infers that a user defined `sep` using either the `fun` keyword or the `function` keyword, rather than other keywords such as `primcorec` or `definition`.

We wrote some assertions to reflect our own expertise in Isabelle/HOL. One example is the assertion that checks if the proof goal or chained facts involve the constant, `Filter.eventually`, defined in Isabelle's standard library. We developed such an assertion because we knew that the

List 1 List of Selected Assertions.

- Assertions about proof obligations themselves.
  - constants defined in Isabelle's standard library
    - check if the first goal has the `BNF_Def.rel_fun` constant or the `Fun.map_fun` constant.
    - check if the first goal has `"Orderings.ord_class.less_eq"`, `Orderings.ord_class.less`, or `Groups.plus_class.plus`.
    - check if the fist goal and its chained facts have `Filter.eventually`
  - constants defined in Isabelle's standard library at certain locations in the first proof obligation
    - check if the outermost constant of the first goal is the meta-logic universal quantifier
    - check if the first goal has the HOL existential quantifier but not as the outermost constant
  - terms of certain types defined in Isabelle's standard library
    - check if the first goal has a term of type `Word.word`
    - check if the first goal has a schematic variable
  - existence of constants defined in certain theory files
    - check if the first goal has a constant defined in the `Nat` theory
    - check if the first goal has a constant defined in the `Real` theory
    - check if the first goal has a constant defined in the `Set` theory
- Assertions about the relation between proof obligations and proof contexts.
  - types defined with a certain Isar keyword
    - check if the goal has a term of a type defined with the `datatype` keyword
    - check if the goal has a term of a type defined with the `codatatype` keyword
    - check if the goal has a term of a type defined with the `record` keyword
  - constants defined with a certain Isar keyword
    - check if the goal has a constant defined with the `lift_definition` keyword
    - check if the goal has a constant defined with the `primcorec` keyword
    - check if the goal has a constant defined with the `inductive` keyword or `inductive_set` keyword.

proof method called `eventually_elim` can handle many proof obligations involving this constant. But in some cases we were not sure which assertion can be useful to decide which method to use. For example, we have assertions to check if a proof goal has constants defined in `Set.thy`, `Int.thy`, or `List.thy` as these theory files define commonly used concepts in theorem proving. But their effects to proof method selection were unclear before the evaluation in Section 6.

More importantly, we did not know numerical estimates on which assertion is more useful than others when developing these assertions. For instance, we guessed that the assertion to check the use of the constant `Filter.eventually` to be useful to recommend the use of the `eventually_elim` method, but we did not have means of comparing the accuracy of this guess with other hints prior to this project. To obtain numerical assessments for proof method prediction, we applied the multi-output regression algorithm described in Section 4.

Unfortunately, not all assertions we developed turned out to be applicable to construct a database from large proof corpora. And we had to abandon some assertions that involve expensive operations, as they took more memory space than we could afford when applied to many proof states appearing in proof corpora. Even though small-scale experiments indicated that some of these abandoned assertions are useful to predict the use of certain proof methods, we could not even build a database from the standard library using these assertions and decided `PaMpeR` serves better without them.

The preliminary evaluation in Section 6 corroborates that it is possible to derive meaningful advice about proof methods. This implies that at least some parts of the expertise necessary to select appropriate proof methods are based on the meta-information about proof states or the information available within Isabelle's standard library, and our assertion-based feature extractor preserves at least some parts of the essence of proof states while converting them into simpler format.

3.2 Database Extraction

The first part of the preparation phase is to build a database from existing proof corpora. We modified the proof method application commands, `apply` and `by`, in Isabelle and implemented a logging mechanism to build the database. The modified `apply` and `by` take the following steps to generate the database:

1. apply assertions to the current proof state,
2. represent the proof state as an array of boolean values,
3. record which method is used to that array,
4. apply the method as the standard `apply` or `by` command, accordingly.

This step requires a slight modification to the Isabelle source code to allow us to overwrite the definition of these command. This way, we build its database by running the target proof scripts.

The current version of `PaMpeR` available at Github [3] is based on the database extracted from Isabelle's standard library, but the database extraction mechanism is not specific to this library. In case users prefer to optimise `PaMpeR`'s recommendation for their own proof scripts, they can take the same approach following the instructions at Github [3]; although this process tends to require significant computational resources.

This overwriting of `apply` and `by` is the only modification we made to Isabelle's source code, and we did so only to build the database for our machine learning algorithm. As long as users choose to use the off-the-shelf default learning results, they can use `PaMpeR` without ever modifying Isabelle's source code. In that case, they only have to include the theory file `PaMpeR.thy` into their own theory file using the Isar keyword `import` just as a normal theory file to use `PaMpeR`.

Note that logging mechanism ignores the `apply` commands that contain composite proof methods to avoid data pollution. When multiple proof methods are combined within a single command, the naive logging approach would record proof steps that are backtracked to produce the final result.

One exemplary data point in an extracted database would look as the following:

```
induct, [1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, ...]
```

where `induct` is the name of method applied to this proof state and the nth element in the list shows the result of the nth assertion of the feature extractor when applied to the proof state.

The default database construction from Isabelle standard library took about 462 CPU hours on our server machine, producing a database consisting of 139413 data points. Unfortunately, this database is heavily imbalanced: some proof methods are used far more often than others. For example, we have 26985 data points for the `auto` method, an aggressive general purpose proof method, while we have only 30 data points for the `coinduction` method, a proof method to apply co-induction. We discuss how this imbalance influenced the quality of `PaMpeR`'s recommendation in Section 6.

4 Machine Learning Databases

In this section, we explain the multi-output regression tree construction algorithm we implemented in Standard ML for `PaMpeR`. We chose a multi-output algorithm because there are in general multiple valid proof methods for each proof obligation, and we chose a regression algorithm rather than classification algorithm because we would like to provide numerical estimations about how likely each method would be useful to a given proof obligation.

4.1 Preprocess the Database

We first preprocess the database generated in Section 3.2. This process produces a separate database for each proof method from the raw database, which describes the use of all proof methods appearing in the target proof corpora.

For example, if our preprocessor finds the example line discussed in Section 3.2, it considers that an ideal user represented by the proof corpora decided to use the `induct` method but not other methods, such as `auto` or `coinduction`, and produces the following line in the database for `induct`:

```
used, [1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, ...]
```

and the following line in the databases for other proof methods appearing in the proof corpora:

```
not, [1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, ...]
```

Note that the resulting databases do not always represent a provably correct choice of proof methods but a conservative estimate. In principle, there could be multiple equally valid proof methods for a single proof state, but existing proof corpora describe only one way of attacking it. For example, Nipkow *et al.* applied the `induct_tac` method to the lemma in Section 2.1, but we can prove this lemma with another method for mathematical induction (`induction`) as following:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
apply(induction x xs rule: sep.induct) apply simp_all done
```

For this reason, this preprocessing may misjudge some methods to be inappropriate to a proof state represented by a feature vector in some cases. Unfortunately, exploring all the possible combinations of proof methods for each case is computational infeasible: some proof methods work well only when they are followed by other proof methods or they are applied with certain arguments, and the combination of these proof methods and arguments explodes quickly.

On the other hand, we can reasonably expect that the proof method appearing in our training sample is the right choice to the proof state represented by the feature vector, since Isabelle mechanically checks the proof scripts. Furthermore, we built the default recommendation using Isabelle's standard library, which was developed by experienced Isabelle developers. This allowed us to avoid low quality proof scripts that Isabelle can merely process but are inappropriate. Therefore, we consider the approximation `PaMpeR`'s preprocessor makes to be a realistic point of compromise and show the effectiveness of this approach in Section 6.

4.2 Regression Tree Construction

After preprocessing, we apply our regression tree construction algorithm to each created database separately. We implemented our tree construction algorithm from scratch in Standard ML for better flexibility and tool integration.

In general, the goal of the regression tree construction is to partition the feature space described in each database into partitions of sub-spaces that lead to the minimal Residual Sum of Squares (RSS) [4] while avoiding over-fitting. Intuitively, RSS denotes the discrepancy between the data and estimation based on a model. The RSS in our problem is defined as follows:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (used_i - \widehat{used}_{R_j})^2 \tag{1}$$

where R_j stands for the jth sub-space, to which certain data points (represented as lines in database) belong. The value of $used_i$ is 1.0 if the data point represented by the subscript i says the method was applied to the feature vector, and it is 0.0 if the data point represented by the subscript i says otherwise. $\widehat{used}_{R_j}$ is the average value of $used$ among the data points pertaining to the sub-space R_j.

Computing the RSS for every possible partition of the database under consideration is computational infeasible. Therefore, `PaMpeR`'s tree construction takes a top-down, greedy approach, called *recursive binary splitting* [5], as is often the case in many other implementations of decision tree algorithms.

In recursive binary splitting, we start constructing the regression tree from the root node, which corresponds to the entire dataset for a given method. First, we select a feature in such a way we can achieve the greatest reduction in RSS at this particular step. We find such feature by computing the reduction of the RSS by each feature by one level. For each feature, we split the database into two sub-spaces, $R_{used}(j)$ and $R_{not}(j)$ as follows:

$$R_{used}(j) = \{used | used_j = 1.0\} \text{ and } R_{not}(j) = \{used | used_j = 0.0\} \tag{2}$$

where j stands for the number representing each feature. Then, for each feature represented by j, we compute the following value:

$$\sum_{i:x_i \in R_{used}(j)} (used_i - \widehat{used}_{R_{used}(j)})^2 + \sum_{i:x_i \in R_{not}(j)} (used_i - \widehat{used}_{R_{not}(j)})^2 \tag{3}$$

and choose the feature j which minimizes this value.

Second, we repeat this partition procedure to each emerging sub-node of the regression tree under construction until the depth of tree hits our pre-defined upper limit.

After reaching the maximum depth, we compute the average value of $used(j)$ in the corresponding sub-space R for each leaf node. We consider this value as the expectation that the method is useful to proof states abstracted to the combination of feature values to that leaf node.

`PaMpeR` records these regression trees in a text file, `regression_trees.txt`, so that users can avoid the computationally intensive data extraction and regression tree construction processes unless they want to optimize the learning results based on their own proof corpora.

[4] RSS is also known as the sum of squared residuals (SSR).

Note that if we add more assertions to our feature extractor in future, the complexity of this algorithm increases linearly with the number of assertions given a fixed depth of regression tree, since the partition only takes the best step at each level instead of exploring all the combinations of partitions.

5 Recommendation Phase

Once finishing building regression trees for each proof method appeared in the given proof corpora, one can extract recommendations from `PaMpeR`. By default, `PaMpeR` uses the regression trees built from Isabelle 's standard library. When imported to users' theory file, `PaMpeR` automatically reads these trees using the `read_regression_trees` command in `PaMpeR.thy`.

`PaMpeR` provides three new commands to provide two kinds of information: the `which_method` command tells which proof methods are likely to be useful for a given proof state; the `why_method` command takes a name of proof method and tells why `PaMpeR` would recommend the proof method for the proof state; and the `rank_method` command shows the rank of a given method to the proof state in comparison to other proof methods.. In the following, we explain how these two commands produce recommendations from the regression trees produced in the preparation phase.

5.1 Faster Feature Extractor

Before applying the machine learning algorithm, we were not sure which assertion produces valuable features, but after applying the machine learning algorithm, we have an estimate which assertions are not useful, by checking which features are used to branch each regression tree.

To reduce the waiting time of `PaMpeR`'s users, the `build_fast_feature_extractor` command in `PaMpeR.thy` constructs a faster feature extractor from the regression trees built in the preparation phase and the full feature extractor. It builds the faster feature extractor by removing assertions that do not result in a branch in the regression trees.

5.2 The `which_method` command.

When users invoke the `which_method` command, `PaMpeR` applies the fast feature extractor to convert the ongoing proof state into a feature vector, which consists of those features that are deemed to be important to make a recommendation. The speed of this faster feature vector depends on both the regression trees and what each proof state contains. As a rule of thumb, if the proof goal has less terms, it tends to spend less time.

Then, `PaMpeR` looks up the corresponding node in each regression tree and decides the expectation that the method is the right choice for the proof state represented by the feature vector. `PaMpeR` computes this value for each proof method it encountered in the training proof corpora, by looking up a node in each regression tree. Finally, `PaMpeR` compares these expectations and shows the 15 most promising proof methods with their expectations in Isabelle/jEdit's output panel. In the ongoing example from Section 2.1, one can learn which method to use by typing the `which_method` command as following:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)
which_method
```

Then, PaMpeR shows the following message in the output panel for the top 15 methods [5]:

```
A promising method is simp with expectation of 0.461477173134
A promising method is auto with expectation of 0.163514167155
A promising method is rule with expectation of 0.111671840746
A promising method is induction with expectation of 0.106437454279
A promising method is induct with expectation of 0.0527912621359...
```

Attentive readers might have noticed that PaMpeR's recommendations are not identical to the model answer provided by Nipkow *et al.*. This, however, does not immediately mean PaMpeR's recommendation is not valuable: in fact, PaMpeR recommended the `induction` method at the fourth place, and `induction` is also a valid method for this proof goal as discussed in Section 4.1.

5.3 The `why_method` command.

Our rather straightforward machine learning algorithm makes PaMpeR's recommendation *explainable*. If you wonder why PaMpeR recommends a certain method, for example `coinduction`, to your proof obligation, type `why_method coinduction` in the proof script. PaMpeR first checks features used to evaluate the expectation for the method and their feature values. Second, PaMpeR shows qualitative explanations tagged to both these features and their values in jEdit's output. If you wonder why PaMpeR recommended `induction` in the above example, type the following:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)
why_method induction
```

Then, you will see this message in jEdit's output panel:

```
Because the underlying proof context has a pinduct rule associated to one of the
constants appearing in the first subgoal. Note that these rules are automatically
derived by the function or fun keyword.
Because it is not true that the context has locally defined assumptions.
```

The first reason corresponds to the first branching at the root node in the regression tree for the `induction` method, and the second reason corresponds to the second branching in the tree. Since we are using a greedy method to construct regression trees, the first reason tends to be more important than the second reason, as you might have noticed from this example.

5.4 The `rank_method` command.

Sometimes users already have a guess as to which proof method would be useful to their proof state, but they want to know how PaMpeR the proof method in mind. Continuing with the above example, if you want to know how PaMpeR ranks `conduction` for this proof state, type the following:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)
rank_method coinduction
```

Then, PaMpeR warns you:

```
coinduction 90 out of 159
```

indicating that PaMpeR does not consider `coinduction` to be the right choice for this proof obligation, before you waste your time on emerging sub-goals appearing after applying `coinduction`.

[5] Note that we truncated the message due to the space restriction here.

6 Preliminary Evaluation

We conducted a cross-validation to assess the accuracy of PaMpeR's `which_method` command. For this evaluation, we used Isabelle's standard library as training dataset and 65 articles in the AFP as testing set. We built the training dataset and evaluation dataset separately and removed the occurrences of method invocations that appear in the training dataset from the evaluation dataset, thus avoiding evaluating the accuracy of PaMpeR using data points used for training.

First, we extracted a database from the training dataset. Second, we built regression trees from this database. Then, we evaluated how often PaMpeR's recommendation matches the proof method invocation recorded in the testing dataset.

Since there are often multiple equally valid proof methods for each proof state, it is only reasonable to expect that `which_method` command should be able to recommend the proof method used in the testing set as one of the most important methods for the given proof method invocation. For this reason, for each proof method, we measured how often each proof method used in the testing dataset appears among the top n methods in PaMpeR's recommendations.

Table 1 shows the two parts of the evaluation results. The upper part shows the results for the 15 proof methods that are most frequently used in the training data in the descending order. The lower part shows the results for proof methods that are less frequently used in the training data but PaMpeR managed to recommend among the top 15 proof methods at more than 30% of chance.

For example, the top row for `simp` should be interpreted as following: The `simp` method was used 38251 times in the training data. This amounts to 27.4% of all proof method invocations in the training data that are recorded by PaMpeR . In the testing set, `simp` was used 36643 times, which amounts to 24.3% of proof method invocations in the testing data that are recorded by PaMpeR. For 60% out of 36643 `simp` invocations, PaMpeR predicted that `simp` is the most promising method for the corresponding proof states. For 98% out of 36643 `simp` invocations, PaMpeR recommended that `simp` is either the most promising method or the second most promising method for the corresponding proof states.

Table 2 shows the results for *selected* exemplary proof methods that are less frequently used in the training data but for which PaMpeR failed to recommend them among the top 15 proof methods at more than 30% of chance. Due to the space restriction, we leave the complete table at our Github repository [3] together with the details of this evaluation.

The overall results of this evaluation are as follows: PaMpeR learnt 159 proof methods from Isabelle's standard library. The AFP articles used 89 methods out of these 159 methods. For 38 proof methods, PaMpeR recommended these as one of the top 15 methods, at more than 50% of chance in the testing data. For 44 proof methods, PaMpeR recommended these as one of the top 15 methods, more than 30% of chance in the testing data. For all the 15 most frequently applied methods, PaMpeR managed to recommend them as one of the 15 most promising methods more than 50% of chance.

These two tables show that PaMpeR provides valuable recommendations when proof states are best handled by special purpose proof methods, such as `unfold_locales`, `transfer`, `arith`, `standard`, and so on; however, PaMpeR's recommendations tend to differ from the AFP authors' choice when they chose less commonly used general purpose proof methods, such as `fast` and `clarimp`, as shown in Table 2. This is mainly due to the fact that in many cases multiple general purpose proof methods can handle the same proof obligation equally well.

PaMpeR's straightforward regression tree construction does not severely suffer from the imbalance among proof method invocation, even though class imbalance often causes problems in other

Table 1. Evaluation of PaMpeR.

proof method	training	%	test	%	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
simp	38251	27.4	36643	24.3	60	98	99	99	99	100	100	100	100	100	100	100	100	100	100
auto	26985	19.4	39996	26.5	54	84	94	96	97	98	99	100	100	100	100	100	100	100	100
rule	16547	11.9	15040	10.0	5	7	44	84	96	100	100	100	100	100	100	100	100	100	100
blast	8500	6.1	8003	5.3	0	21	21	31	53	81	94	98	99	100	100	100	100	100	100
metis	5147	3.7	6757	4.5	0	0	0	17	28	37	39	46	58	72	82	88	91	93	96
intro	3063	2.2	2234	1.5	0	0	0	2	10	11	19	42	70	83	89	92	93	94	94
erule	2734	2.0	1463	1.0	0	0	2	8	18	28	38	44	56	61	63	65	65	66	66
induct	2672	1.9	3609	2.4	0	6	8	9	13	19	48	66	74	77	78	78	80	86	92
rule_tac	2586	1.9	1018	0.7	0	21	23	28	30	31	32	34	38	40	41	41	42	46	51
force	2397	1.7	2861	1.9	0	0	0	0	0	3	12	23	34	44	51	67	78	88	93
cases	2381	1.7	3383	2.2	0	0	0	0	0	19	46	65	72	75	75	77	79	82	85
subst	2369	1.7	2746	1.8	0	0	0	0	2	6	14	26	37	42	46	50	55	61	65
simp_all	2088	1.5	2477	1.6	0	0	0	0	0	0	0	1	9	24	44	56	71	81	88
drule	1672	1.2	763	0.5	0	0	0	0	4	5	8	10	20	37	54	72	86	94	97
drule_tac	1563	1.1	208	0.1	0	0	0	1	1	1	1	12	25	37	54	67	74	76	78
fastforce	1315	0.9	2663	1.8	0	0	0	0	0	0	0	0	2	9	14	22	36	51	65
unfold	1261	0.9	1525	1.0	0	0	0	0	2	5	7	7	7	9	18	31	47	60	67
transfer	1161	0.8	1418	0.9	0	49	56	56	56	56	56	56	58	62	72	80	88	92	92
fact	1146	0.8	2354	1.6	0	4	4	4	4	4	4	8	17	31	48	53	53	53	53
case_tac	934	0.7	918	0.6	0	0	0	2	7	8	10	12	15	16	20	41	54	61	68
subgoal_tac	870	0.6	329	0.2	0	0	0	0	0	0	0	0	0	5	9	21	33	44	51
meson	685	0.5	333	0.2	0	0	0	0	0	0	0	0	0	0	5	13	25	43	48
assumption	665	0.5	459	0.3	5	6	6	17	33	42	47	48	48	48	48	48	48	49	49
induction	644	0.5	916	0.6	0	0	0	9	18	23	27	27	27	27	27	27	28	34	36
arith	594	0.4	428	0.3	0	0	0	0	0	0	32	48	49	49	50	50	50	50	50
erule_tac	481	0.3	214	0.1	0	0	0	0	0	0	0	1	17	23	33	41	49	49	50
safe	458	0.3	523	0.3	0	3	9	13	13	13	13	13	14	20	24	30	30	34	40
standard	415	0.3	246	0.2	44	54	55	59	70	76	76	77	78	78	78	78	78	78	78
unfold_locales	176	0.1	1117	0.7	0	0	0	0	8	14	32	39	41	43	47	50	54	54	55
eventually_elim	153	0.1	297	0.2	0	0	27	46	61	84	97	99	99	100	100	100	100	100	100
intro_classes	116	0.1	236	0.2	70	74	91	100	100	100	100	100	100	100	100	100	100	100	100
contradiction	78	0.1	46	0.0	0	0	0	0	0	0	0	0	0	0	70	70	74	74	74
transfer_prover	68	0.0	114	0.1	59	61	69	93	96	96	96	96	96	96	96	96	96	96	96
relation	65	0.0	63	0.0	84	84	84	84	84	84	84	84	84	84	84	84	84	84	84
pat_completeness	57	0.0	115	0.1	96	96	96	96	96	96	96	96	96	96	96	96	96	96	96
countable_datatype	37	0.0	12	0.0	0	0	0	0	0	0	0	0	0	0	0	100	100	100	100
eval	35	0.0	111	0.1	0	0	0	0	0	0	0	0	0	0	12	27	52	52	53
uint_arith	32	0.0	185	0.1	63	69	69	69	69	69	69	69	69	69	69	69	69	69	69
coinduction	30	0.0	118	0.1	0	13	32	33	33	33	33	33	33	33	33	33	33	33	33
coinduct	20	0.0	61	0.0	0	0	0	0	0	0	0	30	33	33	33	33	33	33	33
atomize	20	0.0	9	0.0	0	0	0	44	100	100	100	100	100	100	100	100	100	100	100
moura	10	0.0	7	0.0	0	0	0	0	0	0	0	0	0	0	14	57	100	100	100
lexicographic_order	7	0.0	17	0.0	0	0	0	6	6	6	6	6	29	65	65	65	65	65	65
transfer_step	4	0.0	5	0.0	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Table 2. Evaluation results for some proof methods, for which the `which_method` command did not produce valuable recommendations.

proof method	training	%	test	%	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`fast`	1047	0.8	1311	0.9	0	0	0	8	10	10	11	11	11	11	12	12	13	14	15
`clarsimp`	861	0.6	2311	1.5	0	4	4	4	4	4	4	4	4	4	4	5	7	8	11
`tactic`	468	0.3	30	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3
`induct_tac`	335	0.2	116	0.1	0	0	0	0	2	6	7	7	7	7	7	7	7	7	7
`nominal_induct`	280	0.2	24	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`linarith`	235	0.2	104	0.1	0	0	0	0	14	15	15	15	16	19	19	19	19	19	19
`rename_tac`	221	0.2	258	0.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`rotate_tac`	120	0.1	6	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`smt`	116	0.1	129	0.1	0	0	0	12	16	20	20	20	20	20	20	20	20	20	20
`sos`	72	0.1	7	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`ind_cases`	72	0.1	18	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`normalization`	68	0.0	40	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`vector`	60	0.0	73	0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`vcg`	11	0.0	198	0.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

domains such as fraud detection and medical diagnosis. For example, `PaMpeR` managed to produce valuable recommendations for `relation`, `transfer_prover`, `pat_completeness`, `uint_arith`, and `transfer_step`, even though none of them is used more than 0.1% of times in the training set.

The reason the imbalance did not cause serious problems to `PaMpeR` is that many of these rarely used methods are specialised proof methods, for which we can write assertions that can abstract the essence of the problem very well. Another reason is the fact that commonly used proof methods tend to hold up each other's share, since they address similar problems, lowering expectations for commonly used general purpose methods where both specialised methods and general purpose methods can discharge proof obligations.

On the other hand, `PaMpeR` did not produce valuable recommendations to some special purpose proof methods, such as `nominal_induct`, `normalization`, and `vcg`. For `normalization`, we did not manage to develop assertions that capture the properties shared by the proof obligations that `normalization` can handle well. For `nominal_induct` and `vcg`, we were unaware of their presence in Isabelle's standard library and assertions for these methods remain as our future work.

Some of the seemingly poor results in Table 2 do not directly imply the inaccuracy of `PaMpeR`. For instance, the `tactic` method is just an interface between Standard ML and Isar and does not have semantic meaning itself; predicting such methods is not a very meaningful task for `PaMpeR`.

7 Design Decisions: No Modification to Isabelle's Source Code

Developing assertions involves careful engineering work, which requires familiarity with Isabelle's internal APIs. As described above, our assertions infer the meta information about proof obligation by looking up automatically derived theorems stored in proof states, and `PaMpeR` deduces which definitional mechanism was used to specify the proof obligation. Alternatively, we could have modified each definitional mechanism in Isabelle and added a logging mechanism to all of them. This approach is what we purposefully avoided, since it inevitably involves modifications to many parts

of Isabelle's source code and it has to store an extra persistent state within proof state to keep redundant information, which our assertions can infer without it. For software that is expected to be trustworthy such as Isabelle and involves many developers, such large scale modifications should be best avoided unless there is no other way around. We claim that our approach takes advantage of existing Isabelle mechanisms while respecting its modular design.

The drawback of our approach is that it relies on the naming conventions hard-coded in Isabelle/HOL. If Isabelle developers modify these naming conventions in future, `PaMpeR`'s feature extractor loses the original intention and produces less valuable databases. To detect such change of naming convention, we developed two Isar commands, `assert_nth_true` and `assert_nth_false`, for unit testing assertions. For instance, Isabelle can process the following proof script only if the fourth assertion returns `true` when applied to the conjecture and the fifth assertion returns `false`, otherwise these unit test commands force Isabelle to fail.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)
assert_nth_true 4
assert_nth_false 5
```

We inserted these commands into several parts of our test suite comprising of selected articles from the AFP. This way, we can detect problems automatically, when assertions start producing unexpected results due to a possible future change of Isabelle's naming convention.

8 Discussion and Future Work

Before the advent of PaMpeR, new Isabelle/HOL users have to go through various documentations and the archive of mailing lists to learn how to prove lemmas in Isabelle/HOL independently.

Choosing the right methods was a difficult task for new ITP users especially when they should choose special-purpose proof methods, since new users tend not to know even the existence of those rarely used proof methods. Some proof methods are strongly related to certain definitional mechanisms in Isabelle. Therefore, when Isabelle experts use such definitional mechanisms, they can often guess which proof methods they should use later. But this is not an easy task for new Isabelle users. This is becoming truer nowadays, since large scale theorem proving projects are slowly becoming popular and new ITP users often have to take over proof scripts developed by others and they also have to discharge proof obligations specified by others. `PaMpeR` partially solved this problem by systematically transferring experienced users knowledge to less experienced users. We plan to keep improving `PaMpeR` by incorporating other Isabelle users intuitions as assertions.

Our manually written feature extractor may seem to be naive compared to the recent success in machine learning research: in some problem domains, such as image recognition and the game of Go, deep neural networks extract features of the subject matters via expensive training. Indeed, others have applied deep neural networks to theorem proving, but only with limited success [7].

The two major problems of automatic feature extraction for theorem proving is the lack of enormous database needed to train deep neural networks and the expressive nature of the underlying language, i.e. logic. The second problem, the expressive nature of logic, contributes to the first problem: self-respecting proof engineers tend to replace multiple similar propositions with one proposition from which one can easily conclude similar propositions, aiming at a succinct presentation of the underlying concept. What is worse, when working on modern ITPs, it is often not enough to reason about a proof goal, but one also has to take its proof context into consideration.

A proof context usually contains numerous auxiliary lemmas and nested definitions, and each of them is a syntax tree, making the effective automatic feature extraction harder.

Furthermore, whenever a proof author defines a new constant or prove a new lemma Isabelle/HOL changes the underlying proof context, which affects how one should attack proof obligations defined within this proof context. And proof authors do add new definitions because they use ITPs as specification tools as well as tools for theorem proving. Some of these changes are minor modifications to proof states that do not severely affect how to discharge proof obligations in the following proof scripts, but in general changing proof contexts results in, sometimes unexpected, problems.

For this reason, even though the ITP community has large proof corpora, we are essentially dealing with different problems in each line of each proof corpus. For example, even the AFP has 396 articles consisting of more than 100,000 lemmas, only 4 articles are used by more than 10 articles in the AFP, indicating that many proof authors work on their own specifications, creating new problems. This results in an important difference that lies between theorem proving in an expressive logic and other machine learning domains, such as image recognition where one can collects numerous instances of similar objects. For instance, it is reasonable to expect that we can collect billions of pictures of cats to abstract the features of cats automatically, but it is not realistic to achieve millions of formalizations of the same concept, say Green's Theorem, in Isabelle/HOL to extract the feature of Green's theorem.

We addressed this problem with human-machine cooperation, the philosophy that underpins ITPs. Even though it is hard to extract features automatically, experienced ITP users know that they can discharge many proof goals with shallow reasoning. We encoded experienced Isabelle users' expertise as assertions to simulate their shallow reasoning. Since these assertions are carefully handwritten in Isabelle/ML, they can extract features of proof states (including both proof goal, chained facts, and its context) despite the above mentioned problems.

Currently `PaMpeR` recommends only which methods to use and shows why it suggests that method. This is enough for many special purpose methods which do not take parameters. For other methods, such as `induct`, it is often indispensable to pass the correct parameters to guide methods. If you prefer to know which arguments to pass to the proof method `PaMpeR` recommends, we would advise to use `PSL` [6], the proof strategy language for Isabelle/HOL, which attempts to find the right combination of arguments through an iterative deepening depth first search based on rough ideas about which method to use.

Moreover, none of `PaMpeR`'s assertions takes the sequence of proof method applications into account: even though they can check the information contained in the background proof context, the parse-then-consume style of Isabelle/Isar makes it difficult for `PaMpeR` to trace which methods have been applied to reach the current proof state.

We envision a more powerful proof automation tool backed by `PaMpeR` and our proof strategy language, `PSL`. We use `PaMpeR`'s recommendation to navigate the search of `PSL`, changing `PSL`'s evaluation strategy from the IDDFS to the best-first search. `PSL` provides a mechanism to generate variants of proof methods with different combinations of parameters to find the right combination of parameters for a given goal through a search. `PSL`'s automatic removal of backtracked proof steps eliminates the data pollution problem discussed in Section 3.2. `PSL`'s framework to write history-sensitive proof methods allows us to write history-sensitive assertions, so that `PaMpeR` can take the sequence of proof methods into account.

9 Conclusion and Related Work

We presented the design and implementation of `PaMpeR`. In the preparation phase, `PaMpeR` learns which method to use from existing proof corpora using regression tree construction algorithm. In the recommendation phase, `PaMpeR` recommends which proof methods to use to a given proof obligation and explains why it suggests that method. Our evaluation showed that `PaMpeR` tends to provide valuable recommendations especially for specialised proof methods, which new Isabelle users tend not to be aware of. We also identified problems that arise when applying machine learning to proof method recommendation and proposed our solution to them.

Related Work. ML4PG [8] extends a proof editor, Proof General, to collect proof statistics about shapes of goals, sequence of applied tactics, and proof tree structures. It also clusters the gathered data using machine learning algorithms in MATLAB and Weka and provides proof hints during proof developments. Based on learning, ML4PG lists similar proof obligations proved so far, from which users can infer how to attack the proof obligation at hand, while `PaMpeR` directly works on proof methods. Compared to ML4PG, `PaMpeR`'s feature extractor is implemented within Isabelle/ML, which made it possible to investigate not only proof obligations themselves but also their surrounding proof context.

Gauthier developed *et al.* TacticToe for HOL4. It selects proved lemmas similar to the current proof obligation using premise selection and applies tactics used to these similar obligations to discharge the current proof obligation. Compared to TacticToe, the abstraction via assertions allows `PaMpeR` to provide valuable recommendations even when similar obligations do not exist in the problem domain.

Several people applied machine learning techniques to improve the so-called Hammer-style tools. For Isabelle/HOL, both MePo [12] and MaSh [13] decreased the quantity of facts passed to the automatic provers while increasing their quality to improve Sledgehammers performance. Their approaches attempt to choose facts that are likely to be useful to the given proof obligation, while `PaMpeR` suggests proof methods that are likely to be useful to the goal.

MePo judges the relevance of facts by checking the occurrence of symbols appearing in proof obligations and available facts, while MaSh computes the relevance using sparse naive Bayes and k Nearest Neighbours. They detect similarities between proof obligations and available facts by checking mostly formalization-specific information and only two piece of meta information, while `PaMpeR` discards most of problem specific information and focus on meta information of proof obligations: the choice of relevant fact is a problem specific question, while the choice of proof method largely depends on which Isabelle's subsystem is used to specify a proof obligation.

The original version of MaSh was using machine learning libraries in Python, and Blanchette *et al.* ported them from Python to Standard ML for better efficiency and reliability. Similarly, an early version of `PaMpeR` was also using a Python library [2] until we implemented the regression tree construction algorithm in Standard ML for better tool integration and flexibility. Both MaSh and `PaMpeR` record learning results in persistent states outside the main memory, so that users can preserve the learning results even after shutting down Isabelle.

Acknowledgements This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

References

1. Nipkow, T., Paulson C. P., and Wenzel M. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, Springer, 2002
2. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E. Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, volume 12, 2011
3. Nagashima, Y. PSL and PaMpeR (2017), https://github.com/data61/PSL/tree/PaMpeR/PaMpeR
4. Nagashima, Y. PSL and PaMpeR (2017), https://github.com/data61/PSL/wiki/PMRS-Assertion
5. James, G., Witten, D., Hastie, T., Tibshirani R. An Introduction to Statistical Learning with Applications in R
6. Nagashima, Y., Kumar, R. A Proof Strategy Language and Proof Script Generation for Isabelle, arXiv, (2016)
7. Irving,G., Szegedy, C., Alemi, A. A., Eén, N., Chollet, F., Urban, J. DeepMath - Deep Sequence Models for Premise Selection, Annual Conference on Neural Information Processing Systems (2016)
8. Komendantskaya, E., Heras, J., and Grov, G. Machine Learning in Proof General: Interfacing Interfaces, Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012.
9. Gerwin Klein, Tobias Nipkow, Larry Paulson, René Thiemann, Archive of Formal Proofs, https://www.isa-afp.org/
10. Dmitriy Traytel, A Codatatype of Formal Languages, Archive of Formal Proofs
11. TacticToe: Learning to reason with HOL4 Tactics, Gauthier, T., Kaliszyk, C., Urban, J., LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning
12. Jia Meng, L C. Paulson, Lightweight relevance filtering for machine-generated resolution problems, J. Applied Logic (2009)
13. Blanchette, J., Greenaway, D., Kaliszyk C., Kühlwein, D., Urban, J., A Learning-Based Fact Selector for Isabelle/HOL, J. Autom. Reasoning (2016)
14. Markus Wenzel, Isar - A Generic Interpretative Approach to Readable Formal Proof Documents, TPHOLs'99, Nice, France, September (1999)