PaMpeR: Proof Method Recommendation System for Isabelle/HOL

Yutaka Nagashima
CIIRC, Czech Technical University in Prague
Prague, Czech Republic
Department of Computer Science, University of Innsbruck
Innsbruck, Tyrol, Austria
yutaka.nagashima@uibk.ac.at

Yilun He
School of Information Technologies, University of Sydney
Sydney, New South Wales, Australia
yihe8397@uni.sydney.edu.au

ABSTRACT

Deciding which sub-tool to use for a given proof state requires expertise specific to each interactive theorem prover (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.

CCS CONCEPTS

• **Information systems → Recommender systems**; *Information extraction*; • **Security and privacy → Logic and verification**; • **Human-centered computing** → *User interface toolkits*; • **Theory of computation** → *Higher order logic*;

KEYWORDS

Isabelle/HOL, recommendation system, data mining, proof method, interactive theorem prover

ACM Reference Format:
Yutaka Nagashima and Yilun He. 2018. PaMpeR: Proof Method Recommendation System for Isabelle/HOL. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238210

1 INTRODUCTION

Do you know when to use the proof method[1] called `intro_classes` in Isabelle? What about `uint_arith`? Can you tell when `fastforce` tends to be more powerful than `auto`? If you are an Isabelle expert, 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?*"

Interactive theorem provers (ITPs) are forming the basis of reliable software engineering. Klein *et al.* proved the correctness of the seL4 micro-kernel in Isabelle/HOL [11]. Leroy developed a certifying C compiler, CompCert, using Coq [15]. Kumar *et al.* built a verified compiler for a functional programming language, CakeML, in HOL4 [14]. In mathematics, mathematicians are replacing their pen-and-paper proofs with mechanised proofs to avoid human-errors in their proofs: Hales *et al.* mechanically proved the Kepler conjecture using HOL-light and Isabelle/HOL [6], whereas Gonthier *et al.* finished the formal proofs of the four colour theorem in Coq [5]. In theoretical computer science, Paulson proved Gödel's incompleteness theorems using Nominal Isabelle [22].

To facilitate efficient proof developments in such large scale verification projects, modern ITPs are equipped with many sub-tools, such as proof methods and tactics. For example, Isabelle/HOL comes with 160 proof methods defined in its standard library. These sub-tools provide useful automation for interactive theorem proving; however, it still requires ITP 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. The implementation is available at GitHub [1]. 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 extensive evaluation of PaMpeR to assess the accuracy of PaMpeR's recommendations. Section 7 discusses the strengths and limitations of the current implementation and the future work that might improve PaMpeR's performance further or provide even more detailed evaluation of the current implementation. Section 8 compares our work with other attempts of applying machine learning and data mining to interactive theorem proving.

[1] Proof methods are tools used to discharge proof goals in Isabelle. They are similar to *tactics* in other LCF-style provers.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238210

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* [24] (*Isar* for short) is a proof language used in Isabelle/HOL. Isar provides a human-friendly interface to specify and discharge proof goals. Isabelle users discharge proof goals by applying *proof methods*, which are the Isar syntactic layer of LCF-style tactics.

Each proof goal 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 goal, 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 goals 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 goal, 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 [21] as follows:

```
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 ?, such as ?a0.0, are schematic variables, !! is the meta-logic universal quantifier, ==> is the meta-logic implication[2]. 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 goals. 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:

```
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 serve 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 already 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) [12] 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

[2] Isabelle/HOL is a specialization of Isabelle for Higher-Order Logic (HOL) formalized in Isabelle's meta-logic. Therefore, it has two versions of universal quantifier and implication: one in the meta-logic and the other one in HOL

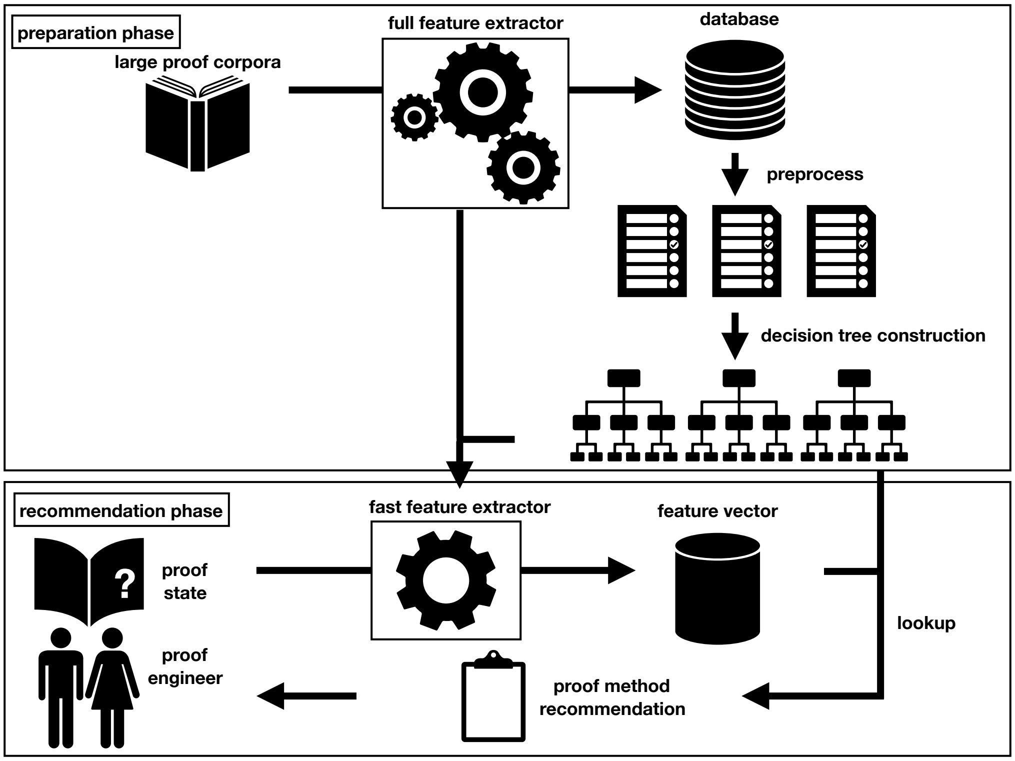

Figure 1: Proof attempt with PaMpeR.

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 goals, 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 108 assertions manually written in Isabelle's implementation language, Standard ML, based on our expertise in Isabelle/HOL. Table 1 shows selected assertions we used in PaMpeR. Most of these assertions fall into two categories: assertions about proof goals themselves, and assertions about the relation between proof goals 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 goal 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 proof method called `eventually_elim` can handle many proof goals 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 until we conducted an extensive evaluation described 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

Table 1: Selected Assertions.

- Assertions about proof goals 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 first goal and its chained facts have `Filter.eventually`
  - constants defined in Isabelle's standard library at certain locations in the first proof goal
    - 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 goals 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.

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.

The evaluation in Section 6 corroborates that it is possible to derive meaningful advice about proof methods. This implies that 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 some essence of proof states while converting them into simpler format.

3.2 Database Extraction from Large Proof Corpora

The first step 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 our website [1] is based on the database extracted from Isabelle's standard library and the AFP, 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 our website [1], even though 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/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 and the AFP took about 6,021 hours 43 minutes of CPU time, producing a database consisting of 425,334 unique data points. We used three multi-core server machines[3] to reduce the clock time necessary to obtain this dataset. Unfortunately, this database is heavily imbalanced: some proof methods are used far more often than others. 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 goal, and we chose a regression algorithm rather than classification algorithm because we would like to provide numerical estimates about how likely each method would be useful to a given proof goal. We chose a regression tree construction algorithm [3] because this simple algorithm allows us to produce qualitative explanations as to why PaMpeR recommends certain methods and it works well for small datasets for rarely used methods as shown in Section 6.The comparisons of various machine learning algorithms remain as our future work.

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 most proof methods appearing in the target proof corpora.

Among the class of problem transformation methods for multi-output regression problems, this straightforward approach is called single-target method: it first transforms a single multi-output problem into several single-target problems, then applies a regression algorithm to each of them separately, then combines the results of each regression algorithm to build a single predictor for the original multi-output problem.

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 preprocessor adds 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 conservative estimates. 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 follows:

```
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 computationally 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, and the AFP, which accepts new proofs only after peer reviews by Isabelle experts. 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:

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

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* [10].

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

[3] One of them has 2 Intel(R) Xeon(R) CPUs E5-2698 v3 @ 2.30GHz with 16 cores for each and with hyperthreading, the other two have 2 Intel(R) Xeon(R) CPUs E5-2690 v4 @ 2.60GHz with 14 cores for each with hyperthreading.

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

feature by one level. For each feature, we split the database into two sub-spaces, $R_{used}(j)$ and $R_{not}(j)$ as follows:

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

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

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

and choose the feature j that 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, three.

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, 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.

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. When imported to users’ theory file, PaMpeR automatically reads these trees using the `read_regression_trees` command in `PaMpeR/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; 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 three 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 can judge which assertions are not useful, by checking which features are used to branch each regression tree. The `build_fast_feature_extractor` command in `PaMpeR/ PaMpeR.thy` constructs a faster feature extractor from the regression trees built in the preparation phase and the full feature extractor to reduce the waiting time of PaMpeR’s users. 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 faster 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 on-going example from Section 2.1, a user can know which method to use by typing the `which_method` command as follows:

```
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]:

```
Promising methods for this proof goal are:
  simp with expectation of 0.4119
  auto with expectation of 0.1593
  rule with expectation of 0.0874
  induction with expectation of 0.06137
  metis with expectation of 0.05260 ...
```

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 out of 239 proof methods, 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 `case_tac`, to your proof goal, type `why_method case_tac` 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:

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

```
Because it is not true that the context has locally
defined assumptions.
Because the underlying proof context has a recursive
simplification rule related to a constant appearing in
the first subgoal.
```

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. In this case, PaMpeR found that the proof goal involves the constant, sep, and the underlying proof context contains a simplification rule, sep.simps(3), which involves a recursive call of sep as following:

```
sep.simp(3):
 sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
```

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 ranks 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 123 out of 239
```

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

6 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 and the AFP as follows: First, we extracted a database from these proof corpora. This database consists of 425,334 data points. Second, we randomly chose 10% of data points in this database to create the evaluation dataset. Third, we built regression trees from the remaining 90%. There is no overlap between the evaluation dataset and training dataset. Then, we applied regression trees to each data point in the evaluation dataset and counted how often PaMpeR's recommendation coincides with the proof methods chosen by human proof authors.

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

Table 2 shows the results for the 15 proof methods that are most frequently used in the training data in the descending order.

For example, the top row for simp should be interpreted as following: The simp method was used 102,441 times in the training data. This amounts to 26.8% of all proof method invocations in the training data that are recorded by PaMpeR. In the evaluation dataset, simp was used 11,385 times, which amounts to 26.8% of proof method invocations in the evaluation dataset that are recorded by PaMpeR. For 58% out of 11,385 simp invocations in the evaluation dataset, PaMpeR predicted that simp is the most promising method for the corresponding proof states. For 98% out of 11,385 simp invocations in the evaluation dataset, PaMpeR recommended that simp is either the most promising method or the second most promising method for the corresponding proof states.

Note that the numbers presented in this table are not the success rates of PaMpeR's recommendation but its conservative estimates. Assume PaMpeR recommends simp as the most promising method and auto as the second most promising method to a proof goal, say pg, in the evaluation dataset, but the human proof author of pg chose to apply auto to this proof goal. This does not immediately mean that PaMpeR failed to recommend auto in the first place, because both simp and auto might be equally suitable for pg. Therefore, the 58% for simp mentioned above should be interpreted as follows: PaMpeR's recommendation coincides with the choice of experienced Isabelle user for 58% of times where human engineers applied simp when PaMpeR is allowed to recommend only one proof method, but the real success rate of PaMpeR's recommendation can be higher than 58% for these cases. To avoid the confusion with *success rate*, we introduce the term, *coincidence rate*, for this measure. Appendix of our technical report [19] presents three tables to provide the complete list of the evaluation results.

The overall results of this evaluation are as follows: PaMpeR learnt 239 methods from Isabelle's standard library and the AFP: 160 of them are defined within Isabelle's standard library, and the others are user-defined proof methods specified in the AFP entries.

Out of the 239 proof methods PaMpeR learnt from the training dataset, 171 proof methods appeared in the evaluation dataset. Out of these 171 proof methods within the evaluation dataset, 133 methods are defined in Isabelle's standard library, and 38 methods were defined by the AFP authors.

The distribution of proof method usage is heavily imbalanced. The three most frequently used proof methods (simp, auto, and rule) account for 59.1% of all data points in the training dataset, and the ten most frequently used methods account for 79.2% in the training dataset. Similarly in the evaluation dataset, the top three methods account for 58.9%, and the top ten methods for 79.1%.

Fig. 2 illustrates this imbalance, in which the horizontal axis represents the rank of method usage for a proof method and the vertical axis stands for the number of methods invocations for that proof method. For instance, the square located at the top-left corner denotes that the most frequently used proof method in the training dataset (simp) is used 102,441 times. And the circle located at (6, 1093) denotes that the sixth most frequently used method in the evaluation dataset (fastforce) is used 1,093 times in the evaluation dataset. With the use of logarithmic scale on the vertical axis, this figure presents the serious imbalance of proof method invocations occurring in Isabelle's standard library and the AFP.

Fig. 3 summarises the overall performance of PaMpeR. In this figure the horizontal axis represents the number of proof methods PaMpeR is allowed to recommend (15 by default), whereas the vertical axis represents the number of proof methods, for which PaMpeR achieves certain coincidence rates.

For example, the square at (3, 23) means that PaMpeR can achieve 50% of coincidence rate for 23 methods if PaMpeR is allowed to recommend three most promising methods. Similarly, PaMpeR achieves

Table 2: Evaluation of PaMpeR on 15 most frequently used proof methods.

proof method	training	%	evaluation	%	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
simp	102441	26.8	11385	26.8	58	98	99	99	100	100	100	100	100	100	100	100	100	100	100
auto	85097	22.2	9527	22.4	60	94	98	99	100	100	100	100	100	100	100	100	100	100	100
rule	38856	10.2	4150	9.8	3	15	86	99	100	100	100	100	100	100	100	100	100	100	100
blast	23814	6.2	2590	6.1	0	26	26	35	84	95	99	100	100	100	100	100	100	100	100
metis	19771	5.2	2149	5.1	0	0	13	72	84	89	93	96	98	99	100	100	100	100	100
fastforce	9477	2.5	1093	2.6	0	0	0	0	5	54	70	81	89	93	96	96	97	98	98
force	6232	1.6	708	1.7	0	0	0	0	1	9	22	32	40	51	66	77	84	89	94
clarsimp	5984	1.6	628	1.5	0	12	14	14	20	29	39	49	54	57	62	64	66	67	73
cases	5842	1.5	689	1.6	0	0	1	16	16	20	34	54	70	80	86	91	93	95	96
erule	5732	1.5	707	1.7	0	0	15	38	44	53	64	70	76	82	85	87	91	92	93
subst	5655	1.5	619	1.5	0	0	19	19	19	20	22	28	45	58	69	77	82	86	90
rule_tac	5342	1.4	631	1.5	0	14	32	34	44	45	46	47	50	51	52	52	53	57	63
intro	4988	1.3	619	1.5	0	0	5	18	24	39	46	47	48	48	49	57	69	77	84
simp_all	4982	1.3	568	1.3	0	0	0	1	3	6	15	21	26	33	45	60	70	78	83
induct	4884	1.3	568	1.3	0	0	0	1	27	45	49	50	50	51	56	62	71	77	79

50% of coincidence rate for 58 methods when recommending 10 methods and for 72 methods when recommending 15 methods.

The number of methods PaMpeR that achieved the four coincident rates (25%, 50%, 75%, and 90%) reached a plateau when PaMpeR is allowed to recommend about 60 proof methods.

Overall, PaMpeR's recommendations tend to coincide with human engineers' choice when Isabelle has only one method that is suitable for the proof goal at hand, whereas PaMpeR's recommendations tend to differ from human engineers' choice when there are multiple equally valid proof methods for the same goal. For example, PaMpeR's coincidence rates are low for less commonly used general-purpose methods, such as `safe`, `clarimp`, `best`, `bestsimp` because multiple general purpose proof methods can often handle the same proof goal equally well.

A careful observation at the raw evaluation results provided in the Appendix of the technical report reveals that PaMpeR provides valuable recommendations when proof states are best handled by special purpose proof methods, such as `unfold_locales`, `transfer`, `eventually_elim`, `standard`, and so on.

PaMpeR's regression tree construction does not severely suffer from the imbalance among proof method invocation, even though class imbalances often cause problems in other domains such as fraud detection and medical diagnosis [7]. The complete evaluation results in Appendix of the report show that PaMpeR achieved 50% of coincidence rate for 34 proof methods that appear less than 0.1% of times in the training dataset.

The reason the imbalance did not cause serious problems to PaMpeR is that some 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 goals.

On the other hand, PaMpeR did not produce valuable recommendations to some special purpose proof methods, such as `vector` and `normalization`, for which we did not manage to develop assertions that capture the properties shared by the proof goals that these methods can handle well. Writing suitable assertions for these remain as our future work.

Some of the proof methods appearing in our evaluation dataset are clearly outside the scope of PaMpeR. For example, `cartouche`, `tactic`, `ml_tactic`, `rotate_tac` do not have much semantic meaning: `tactic` is simply an interface between Isabelle's source code language, Standard ML, and Isabelle's proof language, Isar, whereas `rotate_tac` simply rotates the order of premises when a proof goal has multiple premises. Another good example of proof methods outside the scope of PaMpeR is the `my_simp` method. This method was defined in the standard library to test the domain specific language, *Eisbach*, for writing new proof methods: `my_simp` is simply a synonym of `simp` and nobody is expected to use `my_simp`. Predicting such methods is not a very meaningful task for PaMpeR.

To our surprise, Table V in Appendix of our technical report [19] shows that PaMpeR's recommendation achieved 50% of coincidence rate for 12 methods out of 38 user-defined proof methods defined outside Isabelle's standard library appearing in the evaluation dataset when PaMpeR is allowed to provide 15 most promising proof methods, even though PaMpeR's developers did not know anything about these proof methods at the time of development. This suggests that one does not need to know the problem specific information about proof goals to predict the use of some user-defined proof methods. For example, PaMpeR achieves 100% of coincidence rate for `sepref` when allowed to recommend only four methods, by checking if the first sub-goal has a schematic variable and if the first sub-goal has variables of type record.

7 DISCUSSION AND FUTURE WORK

Prior to PaMpeR, Isabelle had the `print_methods` command, which merely lists the proof methods defined in the corresponding proof context in alphabetical order ignoring the properties of the proof

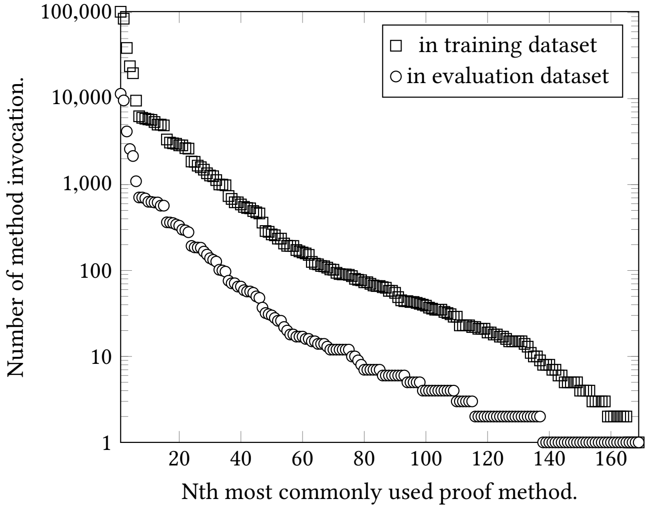

Figure 2: Fig.2: Method usage in large proof corpora.

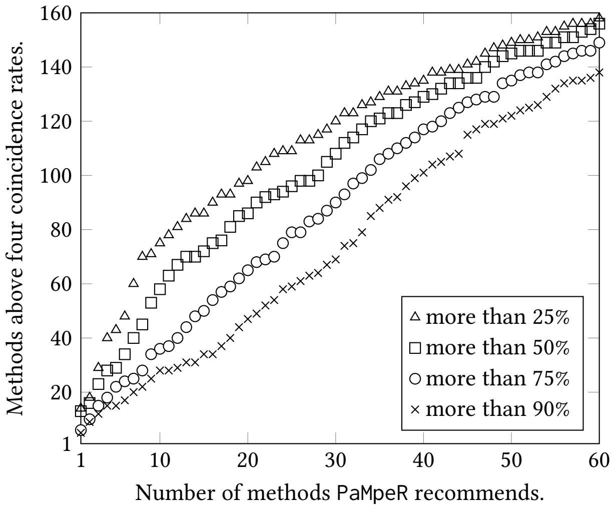

Figure 3: Fig. 3: Coincidence rate for PaMpeR.

goal at hand. Therefore, 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 users. And this problem is becoming severer 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 goals specified by others. PaMpeR addressed 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 without much success [9, 16].

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 with 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 goals 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 attack proof goals 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 essentially deal with different problems in each line of 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 authors work on their own specifications, creating new problems. This results in an important difference between theorem proving in an expressive logic and other machine learning domains, such as image recognition where one can collect numerous instances of similar objects.

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 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 special purpose methods that 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 invite you to use PSL [20], the proof strategy language for Isabelle/HOL, which attempts to find the right combination of arguments via an iterative deepening depth first search based on rough ideas about which method to use. If you want to have those rough ideas, use PaMpeR.

PaMpeR constructs regression trees of a fixed height. We set the height to a small number, three, to avoid over-fitting. It might be possible that advanced pruning methods can improve the accuracy of PaMpeR's recommendation. Furthermore, since we developed 108 assertions based on our limited expertise, it is likely that we have missed out information valuable to recommend proof methods when abstracting proof states using assertions.

Our cross-validation showed that some simple assertions, such as checking the existence of certain constants in proof goals, turned out to be useful. Therefore, it might be possible to find more useful assertions by systematically enumerating more assertions of this kind to check the existence of other constants appearing in proof corpora. Unfortunately, the database construction based on 108 assertions already consumed serious computational resources, and database construction based on generated assertions remains as our future work due to the limitation of resources currently available to PaMpeR's developers.

When conducting the cross-evaluation, we focused on the coincidence rate for each method. It would be worthwhile to compare the results of PaMpeR's overall coincidence rate for all methods with the corresponding overall coincidence rate that would be produced by a naive system that recommends proof methods in order of their frequency in training data without constructing decision trees.

Finally, our choice of machine learning algorithm is not final. We currently use regression tree construction algorithm based on a problem transformation method because the straightforward algorithm lets us produce qualitative explanations of PaMpeR's recommendation; however, other machine learning algorithms might lead to higher coincidence rates. The comparison of various machine learning algorithms on the dataset remains as our future work.

8 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 goal 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 goals proved so far, from which users can infer how to attack the proof goal 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 goals themselves but also their surrounding proof context.

Gauthier *et al.* developed TacticToe for HOL4 [4]. It selects proved lemmas similar to the current proof goal using premise selection and applies tactics used to these similar goals to discharge the current proof goal. Compared to TacticToe, the abstraction via assertions allows PaMpeR to provide valuable recommendations even when similar goals 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 [18] and MaSh [13] decreased the quantity of facts passed to the automatic provers while increasing their quality to improve Sledgehammer's performance. Their approaches attempt to choose facts that are likely to be useful to the given proof goal, 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 goals and available facts, while MaSh computes the relevance using sparse naive Bayes and k Nearest Neighbours. They detect similarities between proof goals 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 goals: 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 goal.

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 [23] 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.

Blanchette *et al.* analysed the AFP, looking at sizes and dependencies for theory files [2]. Matichuk *et al.* investigated the seL4 proofs and two articles in the AFP to find the relationship between the size of statement and the size of proof [17]. None of them analysed the occurrence of proof methods in their target proof corpora nor developed a recommendation system based on their results. Moreover, PaMpeR's database construction is more active compared to their work: it applies 108 hand-written assertions to analyse the properties of not only each proof goal but also the relationship between each goal and its background context and chained facts.

ACKNOWLEDGMENTS

The authors would like to the anonymous referees at ITP2016, ITP2017, and ASE2018 for their valuable comments and helpful suggestions. 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] 2018. PSL with PGT: CICM2018 for Isabelle2017. (2018). https://github.com/data61/PSL/releases/tag/v0.1.1 To use PaMpeR, one first needs to install Isabelle/HOL, which is distributed at https://isabelle.in.tum.de/.

[2] Jasmin Christian Blanchette, Maximilian P. L. Haslbeck, Daniel Matichuk, and Tobias Nipkow. 2015. Mining the Archive of Formal Proofs. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings (Lecture Notes in Computer Science)*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.), Vol. 9150. Springer, 3–17. https://doi.org/10.1007/978-3-319-20615-8_1

[3] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth.

[4] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. 2017. TacticToe: Learning to Reason with HOL4 Tactics. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017 (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 125–143. http://www.easychair.org/publications/paper/340355

[5] Georges Gonthier. 2007. The Four Colour Theorem: Engineering of a Formal Proof. In *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers (Lecture Notes in Computer Science)*, Deepak Kapur (Ed.), Vol. 5081. Springer, 333. https://doi.org/10.1007/978-3-540-87827-8_28

[6] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. 2015. A formal proof of the Kepler conjecture. *CoRR* abs/1501.02155 (2015). arXiv:1501.02155 http://arxiv.org/abs/1501.02155

[7] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

[8] Jónathan Heras and Ekaterina Komendantskaya. 2013. ML4PG: proof-mining in Coq. *CoRR* abs/1302.6421 (2013). arXiv:1302.6421 http://arxiv.org/abs/1302.6421

[9] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2235–2243. http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection

[10] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. [n. d.]. *An Introduction to Statistical Learning.* https://doi.org/10.1007/978-1-4614-7138-7

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

[12] Gerwin Klein, Tobias Nipkow, Larry Paulson, and Rene Thiemann. [n. d.]. . https://www.isa-afp.org/

[13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. 2013. MaSh: Machine Learning for Sledgehammer. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.), Vol. 7998. Springer, 35–50. https://doi.org/10.1007/978-3-642-39634-2_6

[14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[15] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[16] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep Network Guided Proof Search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017 (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 85–105. http://www.easychair.org/publications/paper/340345

[17] Daniel Matichuk, Toby C. Murray, June Andronick, D. Ross Jeffery, Gerwin Klein, and Mark Staples. 2015. Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 722–732. https://doi.org/10.1109/ICSE.2015.85

[18] Jia Meng and Lawrence C. Paulson. 2009. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic* 7, 1 (2009), 41–57. https://doi.org/10.1016/j.jal.2007.07.004

[19] Yutaka Nagashima and Yilun He. 2018. PaMpeR: Proof Method Recommendation System for Isabelle/HOL. *CoRR* abs/1806.07239 (2018). arXiv:1806.07239 http://arxiv.org/abs/1806.07239

[20] Yutaka Nagashima and Ramana Kumar. 2017. A Proof Strategy Language and Proof Script Generation for Isabelle/HOL. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings (Lecture Notes in Computer Science)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 528–545. https://doi.org/10.1007/978-3-319-63046-5_32

[21] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[22] Lawrence C. Paulson. 2015. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J. Autom. Reasoning* 55, 1 (2015), 1–37. https://doi.org/10.1007/s10817-015-9322-8

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. http://dl.acm.org/citation.cfm?id=2078195

[24] Markus Wenzel. 1999. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings (Lecture Notes in Computer Science)*, Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (Eds.), Vol. 1690. Springer, 167–184. https://doi.org/10.1007/3-540-48256-3_12