Machine Learning in Proof General: Interfacing Interfaces

Ekaterina Komendantskaya[*]
School of Computing
University of Dundee, UK
katya@computing.dundee.ac.uk

Jónathan Heras[*]
School of Computing
University of Dundee, UK
jonathanheras@computing.dundee.ac.uk

Gudmund Grov[†]
School of Mathematical & Computer Sciences
Heriot-Watt University, UK
G.Grov@hw.ac.uk

Abstract: We present ML4PG – a machine learning extension for Proof General. It allows users to gather proof statistics related to shapes of goals, sequences of applied tactics, and proof tree structures from the libraries of interactive higher-order proofs written in Coq and SSReflect. The gathered data is clustered using the state-of-the-art machine learning algorithms available in MATLAB and Weka. ML4PG provides automated interfacing between Proof General and MATLAB/Weka. The results of clustering are used by ML4PG to provide proof hints in the process of interactive proof development.
Key words: Interactive Theorem Proving, User Interfaces, Proof General, Coq, SSReflect, Machine Learning, Clustering.

1 Introduction

Over the last few decades, theorem proving has seen major developments. *Automated (first-order) theorem provers (ATPs)* (e.g. E [51], Vampire [49] and SPASS [57]) and SAT/SMT solvers (e.g. CVC3 [5], Yices [20] and Z3 [46]) are becoming increasingly fast and efficient [39]. *Interactive (higher-order) theorem provers (ITPs)* (e.g. Coq [14], Isabelle/HOL [47], Agda [11], Matita [3] and Mizar [23]) have been enriched with dependent types, (co)inductive types, type classes and now provide rich programming environments [21, 25, 35, 52].

The main conceptual difference between ATPs and ITPs lies in the styles of proof development: for ATPs, the proof process is primarily an automatically performed *proof search*, for ITPs – it is mainly user-driven *proof development*. Nevertheless, ITPs have seen major advances in proof automation [22, 27, 43]. One particular trend is to re-enforce proof automation in ITPs by employing state-of-the-art tools from ATPs [1, 43], SAT/SMT solvers [1, 9, 30] or Computer Algebra systems [7, 27, 38]. One major success of this approach is Sledgehammer [48]: it offers Isabelle/HOL users an option to call for an ATP/SMT-generated solution [9].

Integrating ITPs with ATPs requires a lot of research into *methods of interfacing*. Namely, the major challenge is a *sound* and *reliable* translation between inherently different languages [1, 10, 27, 38, 44]. This especially concerns interpreting outputs from ATPs back into the higher-order environment [1, 10, 44], which we will also call here *backward interfacing*. For example, Sledgehammer uses the results provided by external tools to guide the higher-order proofs, but leaves it to the Isabelle/HOL kernel to check that the suggested tactic combination is valid.

[*]The work was supported by EPSRC grant EP/J014222/1.
[†]The work was supported by EPSRC grants EP/H023852/1 and EP/H024204/1.

Submitted to:
UITP 2012

In parallel to the work mentioned above, another trend of research has been developed. It approaches the issue of improving proof automation from the perspective of statistical and machine learning methods. Several aspects of automated and interactive theorem proving can be data-mined:

- proof heuristics can be data-mined to improve proof search in ATPs [16, 17, 34, 37, 41, 53, 55, 56];
- history of successful and unsuccessful proof attempts can be used to inform interactive proof development in ITPs [19, 37].

The former trend has been more successful, mainly directed to improve premise selection in ATPs. In the case of higher-order interactive proofs, there are four main issues that make statistical data-mining challenging:

C.1. The richer tactic language reduces the chance of finding regularities and proof patterns. ITP-based proofs involve an unlimited variety of structures and proof patterns, in comparison to ATPs, where resolution or rewriting may be the two possible rules to apply. Hence, finding statistically significant proof features becomes challenging.

C.2. The notions of a *proof* may be regarded from different perspectives in ITPs: it may be seen as a transition between the subgoals [17, 55, 56], a combination of applied tactics [19], or — more traditionally – a proof tree showing the overall proof strategy [37]. Depending on the nature of the proof and application areas of the machine learning tools, each of the three aspects can be important for statistical proof pattern recognition.

C.3. Backward interfacing – interpreting results provided by the statistical machine learning tool back into the higher-order interactive prover – can be a challenge.

C.4. In interactive proofs, the most time-consuming and challenging part is no longer the time the prover takes to find the proof. It is the time the proof developer takes to understand and guide the proof. Therefore, when data-mining interactive proofs, we are interested not only in the final result – the successful proof, but also in the *proof process*, including failed and discarded derivation steps. We want machine learning to guide the process, not to diagnose or speed up already found proofs. For this, machine learning tools for ITPs need to be interactive.

Up to now, experiments on data-mining interactive proofs were always constrained by the lack of the interactive interfacing between machine learning algorithms and the user-driven proof development. For example, in [34], there was a tool that gathered statistics, but no automated data-mining tools were used; in [37], there was a feature extraction method to data-mine proofs but it was not connected to efficient statistics gathering; in [19], these two were semi-automated.

Because of the inherently interactive nature of proofs in ITPs, user interfaces for ITPs play an important role in proof development. For our experiments, we chose *Proof General* [4] – a general-purpose, emacs-based interface for a range of higher-order theorem provers, e.g. Isabelle, Coq or Lego. Among them, we have chosen Coq [14] and its SSReflect library [22] for our experiments. Although both built upon the same language – *Calculus of Inductive Constructions* [15], they have distinct proof styles, analysis of which plays a special role in this paper, see Section 3.

This idea of maintaining a strong, convenient interface for a range of proof systems is mirrored by a similar trend in the machine learning community. As statistical methods require users to constantly interpret and monitor results computed by the statistical tools, the community has developed uniform interfaces – convenient environments in which the user can choose which machine learning algorithm to use for processing the data and for interpreting results. One such famous interface we take for our experiments is MATLAB [42] which has its own underlying programming language, and comprises

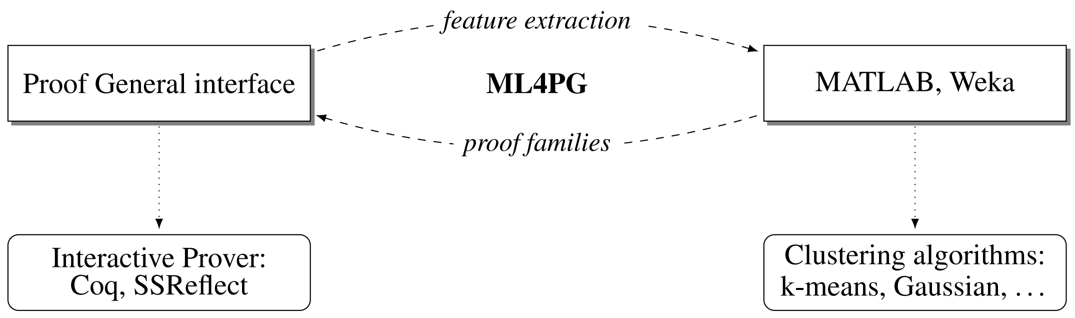

feature extraction
Proof General interface
ML4PG
MATLAB, Weka
proof families
Interactive Prover: Coq, SSReflect
Clustering algorithms: k-means, Gaussian, ...

Figure 1: ***"Interfacing Interfaces" with ML4PG.***

several machine learning toolboxes, from general-purpose *Statistical Toolbox* to the specialised *Neural-network Toolbox*. The second major machine learning interface we explore is Weka [26] – an open source, general purpose interface to run a wide variety of machine learning algorithms.

We have already referred to the two different meanings of the term "interfaces". On the one hand, interfacing may mean translation mechanisms connecting ITPs with other proof automation tools [1, 27, 38, 44]; and on the other hand, it is used as a synonym for user-friendly environment. In this paper, the two views on the notion of interfaces meet. Our primary goal is to integrate the state-of-the-art machine learning technology into ITPs, in order to improve user experience and productivity. However, since machine learning algorithms will need to gather statistics from the user's behaviour, and feed the results back to the user *during* the proof development process, this primary task will never be accomplished without machine learning becoming an integral part of the user interface.

In this paper, we show the results of our work on *interfacing interfaces* – building a user-friendly environment that integrates a range of machine learning tools provided by MATLAB and Weka into Proof General. In particular, we pay attention to addressing the challenges **C.1-C.4**. We implement the following vision of interfacing between ITP and machine learning, and call the result ML4PG (*machine learning for Proof General*), see Figure 1.[1]

1. ML4PG must be able to gather statistics from interactive proofs (challenge **C.3**), and relate this statistics accurately to the three aspects of ITP-based proof development: goal-level, tactic-level, and proof tree level (challenge **C.2**). We focus on this issue in Section 2.

2. ML4PG must automatically extract the relevant features associated with these three aspects in a form suitable for machine learning tools – that is, numerical vectors of fixed length, also known as *feature vectors* (challenges **C.1–C.2**). We present a new method of *proof-trace* feature extraction in Section 3.

3. ML4PG must enable the user to choose from a range of machine learning interfaces and algorithms suitable for proof data-mining (challenge **C.4**). As we do not assume the Proof General user to have machine learning expertise, we want to delegate a substantial amount of pre- and post-processing of statistical results to ML4PG. Section 4 deals with these questions.

4. ML4PG must automatically connect to the chosen machine learning interface, and it should collect, appropriately analyse and interpret the output of these algorithms, at any stage of the interactive proof (challenge **C.3**). Note that in our work "backward interfacing" from the machine learning

[1]It is available at [28], where the reader can download ML4PG, user manual and examples (see also [29]).

Goals	Tactics
$\forall n : nat, 0 = n * 0$	`induction n.`
1. $0 = 0 * 0$; 2. $0 = Sn * 0$	`simpl; trivial.`
$0 = Sn * 0$	`simpl; trivial.`
$\square$	`Qed.`

Goals	Tactics
$\forall l : list\ A, l ++ [] = l$	`induction l.`
1. $[] ++ [] = []$. 2. $(a :: l) ++ [] = a :: l$	`simpl;trivial.`
$(a :: l) ++ [] = a :: l$	`simpl.`
$a :: l ++ [] = a :: l$	`rewrite IHl.`
$a :: l = a :: l$	`trivial.`
$\square$	`Qed.`

Table 3: ***Proof steps for Lemmas*** `mult_n_O` $: \forall n : nat, 0 = n * 0$ ***and*** `app_l_nil`: $\forall l : list\ A, l ++ [] = l$.

tools to Proof General is less demanding compared to [1, 27, 38, 44]. We do not seek a translation of statistical results into the Coq *language* – instead, we use the statistical results to inform the user of arising proof patterns during the proof development. As Sections 4 and 5 show, this kind of light backward interfacing can be efficiently implemented.

5. Finally, ML4PG must interact with the user by providing relevant information about the user's current proof goal in relation to statistically similar proof patterns detected in different libraries or even across different users (challenges **C.1–C.4**). We discuss this in Sections 4 and 5.

There are two aspects to this work: development of methods of interactive interfacing between the ITP and machine learning interfaces; and a more general aspect of studying the potential of machine-learning methods in proof-pattern recognition. This paper mainly focuses on the the first aspect. References [28, 29, 41] are specifically devoted to the benchmarks, evaluation and discussion of statistical proof-pattern recognition methods in theorem proving. As far as ML4PG interface engineering goes, it was important for us to make accessible a number of simple but useful options that the user with no experience in machine learning could use. Section 6 surveys related work on integration of machine learning with theorem proving. Finally, in Section 7, we conclude and discuss future extensions.

2 The Three Levels of an Interactive Proof

In this section, we consider a variety of possible approaches to proof pattern recognition in ITPs; namely, we consider automated proofs from the levels of goal transitions, tactic sequences, and proof trees.

We start with several running examples to illustrate the kind of statistical help we expect from ML4PG. We consider the library containing various lemmas about natural numbers and lists.

Example 2 Suppose the user starts with the following two lemmas about multiplication by 0: `Lemma mult_n_O`$: \forall n : nat, 0 = n * 0$ and `Lemma mult_O_n`: $\forall n : nat, 0 = 0 * n$, see left side of Tables 3 and 4 for their proofs. They state two very similar properties, however, the proofs for them are different; notably, one proof involved induction, while another involved only simplification.

Next, suppose the user switches to the library containing lists; and needs some guidance to proceed with the proofs for `Lemma app_l_nil`: $\forall l : list\ A, l ++ [] = l$, and `Lemma app_nil_l`: $\forall l : list\ A, [] ++ l = l$. The user asks ML4PG to "statistically match" these problems to previously seen proofs in the same or in a different library. We then want ML4PG to tell the user that there are two similar lemmas in the `Nat`

Goals	Tactics	Goals	Tactics
$\forall n : nat, 0 = 0 * n$		$\forall l : list\ A, [] + +l = l$	
	`intro.`		`intro l`
$0 = 0 * n$		$[] + +l = l$	
	`simpl; trivial.`		`simpl;trivial.`
$\square$		$\square$	
	`Qed.`		`Qed.`

Table 4: ***Proof steps for Lemmas*** `mult_0_n`$: \forall n : nat, 0 = 0 * n$ ***and*** `app_nil_l`: $\forall l : list\ A, [] + +l = l$.

library – namely `Lemma mult_n_0`:$\forall n : nat, 0 = n * 0$ and `Lemma mult_0_n`: $\forall n : nat, 0 = 0 * n$. Then the user will adapt these old proofs to complete new proofs as given on the right side of Tables 3 and 4. Note that this guidance will go further than just identifying proofs over the same data type, identifying same tactic combinations, same functions/operations or similar lemma shapes. Such guidance would be based on statistical correlation of several proof features.

As can be seen from these examples, the user may be interested in data-mining the proofs based on either

$\star$ transitions between subgoal-shapes (in which case Lemmas `app_l_nil` and `mult_n_0` of Table 3 have common patterns), or

$\star\star$ statistics of tactic combination (in which case Lemmas `app_nil_l` and `mult_0_n` of Table 4 should be identified), or

$\star\star\star$ a more general understanding of lemma content (in which case all four are similar).

Therefore, we distinguish three levels at which pattern-recognition in ITP proofs can be approached, see also [24]:

1. *Goal-pattern recognition.* Sequences of subgoals may show an apparent pattern in the structure of the formulas. This type of feature abstraction has been used for learning the inputs for automatic provers [17, 56] – which has later been extended to interactive proofs [53].

   Example 5 The left-most columns of Tables 3 and 4 should be used to gather such information about goal-patterns.

2. *Tactic-pattern recognition.* Sequences of tactics applied at every level of the proof bear some apparent patterns, as well. There is always a finite number of tactics for any given proof, and therefore, they can serve as features for statistical learning. Previous work on learning proof strategies [19, 31, 32] has taken this approach. It is important to note that there may be proofs in which the goal structures do not bear any evident pattern; however, the sequence of applied tactics does. Also, as an additional complication, there is a variety of tactic combinations that may lead to a successful proof for one goal; and conversely, different goals may yield same sequences of tactics in successful proofs. Moreover, tactics often have complex configurations, which can be hidden or given as arguments (e.g. rules to apply or instantiations of variables).

   Example 6 The right-most columns of Tables 3 and 4 provide information about such tactic-patterns.

   The disadvantage of tactic-pattern recognition is that any knowledge of when and why a tactic is applied, as well as its result is lost (except with respect to other tactic applications).

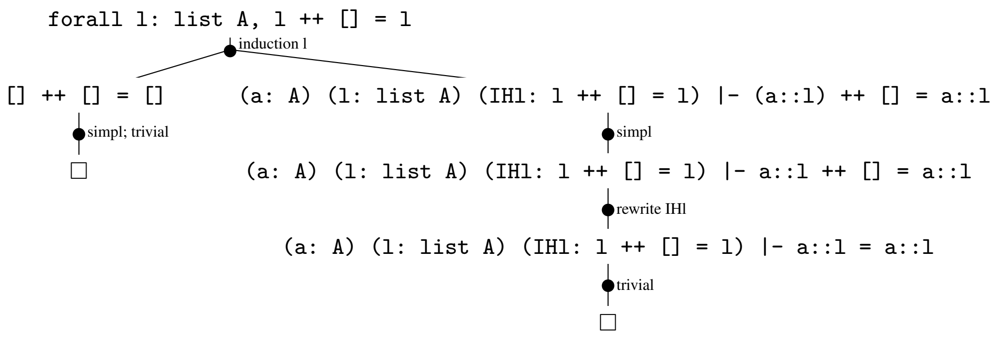

forall l: list A, l ++ [] = l
induction l
[] ++ [] = []
(a: A) (l: list A) (IHl: l ++ [] = l) |- (a::l) ++ [] = a::l
simpl; trivial
simpl
(a: A) (l: list A) (IHl: l ++ [] = l) |- a::l ++ [] = a::l
rewrite IHl
(a: A) (l: list A) (IHl: l ++ [] = l) |- a::l = a::l
trivial
□
□

Figure 7: ***Proof tree for*** *app_l_nil.*

3. *Proof tree pattern recognition.* Finally, there is the level of a proof tree – that shows relations between different proof branches and subgoals and gives a better view of the overall proof flow; this approach was tested in [37] using multi-layer neural networks and kernels.

   Example 8 Figure 7 shows the proof tree for `app_l_nil`. An advantage of the proof tree as opposed to goal or tactic sequence, is that it distinguishes between different proof branches.

Our second running example is based on the `bigop` library of SSReflect. This library is devoted to generic indexed big operations, like $\sum_{i=0}^{n} f(i)$ or $\bigcup_{i \in I} f(i)$.

Example 9 We take three lemmas about number series:

$$\forall n, 2(\sum_{i=0}^{n} i) = n(n+1); \quad \forall n, \sum_{i=0|odd\ i}^{2n} i = n^2; \quad \forall n, \prod_{1}^{n} i = n!$$

The proofs of these three lemmas, both at the level of goals and tactics, are given in Table 10. Intuitively, they show certain similarities and dissimilarities, both at the level of goals and tactics. In the next sections, we will test how ML4PG analyses such cases.

Next, we study how these general considerations about the levels of proof patterns are used in ML4PG to extract features used in statistical data-mining.

3 Feature Extraction in ML4PG: the Proof Trace Method

In this section, we explain algorithms used by ML4PG to gather proof statistics at the levels of goals, tactics, and proof trees.

The discovery of statistically significant features in data is a research area of its own in machine learning, known as *feature extraction*, see [8]. Irrespective of the particular feature-extraction algorithm used, most pattern-recognition tools will require that the number of selected features is limited and fixed.[2] We design our own method of proof feature extraction. The major challenge is to respect the above

[2] *"Sparse methods"* of machine-learning is an exception to this rule, see [40, 41]. We discuss the issue in Section 6.

Goal	Tactic
$2(\sum_{i=0}^{n} i) = n(n+1)$	`elim : n.`
$2(\sum_{i=0}^{0} i) = 0 \times 1$	`by rewrite mul0n big_nat1 muln0.`
$\forall n, 2(\sum_{i=0}^{n} i) = n(n+1) \implies 2(\sum_{i=0}^{n+1} i) = (n+1)(n+2)$	`move => n IH.`
$2(\sum_{i=0}^{n+1} i) = (n+1)(n+2)$	`by rewrite big_nat_recr mulnDr IH -mulnDl addn2 mulnC.`
$\square$	`Qed.`

Goal	Tactic
$\sum_{i=0 \mid odd\ i}^{2n} i = n^2$	`elim : n.`
$\sum_{i=0 \mid odd\ i}^{2\times 0} i = 0^2$	`rewrite exp0n // /index_iota subn0 big1_seq //.`
$\forall i \in \mathbb{N}, odd\ i\ \&\&\ (i \in iota\ 0\ (2\times 0)) \implies i = 0$	`by move => i; move/andP => [_ H2]; move : H2; rewrite muln0 in_nil.`
$\forall n, \sum_{i=0 \mid odd\ i}^{2n} i = n^2 \implies \sum_{i=0 \mid odd\ i}^{2(n+1)} i = (n+1)^2$	`move => n IH.`
$\sum_{i=0 \mid odd\ i}^{2(n+1)} i = (n+1)^2$	`by rewrite big_mkcond -[n.+1]addn1 mulnDr muln1 addn2 !big_nat_recr IH odd2n odd2n1 //= addn0 n1square n2square.`
$\square$	`Qed.`

Goal	Tactic
$\prod_{1}^{n} i = n!$	`elim : n.`
$\prod_1^1 i = 0!$	`by rewrite big_nil.`
$\forall n, \prod_{1}^{n} i = n! \implies \prod_{1}^{n+1} i = (n+1)!$	`move => n IH.`
$\prod_{1}^{n+1} i = (n+1)!$	`by rewrite factS big_add1 -IH big_add1 big_nat_recr mulnC.`
$\square$	`Qed.`

Table 10: ***Interactive Proofs for*** `Lemma sum_first_n`: $2(\sum_{i=0}^{n} i) = n(n+1)$***;*** `Lemma sum_first_n_odd`: $\sum_{i=0 \mid odd\ i}^{2n} i = n^2$***; and*** `Lemma fact_prod`: $\prod_{1}^{n} i = n!$.

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*n subgoals*
g1	**induction**	**1**	**nat**	**no**	**forall**	**2**
g2	**simpl;trivial**	**2**	**none**	**no**	**equal**	**0**
g3	`simpl;trivial`	2	**none**	**no**	**equal**	0
g4	-	-	-	-	-	-
g5	-	-	-	-	-	-

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*n subgoals*
g1	**induction**	**1**	**list**	**no**	**forall**	**2**
g2	**simpl;trivial**	**2**	**none**	**no**	**equal**	**0**
g3	`simpl`	1	**none**	**no**	**equal**	1
g4	`rewrite`	1	Prop	IH	`equal`	1
g5	`trivial`	1	**none**	**no**	`equal`	**0**

Table 12: ***Goal-level feature extraction tables.*** *Parameters inside the double lines are the extracted features. Notation* g1–g5 *is used to denote five consecutive subgoals in the derivation. Columns are the properties of subgoals the feature extraction method of ML4PG will track: names of applied tactics, their number, arguments, link of the proof step to a hypothesis (Hyp), inductive hypothesis (IH) or a library lemma; top symbol of the current goal, and the number of the generated subgoals. The two tables show a comparison of the goal-level feature tables for `mult_n_O` and `app_l_nil`. Intuitively, 18 out of 30 features in these tables show correlation; we highlight them in bold.*

restriction while allowing to data-mine potentially unlimited variety of different higher-order formulas and proofs.

We first focus on the level of goals. ML4PG must choose the relevant features for statistical analysis. At this level, we could consider general goal properties such as "goal shape" (e.g. "associative-shape" or "commutative-shape"), or properties like "the goal embeds a hypothesis", "the goal is embedded into a hypothesis". However, gathering such features uniformly across any set of higher-order proofs would be hard, especially when working with richer theories and dependent types.

Example 11 Consider the proof for `app_l_nil` given in Table 4. One could say that the valuable information about the shape of the (sub)goal 4 is that it embeds the inductive hypothesis, as this fact is later used in the proof. However, for more complex examples, deciding such embeddings unambiguously during feature-extraction may be difficult, see [6]. Finally, detecting a fixed number of properties like e.g. "commutativity" may apply to one type of proof libraries, e.g. natural numbers, but not to others, e.g. lists, in which case uniform comparison of proof patterns across libraries becomes hard.

This is why we developed a method of implicit tracking of proof properties, called the *proof trace method*; its early variant was used in [37]. The idea is as follows. When direct feature-extraction of the goal shapes is infeasible, we still can infer some properties of the goals when gathering statistics of how the user treats the goals. In other words, we let the term-structure show itself through the proof steps it induces. We deliberately do not pre-define the types of *proof patterns* that ML4PG must recognise, or define what a *correlation of proof features* is. Instead, we want statistical machine learning tools to suggest the user what these might be. An advantage of such proof feature extraction method is that it applies uniformly to any Coq library, and does not require any adaptation when ML4PG changes the libraries.

Another important feature ML4PG must be sensitive to, is the long-lasting effect of one proof step on several consecutive proof steps. The dependency between subgoals very often extends much farther than from one proof step to its immediate successor. Thus, we want to capture two dimensions of goal transformation in a proof:

1. various traceable properties of a single (sub)-goal;

	tactics	*N tactics*	*arg type*	*arg is hyp?*	*top symbol*	*n subgoals*
g1	elim	1	nat	Hyp	equal	2
g2	rewrite	1	3×Prop	EL1	equal	0
g3	move =>	1	nat,Prop	no	forall	1
g4	rewrite	1	6×Prop	EL2	equal	0
g5	-	-	-	-	-	-

	tactics	*N tactics*	*arg type*	*arg is hyp?*	*top symbol*	*n subgoals*
g1	**elim**	**1**	**nat**	**Hyp**	**equal**	**2**
g2	**rewrite**	**1**	4×Prop	EL3	**equal**	1
g3	move=>,move/ move:, rewrite	4	nat, 5×Prop	EL4	forall	0
g4	move =>	**1**	nat,Prop	no	forall	1
g5	rewrite	1	11×Prop	EL5	equal	**0**

	tactics	*N tactics*	*arg type*	*arg is hyp?*	*top symbol*	*n subgoals*
g1	**elim**	**1**	**nat**	**Hyp**	**equal**	**2**
g2	**rewrite**	**1**	Prop	big_nil	**equal**	**0**
g3	**move =>**	**1**	**nat,Prop**	**no**	**forall**	**1**
g4	**rewrite**	**1**	**6×Prop**	EL6	**equal**	**0**
g5	**-**	**-**	**-**	**-**	**-**	**-**

Table 15: ***Goal-level feature extraction tables for lemmas*** `sum_first_n`, `sum_first_n_odd` ***and*** `fact_prod`. *In the tables of* `sum_first_n_odd` *and* `fact_prod`. *We use the following notation to note when ML4PG gathers the lemma names: EL*1 *stands for (*`mul0n`, `big_nat1` *and* `muln0`*), EL*2 *– for (*`big_nat_recr`, `mulnDr`, `IH`, `mulnD1`, `addn2` *and* `mulnC`*), EL*3 *– for (*`exp0n`, `index_iota` `subn0` *and* `big1_seq`*), EL*4 *– for (*`/andP`, `muln0` *and* `in_nil`*), EL*5 *– for (*`big_mkcond`, `addn1` *,* `mulnDr`, `muln1`, `addn2`, `big_nat_recr`, `IH`, `odd2n`, `odd2n1`, `addn0`, `n1square` *and* `n2square`*) and EL*6 *– for (*`factS`, `big_add1`, `IH`, `big_add1`, `big_nat_recr` *and* `mulnC`*); the lemmas and libraries can be found in [28]. We highlight in bold the correlation with the features of* `sum_first_n`. *There is a strong correlation between* `sum_first_n` *and* `fact_prod` *(27 out of 30 features); on the contrary, there is a weak correlation between* `sum_first_n` *and* `sum_first_n_odd` *(11 out of 30 features).*

2. transformations of each such property throughout several proof steps.

This is why, we design two dimensional arrays as shown in Table 12 to allow for statistical data-mining of the two dimensions in their relation.

Example 13 Consider Table 12 where the correlation between `mult_n_O` and `app_l_nil` at the goal level is shown. If we consider the table associated with `app_l_nil`, the fact of using the tactics `induction` and `(simpl;trivial)` may not be significant, as this combination can be applied to a variety of goals. It may be insignificant, on its own, that the top symbol of the goal was the quantifier $\forall$. However, the table related to this lemma allows us to characterise the goal $\forall$ `l : list A, []++l=l` by the 30 features (entries) of the table. Correlations of values of these features will be more likely to show significant proof patterns, if such exist.

Example 14 As can be seen in Table 15, there is a strong correlation between the features associated with the first step in the proofs of `sum_first_n`, `sum_first_n_odd` and `fact_prod`. However, this strong correlation only remains between `sum_first_n` and `fact_prod` when successive proof steps are considered. This illustrates the fact that we cannot focus just on the first goal of a proof, but we have to study its proof trace to obtain relevant patterns.

Another advantage of the method is its focus on user interaction: ML4PG learns proof patterns specific to the user's proof style as given in the chosen library of proofs.

	arg-1 type	*rest arg types*	*arg is hyp?*	*top symbol*	*n times used*
`intro`	list	**none**	**no**	**forall**	**1**
`case`	**-**	**-**	**-**	**-**	**-**
`simpl`	**none**	**none**	**no**	**equal**	**1**
`trivial`	**none**	**none**	**no**	**equal**	**1**
	arg-1 type	*rest arg types*	*arg is hyp?*	*top symbol*	*n times used*
`intro`	nat	**none**	**no**	**forall**	**1**
`case`	**-**	**-**	**-**	**-**	**-**
`simpl`	**none**	**none**	**no**	**equal**	**1**
`trivial`	**none**	**none**	**no**	**equal**	**1**

Table 17: ***Tactic-level feature extraction table for Coq.*** *The rows are the tactics implemented in the given ITP. The columns of this table encode the same properties for both plain Coq proofs and proofs in SSReflect: the type of the first argument, a number that denotes the types of the remaining tactic arguments; identification whether the arguments of the tactic are hypotheses (Hyp), inductive hypotheses (IH), external lemmas or none of them. Finally, ML4PG populates the last two columns with the list of top symbols of the goals where the tactic has been applied; and the number of times the tactic was applied. Tactic-level feature extraction tables for* `mult_O_n` *and* `app_nil_l` *show close correlation (in bold).*

On the tactic-level, ML4PG focuses on features associated with each tactic applied in a proof script. The action of the tactic-level feature extraction algorithm is shown in Table 17. It is worth noting that the structure of the goal-level Table 12 can be reused in all the systems based on the application of a sequence of tactics (e.g. Coq, Isabelle/HOL, Matita, etc.); the only difference would be the values which populate the table. On the other hand, the structure of the tactic-level table depends on the concrete system, since each ITP has its concrete set of tactics.

The case of Coq is special, since we can find two proof styles: plain Coq and SSReflect. Although SSReflect is an extension of Coq, this package implements a set of proof tactics designed to support the extensive use of small-scale reflection in formal proofs [22]. In addition, the behaviour of some Coq tactics has been modified (for instance, the `rewrite` tactic); so, the SSReflect imposes a distinct proof style. ML4PG works with both styles of Coq proofs. In the case of plain Coq, the rows of the tactic table represent the main Coq tactics (from almost 100 Coq tactics ML4PG currently distinguishes 10 most popular). The set of SSReflect tactics consists of less than 10 tactics, so we have included all of them.

Example 16 Consider the fragments of tactic-level tables associated with the Lemma `app_nil_l` and `mult_O_n`, in Table 17. The extracted features show close correlation, as expected.

Example 18 Fragments of tactic tables for Lemmas `sum_first_n`, `sum_first_n_odd` and `fact_prod` are given in Table 19. There is a strong correlation between `sum_first_n` and `fact_prod` at this level.

Finally, ML4PG can extract the tree-level features, see Table 21. Currently, it considers the proof flow using up to the depth 5 of the proof tree.

Example 20 The tables for Lemmas `sum_first_n`, `sum_first_n_odd` and `fact_prod` at the proof tree level are given in Table 21.

The feature extraction procedures explained in this section run in the background of ML4PG during Coq compilation. Some of the features are obtained just by inspecting the names and numbers of the applied tactics. In other cases, ML4PG needs to internally invoke Coq compiler to obtain the features, for instance, when recording types of tactic arguments. Thus, statistics related to the three proof levels is automatically gathered during the proof development.

	arg-1 type	*rest arg types*	*arg is hyp?*	*top symbol*	*n times used*
`move =>`	nat	Prop	None	forall	1
`move :`	-	-	-	-	-
`move/`	-	-	-	-	-
`rewrite`	Prop	8×Prop	EL'	2× equal	2
`case`	-	-	-	-	-
`elim`	nat	-	Hyp	equal	1

	arg-1 type	*rest arg types*	*arg is hyp?*	*top symbol*	*n times used*
`move =>`	**nat**	nat,Prop	**None**	2×forall	2
`move :`	Prop	**None**	Hyp	forall	1
`move/`	Prop	**None**	andP	forall	1
`rewrite`	**Prop**	17×Prop	EL"	equal, forall,equal	3
`case`	**-**	**-**	**-**	**-**	**-**
`elim`	**nat**	**-**	**Hyp**	**forall**	**1**

	arg-1 type	*rest arg types*	*arg is hyp?*	*top symbol*	*n times used*
`move =>`	**nat**	**Prop**	**None**	**forall**	**1**
`move :`	**-**	**-**	**-**	**-**	**-**
`move/`	**-**	**-**	**-**	**-**	**-**
`rewrite`	**Prop**	6×Prop	EL"'	**2×equal**	**2**
`case`	**-**	**-**	**-**	**-**	**-**
`elim`	**nat**	**-**	**Hyp**	**equal**	**1**

Table 19: ***Tactic-level feature extraction table for*** `sum_first_n`, `sum_first_n_odd` ***and*** `fact_prod` ***in SSReflect****, showing correlation in bold: 15 out of 30 between* `sum_first_n` *and* `sum_first_n_odd`*; and 28 out of 30 between* `sum_first_n` *and* `fact_prod`*. Where we use notation EL', EL" and EL"', ML4PG gathers respectively the lemma names: (*`mul0n`*,* `big_nat1`*,* `muln0`*,* `big_nat_recr`*,* `mulnDr`*,* `IH`*,* `mulnD1`*,* `addn2` *and* `mulnC`*), (*`exp0n`*,* `/index_iota`*,* `subn0`*,* `big1_seq`*,* `muln0`*,* `in_nil`*,* `big_mkcond`*,* `addn1`*,* `mulnDr`*,* `muln1`*,* `addn2`*,* `big_nat_recr`*,* `IH`*,* `odd2n`*,* `odd2n1`*,* `addn0`*,* `n1square` *and* `n2square`*) and (*`big_nil`*,* `factS`*,* `big_add1`*,* `IH`*,* `big_add1`*,* `big_nat_recr` *and* `mulnC`*); the lemmas and libraries can be found in [28].*

	`move =>`	`move :`	`move/`	`rewrite`	`case`	`elim`	●	□
td1	-	-	-	-	-	nat	12	0
td2	nat, Prop	-	-	EL'	-	-	201	1
td3	-	-	-	EL"	-	-	30	1

	`move =>`	`move :`	`move/`	`rewrite`	`case`	`elim`	●	□
td1	**-**	**-**	**-**	**-**	**-**	**nat**	**12**	**0**
td2	**nat, Prop**	**-**	**-**	big_nil	**-**	**-**	**201**	**1**
td3	**-**	**-**	**-**	EL"'	**-**	**-**	**30**	**1**

Table 21: ***Proof tree level feature extraction table for SSReflect.*** *The rows, marked as* td1–td5 *represent tree levels. The values of columns depend on the tactics, and for each tactic, a different parameter is tracked: for tactics* `case` *and* `elim` *– it is the type of argument; for tactic* `rewrite` *– whether the rewriting rule is a hypothesis, inductive hypothesis, or external lemma. Where we use notation EL', EL" and EL"', ML4PG gathers respectively the lemma names: (*`mul0n`*,* `big_nat1` *and* `muln0`*), (*`big_nat_recr`*,* `mulnDr`*,* `IH`*,* `mulnDl`*,* `addn2` *and* `mulnC`*) and (*`factS`*,* `big_add1`*,* `IH`*,* `big_add1`*,* `big_nat_recr` *and* `mulnC`*); the lemmas and libraries can be found in [28]. In the ● column ML4PG stores the branching factor of the proof tree. It encodes this information globally using the following convention. The first number represents the tree depth level, so, if we are at level* 1 *the first number will be* 1*, at level* 2 *the first number will be* 2 *and so on. Then for each one of the branches of the level we store the number of its subbranches. The □ column indicates the number of proof branches closed at the given level. These tables show a fragment of proof tree level feature table for* `sum_first_n` *and* `fact_prod`*. We highlight in bold the corresponding features between* `sum_first_n` *and* `fact_prod` *(22 out of 24 features).*

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*n subgoals*
g1	3	1	-2	1	6	2
g2	7	1	-444	126106	6	1
g3	1517	4	-244444	112113105176	5	0
g4	1	1	-24	0	5	1
g5	7	1	-44444444444	25484634152437143325432	6	0

Table 23: ***Numerical goal-level feature extraction table for*** `sum_first_n_odd`.

Machine learning algorithms expect numerical feature vectors as inputs; therefore, ML4PG converts the features into numbers. As we explained in [37], the concrete function that ML4PG uses for this purpose may vary, but the numeric conversion must be consistent. Dynamic calculation of the function that converts table features into numbers is implemented in ML4PG. In particular, we have defined 4 one-to-one functions $[[\ .\ ]]_{Tactic}, [[\ .\ ]]_{Type}, [[\ .\ ]]_{Top\_symbol}$ and $[[\ .\ ]]_{hyp\_or\_lemma}$ that assign respectively a numerical value to each tactic, type, top symbol and lemma appearing in a proof. The conversion provided by these functions is blind, assigning respectively unique consecutive integers to tactics, types, top symbols and lemmas in order of their appearance in the library. If several elements appear in a cell, the value of that cell is the concatenation of the values of each element.

Example 22 The numerical table for Lemma `sum_first_n_odd` at the goal level is given in Table 23. This table is flattened into a vector to be given as input to machine learning algorithms. Namely, Table 23 gives rise to the following feature vector:
$[3,1,-2,1,6,2,7,1,-444,126106,6,1,1517,4,-244444,112113105176,5,0,1,1,-24,0,5,1,7,1,$
$-44444444444,25484634152437143325432,6,0]$.

Once the feature vectors are collected, ML4PG can data-mine the proofs using different machine learning algorithms.

4 Interactive Proof-Clustering in ML4PG

ML4PG is designed to prove the concept: *it is possible to interface higher-order proofs with machine learning engines, and do it interactively during the proof process*. Interaction with several machine learning engines and algorithms is in the core of this process. This differs from the experiments performed in the literature, see [19,37,41,56], where the data-mining of proofs is performed separately from the user interface. In this section, we explain how ML4PG enables the user interaction with a range of machine learning engines and algorithms; and give some technical details of the ML4PG implementation.

The ML4PG user may or may not be familiar with machine learning. Either way, ML4PG must offer him a number of simple but useful options to configure machine learning tools while staying within the Proof General environment. Therefore, ML4PG takes the burden of connecting to the machine learning algorithms.

The first choice the user makes concerns the *proof level*: proofs can be data-mined at the level of goals, tactics or proof trees, as explained in the previous sections. It is worth mentioning here that there are several choices of how to run this feature extraction. One option would be to extract features on demand – that is, once the user chooses the proof level, ML4PG could re-run Coq again to complete the feature extraction. The disadvantage of this is that the proof engine will have to be re-run every time one uses ML4PG for data-mining. We made a different choice: ML4PG extracts features in the background during the interactive proofs. It does the extraction at all three proof levels whenever the proof library is compiled. Then, the choice of proof level in the menu just indicates which data set will be sent to the machine learning algorithms. The advantage is that the time taken by data mining does not include the proof engine run. Our experiments show that the time involved in the feature extraction during the normal Coq compilation is unnoticeable, and does not significantly slow down the proof development.

Now ML4PG is ready to communicate with *machine learning interfaces*. ML4PG is built to be modular – that is, when the feature extraction of Section 3 is completed within the emacs environment, the data is gathered in the format of hash tables. The first elements of these tables are the names of the lemmas, and the second elements are the feature vectors encoded as lists of numbers (let us note that emacs is a Lisp environment; therefore, it is sensible to use lists to represent the feature vectors). However, every machine learning engine has its concrete format to represent feature vectors; therefore, it is necessary to define *translators* to adapt ML4PG's internal encoding of feature vectors to the concrete representation of the machine learning engine. We have defined translators for two different, but equally popular, machine learning interfaces – MATLAB and Weka. ML4PG transforms the feature vectors to a *comma separated values* (csv) file in the case of MATLAB; and, to *arff* files in the case of Weka. In principle, extending the list of machine learning engines does not require any further modifications to the feature extraction algorithm, but just defining new *translators*. Notice the similarity with implementation of the proof level choice: again, once the features are extracted, the ML4PG engine is flexible to use them for all sorts of data mining tasks and machine learning interfaces.

Once the feature vectors are in a suitable format, ML4PG can invoke the machine learning engine. The ML4PG mechanism connecting to machine learning interfaces is similar to the native mechanism of Proof General used to connect to ITPs. Namely, there is a synchronous communication between ML4PG and the machine learning interfaces, which run on the background waiting for ML4PG calls.

The next configuration option ML4PG offers is the choice of the particular *pattern-recognition algorithm* available from the chosen machine learning interface. Again, this choice is made within the proof environment of Proof General. There are several machine learning algorithms available in MATLAB and Weka. We connected ML4PG only to *clustering algorithms* [8] – a family of *unsupervised learning methods*. Unsupervised learning is chosen when no user guidance or class tags are given to the algorithm in advance.

One could in principle envisage *supervised machine learning applications* in proof pattern recognition, where the user labels every proof using some finite tags, such as "fundamental lemma", "auxiliary lemma", "proof experiment". And, on the basis of such labels and some number of training examples, the machine learning algorithm would be able to predict labels for any new proof. Here, we do not assume existence of such labels. However, our modular approach to interfacing with Proof General implies that, if the labels are available, interfacing with supervised algorithms will not be hard for ML4PG. In fact, we envisage the feature extraction method to remain the same. Section 6 discusses related work using supervised learning in proof-pattern recognition.

In case of MATLAB, the algorithms included in ML4PG are the two most popular methods for clustering: K-means and Gaussian [58]. If the user selects Weka as a machine learning engine; then, he can select among K-means, FarthestFirst and simple Expectation Maximisation.

To improve the accuracy of the clustering algorithms, a technique called *Principal Components Analysis* (PCA) [33] is applied. This functionality reduces the size of feature vectors but without much loss of information. The application of techniques like PCA, known in general as *dimensionality reduction procedures* [58], is recommended when dealing with feature vectors whose size is higher than 15 – as in our case.

Finally, the user can choose *proof libraries* that he wants to access using ML4PG. Before using them, those libraries must be exported with the mechanism provided by ML4PG. ML4PG extends the compilation procedure that Coq uses for imported libraries with the feature extraction algorithm described above. Such a mechanism checks that all the proofs of the library are finished, and generates a file which contains the list of the lemmas of the library, and three files encoding respectively the feature vectors at the goal level, tactic level and proof tree level. Subsequently, when the user chooses a library, ML4PG transforms the files to the internal encoding of feature vectors (implemented by Lisp lists) and attaches those vectors to those obtained in the current development.

By default, ML4PG clusters the current library, but the user can add more libraries to perform clustering. The reason for not using all the available libraries is twofold. First of all, it is a question of *performance*, since the time needed to obtain clusters increases with the amount of libraries. The second reason is *usability*, because if ML4PG uses all the available libraries for clustering, it can obtain proof patterns from lemmas which belong to libraries unknown to the user, and this may or may not be convenient.

We now return to the examples introduced in Section 2 to illustrate how proof patterns are shown to the ML4PG user.

Example 24 We created a small library (70 lemmas) to help us with the initial tests of ML4PG: it contains some basic lemmas about natural numbers and lists, as well as our running examples of Tables 3 and 4. We also included efficient and inefficient proofs, and cases when similar lemmas were proven using different strategies, and different lemmas were proven using the same proof strategy. In the rest of the paper, we will call the library *`Initial`*. Figure 25 shows the result of running ML4PG on this library, with the following settings:

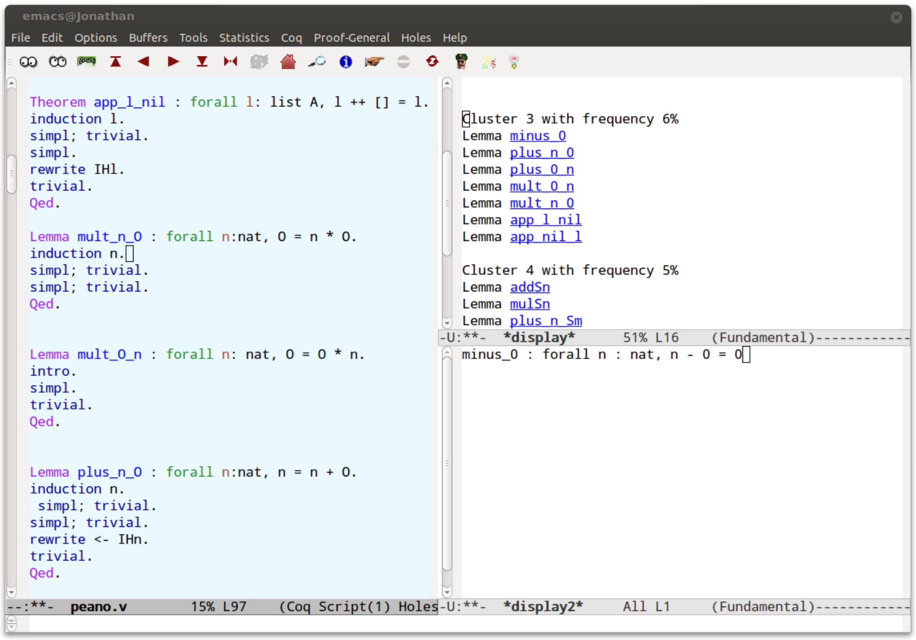
- statistics were taken using goal-level feature extraction;
- machine learning interface: Weka;
- machine learning algorithm: K-means.

ML4PG shows that all lemmas of Tables 3 and 4 belong to the same group of proofs. It agrees with one possible interpretation of the content of these lemmas, see $\star\star\star$ in Section 2. But there are other lemmas in that cluster; in particular, this cluster gathers “fundamental” lemmas about various operations on natural numbers involving `0` and operations on lists involving `nil`.

The example above shows one mode of working with ML4PG: that is, when a library is clustered irrespective of the current proof goal. However, it may be useful to use this technology to aid the interactive proof development. In which case, we can cluster libraries relative to a few initial proof steps for the current proof goal. Note that ML4PG graphical interface offers two menu buttons for these two options – the two right-most buttons of Figure 25. The next example illustrates this.

Example 26 On the left side of Figure 27, an incomplete development of Lemma `sum_first_n` is shown. Using the `bigop` and `binomial` libraries of SSReflect (205 lemmas), ML4PG can obtain proof patterns similar to the lemma that we are proving, see right side of Figure 27. The ML4PG settings in this case were:

- statistics was taken using goal-level feature extraction;
- machine learning interface: MATLAB;
- machine learning algorithm: K-means.


emacs@Jonathan
File Edit Options Buffers Tools Statistics Coq Proof-General Holes Help

```
Theorem app_l_nil : forall l: list A, l ++ [] = l.
induction l.
simpl; trivial.
simpl.
rewrite IHl.
trivial.
Qed.

Lemma mult_n_O : forall n:nat, O = n * O.
induction n.
simpl; trivial.
simpl; trivial.
Qed.


Lemma mult_O_n : forall n: nat, O = O * n.
intro.
simpl.
trivial.
Qed.


Lemma plus_n_O : forall n:nat, n = n + O.
induction n.
 simpl; trivial.
simpl; trivial.
rewrite <- IHn.
trivial.
Qed.
--:**-  peano.v        15% L97     (Coq Script(1) Holes
```

```
Cluster 3 with frequency 6%
Lemma minus_0
Lemma plus_n_0
Lemma plus_0_n
Lemma mult_0_n
Lemma mult_n_0
Lemma app_l_nil
Lemma app_nil_l

Cluster 4 with frequency 5%
Lemma addSn
Lemma mulSn
Lemma plus_n_Sm
-U:**-  *display*      51% L16     (Fundamental)------------
minus_0 : forall n : nat, n - 0 = 0
-U:**-  *display2*     All L1      (Fundamental)------------
```

Figure 25: ***Clusters for the library*** `Initial`. *The Proof General window has been split into two windows positioned side by side: the left one keeps the current proof script, and the right one shows the clusters and their frequencies. If the user clicks on the name of a theorem showed in the right screen, such a window is split horizontally and a brief description of the selected theorem is shown.*

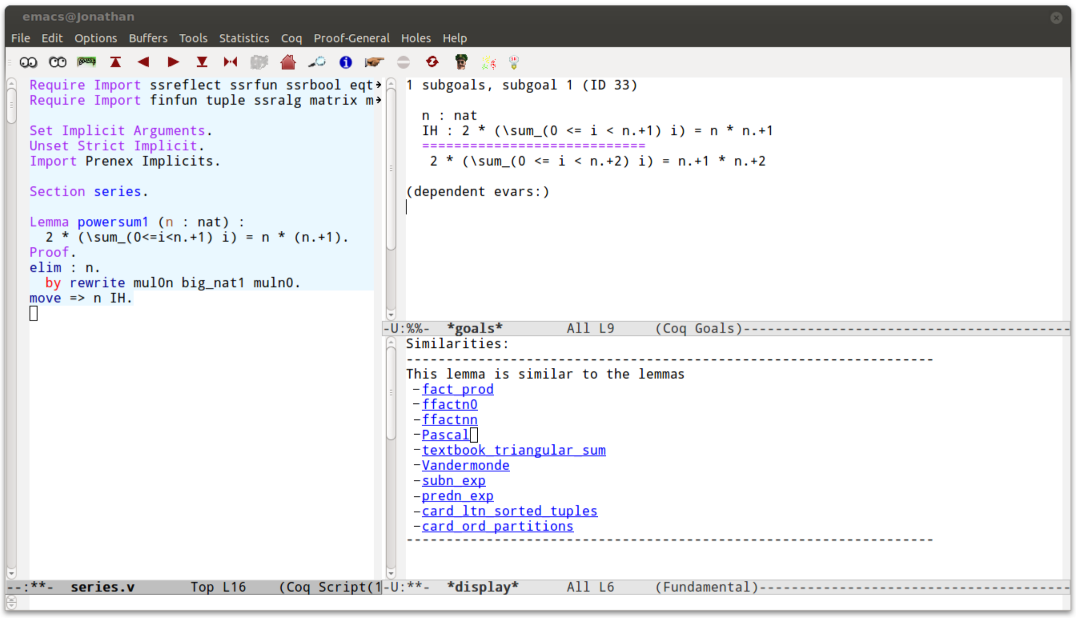
Among the suggestions provided by ML4PG, we find the Lemma `fact_prod`. Note that, as we have seen in Section 3, there is a high correlation between their feature vectors. Other lemmas ML4PG discovered are related to series of natural numbers (including properties about big sums and big products). Lemmas like `sum_first_n_odd`, where there is a restriction on the elements of the series, belong to a different cluster since the correlation with lemmas like `sum_first_n` and `fact_prod` is low.

We have shown flexibility, modularity and interactivity of ML4PG in interfacing with machine learning environments. These features come for free with the light version of *"backward interfacing"* that ML4PG implements: that is, it does not translate the outputs of the clustering algorithms back into the Coq language. Its only form of backward interfacing is conversion of clustered feature vectors back into lemma names – the output shown in Figures 25 and 27.

Generally, interfacing the ITPs with external tools (e.g. ATPs) is a challenging task, see [1,27,38,44]. A special concern is the translation of the output produced by the external tools into the ITP. This is due to the fact that unsound translation can introduce inconsistencies in the system; see e.g. [10]. In case of interfacing with machine learning, the external tool is even more alien to ITP's syntax than ATPs. Light backward interfacing implemented in ML4PG may well be the optimal solution to the problem.

5 Handling Proof Statistics in ML4PG

The previous sections highlighted two features of ML4PG – light backward interfacing and interactive handling of machine learning interfaces. To handle these features gracefully, ML4PG must offer the user a convenient environment for processing and analysing the *results* obtained by machine learning


emacs@Jonathan
File Edit Options Buffers Tools Statistics Coq Proof-General Holes Help

```
Require Import ssreflect ssrfun ssrbool eqt→
Require Import finfun tuple ssralg matrix m→

Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.

Section series.

Lemma powersum1 (n : nat) :
  2 * (\sum_(0<=i<n.+1) i) = n * (n.+1).
Proof.
elim : n.
  by rewrite mul0n big_nat1 muln0.
move => n IH.
```

```
1 subgoals, subgoal 1 (ID 33)

  n : nat
  IH : 2 * (\sum_(0 <= i < n.+1) i) = n * n.+1
  ============================
   2 * (\sum_(0 <= i < n.+2) i) = n.+1 * n.+2

(dependent evars:)
```

```
-U:%%-  *goals*      All L9     (Coq Goals)------------------------------------
```

```
Similarities:
---------------------------------------------------------------
This lemma is similar to the lemmas
 -fact_prod
 -ffactn0
 -ffactnn
 -Pascal
 -textbook_triangular_sum
 -Vandermonde
 -subn_exp
 -predn_exp
 -card_ltn_sorted_tuples
 -card_ord_partitions
---------------------------------------------------------------
```

```
--:**-  series.v      Top L16    (Coq Script(1-U:**-  *display*     All L6     (Fundamental)--------------------------------------
```

Figure 27: ***Similarities of Lemma*** *sum_first_n.* *On the left side, the development about the Lemma* *sum_first_n.* *On the top part of the right side, the current goal. On the bottom part of the right side, several suggestions provided by ML4PG. If the user clicks on the name of one of the suggested lemmas, a brief description about it is shown.*

algorithms.

Clustering techniques divide data into n groups of similar objects (called clusters), where the value of n is a "learning" parameter provided by the user together with other inputs to the clustering algorithm. Increasing the value of n means that the algorithm will try to separate objects into more classes, and, as a consequence, each cluster will contain fewer examples with higher correlation. The frequencies of clusters can serve for analysis of their reliability. Results of one run of a clustering algorithm may differ from another, even on the same data set. This is due to the fact that clustering algorithms randomly choose examples to start from, and form clusters relative to those examples. However, it may happen that certain clusters are found repeatedly – and frequently – in different runs; then, we can use these frequencies to determine the reliable clusters.

ML4PG's tools handling statistical results include a set of programs written in MATLAB and Weka, that post-process outputs of the clustering algorithms. For each clustering algorithm the user invokes, ML4PG generates one corresponding program handling the output statistics. These various programs always have three arguments: a file and two natural numbers representing the number of clusters and frequency threshold. We explain these settings in this section.

Various numbers of clusters can be useful for interactive proof data-mining: this may depend on the size of the data set, and on existing similarities between the proofs. We want ML4PG to accommodate such choices. In general, small values of n are useful when searching for general proof patterns which can later be refined by increasing the value of n. However, extreme values are to be avoided: small values of n can produce meaningless proof clusters for big proof libraries; whereas trivial clusters with just one proof may be found for big values of n. Very often in machine learning, the optimal number of clusters is determined experimentally, but we cannot afford this in ML4PG setting, as we assume the user focuses mainly on the Coq proofs.

In the machine learning literature, there exists a number of heuristics to determine this optimal num-

Granularity	Number of clusters
1	$\lfloor l/10 \rfloor$
2	$\lfloor l/9 \rfloor$
3	$\lfloor l/8 \rfloor$
4	$\lfloor l/7 \rfloor$
5	$\lfloor l/6 \rfloor$

Frequency parameter	Frequency Threshold
1	5%
2	15%
3	30%

Table 28: ***ML4PG formulas computing clustering parameters. Left:*** *the formula computing the number of clusters given the granularity value, where l is the number of lemmas in the library.* ***Right:*** *the formula computing frequency thresholds given a frequency parameter.*

ber of clusters, [58]. We used them as an inspiration to formulate our own algorithm for ML4PG, tailored to the interactive proofs. At any given stage of the interactive proof, it takes into consideration the size of the proof library and an auxiliary parameter we introduce here – called *granularity*. This parameter is used to calculate the optimal number of proof clusters, using the formulas of Table 28. As a result, the user does not provide the value of n, but just decides on *granularity* in ML4PG menu, by selecting a value between 1 and 5, where 1 stands for a low granularity (producing big and general clusters) and 5 stands for a high granularity (producing small and precise clusters).

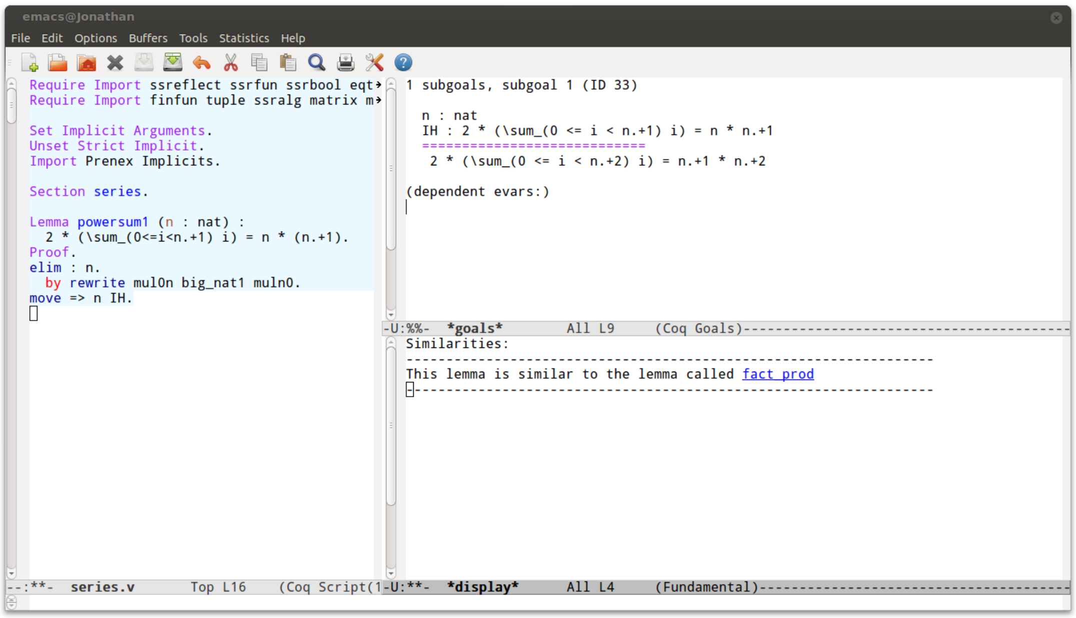
Example 29 Consider Example 24: there, the default granularity was 3, and the cluster contained all lemmas from Tables 3 and 4. Increasing the granularity, ML4PG discovers only Lemmas `app_l_nil` and `mult_n_O` (see $\star$ from Section 2), as well as similar proofs for `plus_n_O` and `minus_n_O`. All of them use induction, and prove similar base cases. Note that in Section 2 we conjectured this separation of examples of Tables 3 and 4 into two clusters as a desirable feature.

Example 30 In Example 26, ML4PG used the default granularity value of 3, to obtain ten suggestions related to the Lemma `sum_first_n`. If the ML4PG user increases such granularity value to 5, he obtains only one suggestion, see Figure 31: the Lemma `fact_prod`. Inspecting the proof of Lemma `fact_prod` can give an insight into how to finish the proof for `sum_first_n`. We notice that we can apply the Lemma `big_nat_recr` to our current goal and, subsequently use the inductive hypothesis. The rest of the proof is based on rewriting rules of natural numbers.

As implied by the above examples, the configuration of the granularity parameter can be approached in two different ways: *top-down* and *bottom-up*. The top-down approach suggests first using a small value for the granularity to obtain a general proof pattern, and then refine that pattern increasing the granularity value. On the contrary, in the bottom-up approach a high value for the granularity is used to see what the most similar lemmas are and then decrease the granularity value to see more general – and potentially less trivial – patterns.

Finally, the third parameter ML4PG uses to analyse clustering outputs is the *frequency threshold*. For this purpose, ML4PG actually uses double criteria: the *proximity* and *frequency* of the cluster. The clustering algorithm output contains not only clusters but also a proximity value – a measure of how close each object in one cluster is with respect to objects in other clusters. This measure ranges from $+1$, indicating points that are very distant from other clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are probably assigned to the wrong cluster. We have fixed 0.5 as an accuracy threshold, and all the clusters whose measure is under such value are ignored by ML4PG. This criterion is fixed, and the user interface does not give access to it. However, the second criterion, the frequency parameter, is customizable within the interface.

Our experience shows that analysis of frequencies may give two opposite effects.



```
Require Import ssreflect ssrfun ssrbool eqt
Require Import finfun tuple ssralg matrix m

Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.

Section series.

Lemma powersum1 (n : nat) :
  2 * (\sum_(0<=i<n.+1) i) = n * (n.+1).
Proof.
elim : n.
  by rewrite mul0n big_nat1 muln0.
move => n IH.
```

```
1 subgoals, subgoal 1 (ID 33)

  n : nat
  IH : 2 * (\sum_(0 <= i < n.+1) i) = n * n.+1
  ============================
   2 * (\sum_(0 <= i < n.+2) i) = n.+1 * n.+2

(dependent evars:)
```

```
-U:%%-  *goals*        All L9     (Coq Goals)----------------------------------------
```

```
Similarities:
-----------------------------------------------------------------
This lemma is similar to the lemma called fact prod
-----------------------------------------------------------------
```

```
--:**-  series.v       Top L16    (Coq Script(1-U:**-  *display*      All L4     (Fundamental)--------------------------------------
```

Figure 31: ***Suggestion about the Lemma*** sum_first_n ***using the granularity value*** 5.

* On the one hand, high frequencies suggest that the proofs found in clusters have a high correlation, and that is a desirable property.

** On the other hand, proofs with too high correlation may be too trivial for providing interesting proof hints. Therefore, it is sometimes useful to look for proof clusters with lower frequencies – as they may potentially contain those non-trivial analogies.

Example 32 Illustrating this, in our running example, the four proofs from Tables 3 and 4 were initially found only in 6% of runs (low frequency), see Figure 25; whereas there were other clusters with high frequencies that contained trivially similar lemmas (for instance, a cluster contains all the lemmas of the shape $x+0 = x$ where x is a term – the proofs of all these lemmas are the same).

To gather sufficient statistical data from proofs, ML4PG runs the chosen clustering algorithm 200 times at every call of clustering, and collects the frequencies of each cluster. After discarding those with low proximity, it calculates the frequency of the significant patterns. Once frequencies are calculated, ML4PG applies the following methodology. As item * suggests, one purpose of the frequencies is to serve as thresholds: if the number of times the cluster occurs falls below the pre-set threshold, the corresponding proofs will not be displayed to the user. On the other hand, as item ** suggests, the acceptable *frequency threshold* values may differ from proof to proof, and may depend on the purpose of proof pattern recognition. For this, ML4PG allows the user to vary the threshold values. At the moment, we implemented three choices: frequency parameters 1–3 as shown in Table 28. This particular range of thresholds comes from our experience with several Coq and SSReflect libraries. However, in line with our general modular approach to ML4PG design, we assume a wider range can be implemented, if desired. Our current choice is to keep the ML4PG interface simple and minimalistic.

Example 33 Table 33 shows the results for different choices of algorithms and parameters for Example 26, we highlight the result presented in Figure 27.

	$g=1$ $f=5\%$ $n=20$	$g=1$ $f=15\%$ $n=20$	$g=1$ $f=30\%$ $n=20$	$g=2$ $f=5\%$ $n=22$	$g=2$ $f=15\%$ $n=22$	$g=2$ $f=30\%$ $n=22$	$g=3$ $f=5\%$ $n=25$	$g=3$ $f=15\%$ $n=25$	$g=3$ $f=30\%$ $n=25$
Gaussian	99	0	0	76	76	0	30	30	30
K-means (MATLAB)	30	30	30	21	21	21	10	**10**	10
K-means (Weka)	26	26	0	22	22	22	20	20	20
Expectation Maximisation	80	0	0	72	0	0	64	64	0
FarthestFirst	0	0	0	81	0	0	73	0	0

	$g=4$ $f=5\%$ $n=29$	$g=4$ $f=15\%$ $n=29$	$g=4$ $f=30\%$ $n=29$	$g=5$ $f=5\%$ $n=34$	$g=5$ $f=15\%$ $n=34$	$g=5$ $f=30\%$ $n=34$
Gaussian	8	8	8	1	1	1
K-means (MATLAB)	7	7	7	1	**1**	1
K-means (Weka)	20	20	20	11	11	11
Expectation Maximisation	20	20	20	3	3	3
FarthestFirst	55	55	0	20	20	20

Table 34: ***A series of clustering experiments for Example 26****. The rows of the table indicate the algorithm used by ML4PG, and the columns show the values of the parameters: granularity (g), frequency thresholds (f) and the number of clusters n (computed from the granularity parameter). We record here only the clusters containing Lemma* `fact_prod` *– the most "useful" proof hint in the context. Note that such clusters are found irrespective of the chosen algorithm. The size of the data set is 205 lemmas. The results in bold provide the settings to obtain the result presented in Figures 27 and 31*

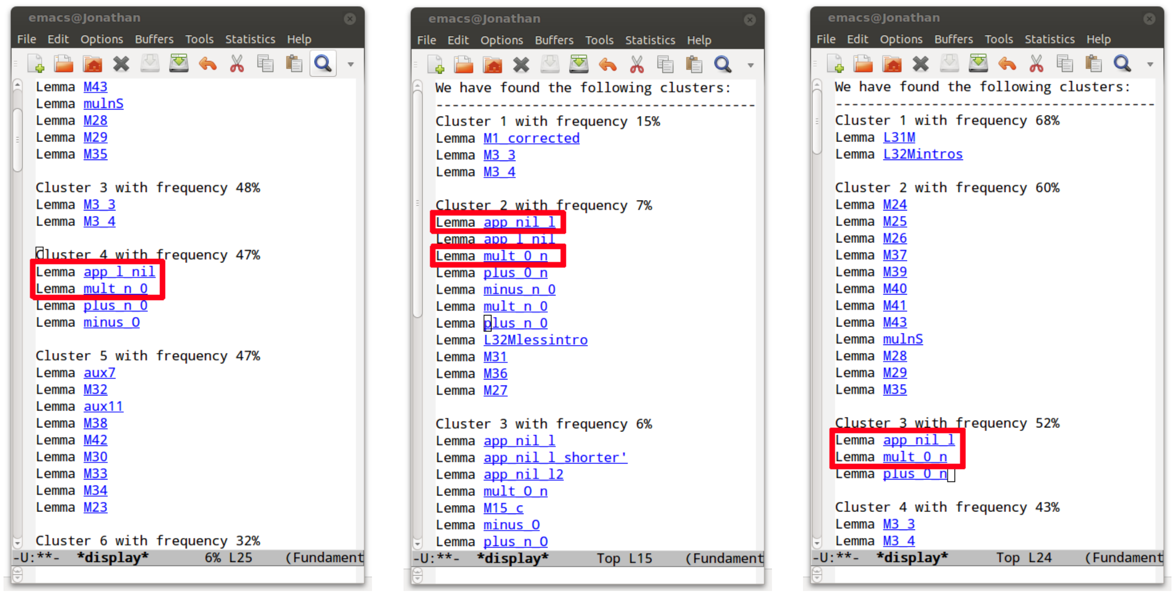
Our next example shows an interesting interplay between the effects of varying granularity, frequency, and proof level in the process of proof data-mining.

Example 35 In Example 24, ML4PG shows the clusters for our four running examples from the library `Initial`, when using the default frequency value of 1. As Example 32 showed, when increasing the frequencies parameter, such a cluster would fall below the threshold (compare with Figure 25, where the frequency of this cluster was 6%). At the same time, when increasing the granularity parameter to 5, our four proofs will be split into two smaller clusters, each having higher frequencies. Notably, inductive proofs (see $\star$ and Table 3 from Section 2 and Example 29) are separated from those by simplification (see $\star\star$ and Table 4). The cluster containing only lemmas from Table 3 has a frequency of 47% (see the left screenshot of Figure 36). The proofs from Table 4 also form a smaller cluster, but with frequency of 7%. Therefore, both clusters are shown if the frequency parameter is 1, but we also have an option of choosing a higher frequency (15% or 30%) to discard the second, less significant, cluster.

This is a typical situation, small values of the granularity parameter usually produce big clusters with small frequencies. When the granularity value is increased, the big clusters are split into smaller ones with high frequencies. Note the interactive nature of this proof pattern recognition process.

Example 37 A similar effect of increasing granularity parameter and increasing frequencies for the tactic-level proof features is shown in Figure 36. The figure also demonstrates that data-mining the same library using goal-level features and tactic-level features can bring different results. Interestingly, with increase of granularity, the goal-level clustering focuses on the examples related to lemmas in Table 3, whereas the tactic-level clustering focuses on examples related to lemmas of Table 4; as we conjectured in items $\star$ and $\star\star$ of Section 2.

We finish this section with a discussion of the role of this statistical analysis in our approach to the *light backward interfacing*. ML4PG handles the results obtained with MATLAB and Weka in a uniform way: for this purpose, we devised an XML format, see Figure 38. Using this approach, ML4PG can deal with the output generated by any system which follows this XML standard using just one program which



```
Lemma M43
Lemma mulnS
Lemma M28
Lemma M29
Lemma M35

Cluster 3 with frequency 48%
Lemma M3_3
Lemma M3_4

Cluster 4 with frequency 47%
Lemma app_l_nil
Lemma mult_n_0
Lemma plus_n_0
Lemma minus_0

Cluster 5 with frequency 47%
Lemma aux7
Lemma M32
Lemma aux11
Lemma M38
Lemma M42
Lemma M30
Lemma M33
Lemma M34
Lemma M23

Cluster 6 with frequency 32%
-U:**-  *display*      6% L25     (Fundament
```

```
We have found the following clusters:
-------------------------------------
Cluster 1 with frequency 15%
Lemma M1_corrected
Lemma M3_3
Lemma M3_4

Cluster 2 with frequency 7%
Lemma app_nil_l
Lemma app_l_nil
Lemma mult_0_n
Lemma plus_0_n
Lemma minus_n_0
Lemma mult_n_0
Lemma plus_n_0
Lemma L32Mlessintro
Lemma M31
Lemma M36
Lemma M27

Cluster 3 with frequency 6%
Lemma app_nil_l
Lemma app_nil_l_shorter'
Lemma app_nil_l2
Lemma mult_0_n
Lemma M15_c
Lemma minus_0
Lemma plus_n_0
-U:**-  *display*      Top L15     (Fundament
```

```
We have found the following clusters:
-------------------------------------
Cluster 1 with frequency 68%
Lemma L31M
Lemma L32Mintros

Cluster 2 with frequency 60%
Lemma M24
Lemma M25
Lemma M26
Lemma M37
Lemma M39
Lemma M40
Lemma M41
Lemma M43
Lemma mulnS
Lemma M28
Lemma M29
Lemma M35

Cluster 3 with frequency 52%
Lemma app_nil_l
Lemma mult_0_n
Lemma plus_0_n

Cluster 4 with frequency 43%
Lemma M3_3
Lemma M3_4
-U:**-  *display*      Top L24     (Fundament
```

Figure 36: ***Effects of increasing granularity of clustering on the levels of goals and tactics for the library*** `Initial`. ***Left:*** *compare to Figure 25, smaller clusters are formed when 5 is chosen as the granularity value at the goal level; frequencies increase. As we conjectured in Item ⋆ of Example 2, Lemmas* `app_l_nil` *and* `mult_n_0` *belong to the same cluster (see the lemmas highlighted with a circle in the screenshot).* ***Centre:*** *Tactic-level clusters formed with granularity parameter 2; the frequencies are relatively low. As we conjectured in Item ⋆⋆ of Example 2, Lemmas* `app_nil_l` *and* `mult_0_n` *belong to the same cluster when considering the tactic-level (see the lemmas highlighted with a circle in the screenshot).* ***Right:*** *Tactic-level clusters formed with granularity parameter 5, the clusters become smaller, and the frequencies increase. Even increasing the granularity, Lemmas* `app_nil_l` *and* `mult_0_n` *are still in the same cluster.*

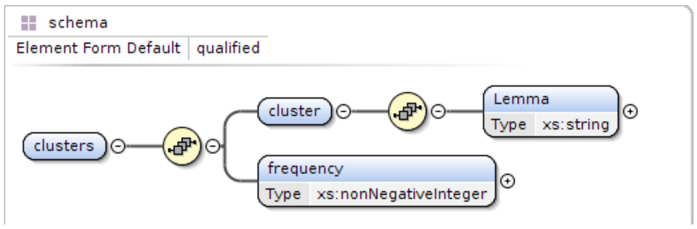


Figure 38: ***XML schema for handling results produced by different machine learning engines.*** *An XML file following this schema will consists of a main node called clusters, which has as child a sequence of pairs. The first element of the pair is a node called cluster whose child is the sequence of lemmas belonging to the cluster. The second element of the pair is the frequency of the cluster.*

transforms the XML files into a suitable format for the user. As a consequence, ML4PG can be easily extended with new engines and machine learning algorithms.

The XML files returned by the machine learning engines are processed in two different ways depending on the mode of using ML4PG: that is, general clustering (as illustrated in Example 24) or goal-dependent clustering (as shown in Example 26). In both cases, the XML file is converted to a list of pairs where the first element of the pair contains the lemma names and the second element the frequency of each cluster. In the general clustering case, such a list is processed to be shown as in Figure 25. For the goal-dependent clustering, ML4PG searches for those pairs of the list where the current proof is included. If the current proof belongs to several clusters, then ML4PG takes the one with the highest frequency and displays it as shown in Figure 27.

6 Integrating machine learning with theorem proving: related work

In this section, we present an overview of the integration of machine learning techniques into automated and interactive proofs. There are two machine learning styles that can be useful in this context: *symbolic* and *statistical*.

Symbolic machine learning methods can formulate auxiliary lemmas or proof strategies from background knowledge. As we have explained in the introduction, Proof General is a general-purpose interface for a range of higher-order theorem provers, and this is probably the reason why this interface has been used in different works to integrate machine learning techniques. This is the case of IsaPlanner [18], a generic framework for proof planning that integrates techniques like rippling [6] in the interactive theorem prover Isabelle. Also using Isabelle as a prover and Proof General as interface, PGtips [45] is a tactic recommender system based on data-mining techniques [19]. Another advisor implemented in Proof General, but in this case for the Mizar system, was incorporated into MizarMode [54] to suggest similar results which could be useful in the current proof.

Inductive provers like ACL2 [35] and Hip [50] also include symbolic machine learning techniques. In particular, ACL2 uses these methods for proving termination of programs written in Lisp [36], and Hip implements a theory discovery mechanism [13] for automatically derive and prove properties about functional programs implemented in Haskell.

The main drawback of the symbolic methods is their scope. These techniques are often tailored for certain fragments of first order language or a certain library, or a certain proof shape; therefore, they do not scale properly to deal with big libraries. On the contrary, statistical machine learning methods scale to big libraries without problems, but they have very weak power for generalisation.

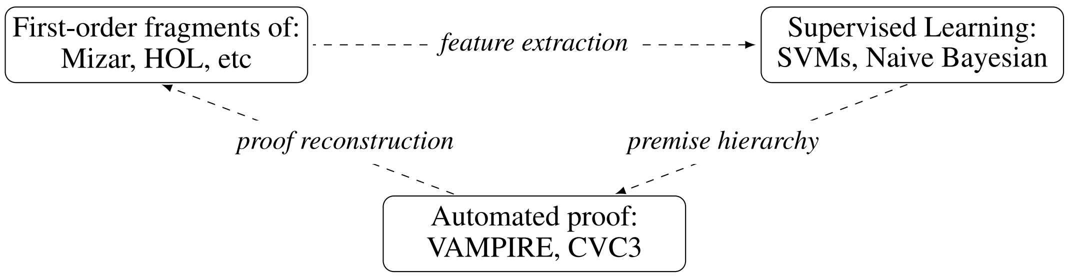

First-order fragments of: Mizar, HOL, etc
feature extraction
Supervised Learning: SVMs, Naive Bayesian
proof reconstruction
premise hierarchy
Automated proof: VAMPIRE, CVC3

Figure 39: ***Machine learning techniques for premise selection.***

Statistical machine learning methods can discover data regularities based on numeric proof features. ML4PG belongs to this category of methods. Among other successful statistical tools is the method of statistical proof-premise selection [39–41, 53]. Applied in several ATPs, it provides statistical ratings of existing lemmas; and this makes automated rewriting more efficient.

This technique has also been used to integrate ITPs, ATPs and machine learning. The workflow of tools like Sledgehammer to prove a theorem consists of the following steps: (1) translation of the statement of the theorem (from Isabelle, HOL or Mizar format) to a first order format suitable for ATPs; (2) selection of the lemmas (or premises) that could be relevant to prove the theorem; (3) invocation of several ATPs to prove the result; and (4) if an ATP is successful in the proof, reconstruction of the proof in the ITP from the output generated by the ATP. An important issue in this procedure is the premise selection mechanism, especially when dealing with big libraries, since proofs of some results can be infeasible for the ATPs if they receive too many premises.

Statistical machine learning methods are used to tackle this problem in [40, 41]. In particular, a classifier is constructed for every lemma of the library. Given two lemmas A and B, if B was used in the proof of A, a feature vector $|A|$ is extracted from the statement of A, and is sent as a positive example to the classifier $< B >$ constructed for B; otherwise, $|A|$ is used as a negative example to train $< B >$. Note that, $|A|$ captures statistics of A's syntactic form relative to every symbol in the library; and the resulting feature vector is a sparse (including up to 10^6 features). After such training, the classifier $< B >$ can predict whether a new lemma C requires the lemma B in its proof, by testing $< B >$ with the input vector $|C|$. On the basis of such predictions for all lemmas in the library, this tool constructs a hierarchy of the premises that are most likely to be used in the proof of C. Figure 39 illustrates this approach.

Table 40 summarises the main differences between premise-selection tools and ML4PG. It is important to notice that the two methods have different approaches to handling large-size data. Premise-selection tools rely on advanced *"sparse"* machine-learning algorithms to process the growing feature vectors. ML4PG achieves scaling at the stage of feature extraction, by using the proof trace method to produce compact feature vectors. As a result, ML4PG also works well with libraries of small size (and hence can interact with the user at any stage of the proof), whereas sparse methods need proof libraries of big size to perform well.

7 Conclusions and Further work

In this paper, we have presented a Proof General extension, called ML4PG, to interface ITPs and machine learning engines. Our main goal was to prove that it is possible to interface higher-order *interactive* theorem proving with statistical machine learning; and the resulting tool can provide fast and non-trivial

	premise-selection tools	ML4PG
Aim	increase number of goals automatically proved by ATPs	assist the user providing proof families as proof hints
Features	extracted from first-order formulas	extracted from higher-order formulas and proofs
Feature vectors representation	sparse (up to 10^6 features)	dense (≈ 30 features)
Machine learning methods	supervised learning (SVMs, Naive Bayesian, ...)	unsupervised learning (K-means, Gaussian, ...)
Scaling for big libraries	on machine-learning level: *sparse* algorithms	on the interface level: *proof-trace* method

Table 40: ***Differences between premise-selection tools and ML4PG.***

proof hints during the proof development. The technical highlights of ML4PG are:

- the *proof trace method* is a flexible, extendable technique that gathers statistics from proofs on the basis of the relative transformations of simple parameters within several proof steps; and,
- the *light backward interfacing* implemented in ML4PG automates the Proof General interaction with machine learning engines. It helps to analyse and interpret the output of machine learning algorithms; however, it avoids full translation of the statistical outputs into the prover's language.

The ML4PG approach has several benefits. First of all, it does not assume any knowledge of machine learning interfaces from the user; and automates initial statistical experiments (determining the number of clusters, calculating frequencies) that otherwise would have been performed by hand. The choices for various measures of cluster granularity and frequency can be easily extended in the future. Moreover, it is a modular tool which allows the user to make choices regarding approach to levels of proofs and particular statistical algorithms. By design, it allows further extensions to different machine learning environments, modes of supervised/unsupervised learning, and various learning algorithms within those modes. In addition, it is tolerant to mixing and matching different proof libraries, different notations and proof styles used across several developments.

Comparing across different proof levels and different styles of proofs, our experiments show that data-mining the goal-level features shows more interesting clusters compared to the other two feature extraction methods. We plan to improve the other two feature extraction methods in the future. Proofs in SSReflect yield more consistent classification results compared to the plain Coq style. This is due to a stricter proof discipline in SSReflect, which allows ML4PG to detect more significant proof patterns.

The feature extraction method implemented in ML4PG can be improved in two different ways. First of all, the proof trace method considers just the first five proof steps, losing some information. As the machine learning algorithms integrated in ML4PG require a fixed number of features whereas Coq proofs may have varied length, we can study big proofs considering either small patches of proofs (by partially reusing the proof trace method), or proofs as a whole (in this case, a sparse representation will be necessary). The numerical assignment provided by the function $[[\,.\,]]_{hyp\_or\_lemma}$ for the lemmas of the libraries gives a big value spread (especially if ML4PG works with big proof libraries). We can tackle this problem assigning closer numbers to similar lemmas. We are currently working in the implementation of these new features in ML4PG.

ML4PG can be combined with other tools to make the proof development easier. Search mechanisms implemented in Coq, such as the `Search` command of SSReflect [22] and Whelp [2], can find patterns in lemma statements, but ML4PG can discover proof patterns that cannot be found using existing Coq's search mechanisms, see [28].

Moving towards symbolic machine-learning, we envisage that new lemmas or strategies could be conceptualised from the proof families obtained with ML4PG using such techniques as Rippling [6] or Theory Exploration [32].

In addition, we plan to integrate more machine learning methods to help in the proof process. To this aim, we need a tool which tracks not only the successful proofs, but also failed and discarded derivations steps. In this way, we could use *supervised* machine learning algorithms to indicate a user whether he is following a sensible strategy based on the previous experience. Supervised machine learning could also be used to discover various proof styles.

We are interested in increasing the number of proof assistants included in our framework. This will allow us to study proof similarities across different theorem provers. Since the interaction with theorem provers such as Isabelle or Lego is already available in Proof General, we just need the implementation of the feature extraction mechanism for them; their interaction with the machine learning engines would be the same as developed for Coq and SSReflect.

Finally, current implementation of ML4PG is centralised; this means that the user can obtain proof clusters of the libraries available on his computer. However, we think that a client-server architecture, where the proof information is shared among several users could also be useful, especially for team-based program development.

For this purpose, feature extraction in ML4PG is already designed to be lemma name and notation independent.

8 Acknowledgements

We are grateful to the anonymous referees for their comments and suggestions; and the following individuals and research groups for inspiring discussions on the topics of machine learning in automated and interactive theorem proving: J Strother Moore, members of the AI4FM project Alan Bundy, Cliff Jones, Leo Freitas, Ewen Maclean; and participants of Dagstuhl seminar *AI meets Formal Software Development* [12].

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry & B. Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *1st International Conference on Certified Programs and Proofs (CPP'11)*, *Lecture Notes in Computer Science* 7086, pp. 135–150.

[2] A. Asperti, F. Guidi, C. Sacerdoti Coen, E. Tassi & S. Zacchiroli (2006): *A Content Based Mathematical Search Engine: Whelp*. In: *Post-Proceedings of the TYPES'04 International Conference*, *Lecture Notes in Computer Science* 3839, pp. 17–32.

[3] A. Asperti, W. Ricciotti, C. Sacerdoti Coen & E. Tassi (2011): *The Matita interactive Theorem prover*. In: *23rd International Conference on Automated Deduction (CADE'11)*, *Lecture Notes in Computer Science* 6803, pp. 64–69.

[4] D. Aspinall (2000): *Proof General: A Generic Tool for Proof Development*. In: *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, *Lecture Notes in Computer Science* 1785, pp. 38–43.

[5] C. Barrett & C. Tinelli (2007): *CVC3*. In: *19th International Conference on Computer Aided Verification (CAV'07)*, *Lecture Notes in Computer Science* 4590, pp. 298–302.

[6] D. Basin, A. Bundy, D. Hutter & A. Ireland (2005): *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press.

[7] A. Bauer, E. M. Clarke & X. Zhao (1998): *Analytica - an experiment in combining theorem proving and symbolic computation*. Journal of Automated Reasoning 21(3), pp. 295–325.

[8] C. Bishop (2006): *Pattern Recognition and Machine Learning*. Springer.

[9] J. C. Blanchette, S. Böhme & L. C. Paulson (2011): *Extending Sledgehammer with SMT Solvers*. In: 23rd International Conference on Automated Deduction (CADE-23), Lecture Notes in Computer Science 6803, pp. 116–130.

[10] J. C. Blanchette, S. Böhme, A. Popescu & N. Smallbone (2013): *Encoding monomorphic and polymorphic types*. In: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), Lecture Notes in Computer Science 7795, pp. 493–507.

[11] A. Bove, P. Dybjer & U. Norell (2009): *A Brief Overview of Agda — A Functional Language with Dependent Types*. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09), Lecture Notes in Computer Science 5674, pp. 73–78.

[12] A. Bundy, D. Hutter, C. B. Jones & J S. Moore (2012): *AI meets Formal Software Development (Dagstuhl Seminar 12271)*. Dagstuhl Reports 2(7), pp. 1–29.

[13] K. Claessen, M. Johansson, L. Dixon & A. Bundy (2013): *Automating Inductive Proofs using Theory Exploration*. In: 24th International Conference on Automated Deduction (CADE–24), Lecture Notes in Computer Sciene.

[14] COQ development team (2012): *The* COQ *Proof Assistant Reference Manual, version 8.4*. Technical Report.

[15] Thierry Coquand & Gérard P. Huet (1988): *The Calculus of Constructions*. Inf. Comput. 76(2/3), pp. 95–120.

[16] J. Denzinger, M. Fuchs, C. Goller & S. Schulz (1999): *Learning from Previous Proof Experience: A Survey*. Technical Report, Technische Universitat Munchen.

[17] J. Denzinger & S. Schulz (2000): *Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts*. Information and Computation 162(1-2), pp. 59–79. Available at `http://dx.doi.org/10.1006/inco.1999.2857`.

[18] L. Dixon & J. D. Fleuriot (2003): *IsaPlanner: A Prototype Proof Planner in Isabelle*. In: 19th International Conference on Automated Deduction (CADE'2003), Lecture Notes in Computer Science 2741, pp. 279–283.

[19] H. Duncan (2002): *The use of Data-Mining for the Automatic Formation of Tactics*. Ph.D. thesis, University of Edinburgh.

[20] B. Dutertre & L. de Moura (2006): *The Yices SMT solver*. Available at `http://yices.csl.sri.com/tool-paper.pdf`.

[21] G. Gonthier (2008): *Formal proof - The Four-Color Theorem*. Notices of the American Mathematical Society 55(11), pp. 1382–1393.

[22] G. Gonthier & A. Mahboubi (2010): *An introduction to small scale reflection*. Journal of Formalized Reasoning 3(2), pp. 95–152.

[23] A. Grabowski, A. Kornilowicz & A. Naumowicz (2010): *Mizar in a nutshell*. Journal of Formalized Reasoning 3(2), pp. 153–245.

[24] G. Grov, E. Komendantskaya & A. Bundy (2012): *A Statistical Relational Learning Challenge - extracting proof strategies from exemplar proofs*. In: ICML'12 worshop on Statistical Relational Learning.

[25] T. Hales (2005): *The Flyspeck Project fact sheet*. Project description available at `http://code.google.com/p/flyspeck/`.

[26] M. Hall et al. (2009): *The WEKA Data Mining Software: An Update*. SIGKDD Explorations 11(1), pp. 10–18.

[27] J. Harrison & L. Théry (1998): *A skeptic approach to combining HOL and Maple*. Journal of Automated Reasoning 21, pp. 279–294.

[28] J. Heras & E. Komendantskaya (2012): *ML4PG: machine learning interface for Proof General. Program files and user manual.* `http://www.computing.dundee.ac.uk/staff/katya/ML4PG/`.

[29] J. Heras & E. Komendantskaya (2013): *ML4PG in Computer Algebra Verification.* In: *Conferences on Intelligent Computer Mathematics (CICM'13)*, Lecture Notes in Computer Science.

[30] W. A. Hunt & E. Reeber (2006): *A SAT-based procedure for verifying finite state machines in ACL2.* In: *6th International workshop on the ACL2 theorem prover and its applications*, pp. 127–135.

[31] M. Jamnik, M. Kerber & C. Benzmller (2002): *Automatic learning of proof methods in proof planning.* Technical Report.

[32] M. Johansson, L. Dixon & A. Bundy (2011): *Conjecture Synthesis for Inductive Theories.* *Journal of Automated Reasoning* 47(3), pp. 251–289.

[33] I. Joliffe (1986): *Principal Components Analysis.* Springer-Verlag.

[34] M. Kaufmann & J S. Moore (2012): *Accumulated persistence in ACL2.* `http://www.cs.utexas.edu/users/moore/acl2/current/ACCUMULATED-PERSISTENCE.html`.

[35] M. Kaufmann, P. Manolios & J S. Moore, editors (2000): *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers.

[36] M. Kaufmann, P. Manolios, J S. Moore & D. Vroon (2006): *Integrating CCG analysis into ACL2.* In: *Eighth International Workshop on Termination, part of FLOC'06.*

[37] E. Komendantskaya & K. Lichota (2012): *Neural Networks for Proof-Pattern Recognition.* In: *International Conference on Artificial Neural Networks (ICANN'12)*, Lecture Notes in Computer Science, pp. 427–435.

[38] V. Komendantsky, A. Konovalov & S. Linton (2012): *Interfacing Coq + SSReflect with GAP.* In: *9th International Workshop On User Interfaces for Theorem Provers (UITP'10)*, *Electronic Notes in Theoretical Computer Science* 285, pp. 17–28.

[39] D. Kuehlwein & J. Urban (2012): *Learning from Multiple Proofs: first experiments.* In: *3rd Workshop on Practical Aspects of Automated Reasoning (PAAR'12)*, pp. 82–94.

[40] D. Kühlwein, J. C. Blanchette, C. Kaliszyk & J. Urban (2013): *MaSh: Machine Learning for Sledgehammer.* In: *4th Conference on Interactive Theorem Proving (ITP'13)*, Lecture Notes in Computer Sciene.

[41] D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban & T. Heskes (2012): *Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics.* In: *6th International Joint Conference on Automated Reasoning (IJCAR'12)*, *Lecture Notes in Computer Science* 7364, pp. 378–392.

[42] MATLAB (2012): *version 7.14.0 (R2012a).* The MathWorks Inc., Natick, Massachusetts.

[43] J. Meng & L. C. Paulson (2009): *Lightweight relevance filtering for machine-generated resolution problems.* *Journal of Applied Logic* 7(1), pp. 41–57.

[44] J. Meng & L. C. Paulson (2009): *Translating higher-order clauses to first-order clauses.* *Journal of Automated Reasoning* 40(1), pp. 35–60.

[45] A. Mercer, A. Bundy, H. Duncan & D. Aspinall (2006): *PG Tips: A Recommender System for an Interactive Theorem Prover.* In: *Mathematical User-Interfaces Workshop (MathUI'2006).*

[46] L. M. de Moura & N. Bjorner (2008): *Z3: An efficient SMT solver.* In: *14th Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, *Lecture Notes in Computer Science* 4963, pp. 337–340.

[47] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* *Lecture Notes in Computer Science* 2283, Springer.

[48] L. C. Paulson & J. C. Blanchette (2010): *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers.* In: *8th International Workshop on the Implementation of Logics (IWIL'10).*

[49] A. Riazano & A. Voronkov (2002): *The design and implementation of Vampire.* *Artificial Intelligence Communications* 15(2–3), pp. 91–110.

[50] D. Rosén (2012): *Proving Equational Haskell Properties using Automated Theorem Provers*. Master's thesis, University of Gothenburg, Sweden.

[51] S. Schulz (2004): *System description: E 0.81*. In: *2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, *Lecture Notes in Computer Science* 3097, pp. 223–228.

[52] Mathematical components team (2012): *Formalization of the Odd Order theorem*. Technical Report. `http://www.msr-inria.inria.fr/Projects/math-components`.

[53] E. Tsivtsivadze, J. Urban, H. Geuvers & T. Heskes (2011): *Semantic Graph Kernels for Automated Reasoning*. In: *SIAM Conference on Data Mining (SDM'11)*, SIAM / Omnipress, pp. 795–803.

[54] J. Urban (2006): *MizarMode – an integrated proof assistance tool for the Mizar way of formalizing mathematics*. *Journal of Applied Logic* 4(4), pp. 414–427.

[55] J. Urban (2006): *MPTP 0.2: Design, Implementation, and Initial Experiments*. *Journal of Automated Reasoning* 37(1-2), pp. 21–43.

[56] J. Urban, G. Sutcliffe, P. Pudlák & J. Vyskocil (2008): *MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance*. In: *4th International Joint Conference on Automated Reasoning (IJCAR'08)*, *Lecture Notes in Computer Science* 5195, pp. 441–456.

[57] C. Weidenbach (2001): *Combining superposition, sorts and splitting*, pp. 1965–2013. Handbook of Automated Reasoning, Elsevier.

[58] R. Xu & D. Wunsch (2005): *Survey of Clustering Algorithms*. *IEEE Transactions on Neural Networks* 16(3), pp. 645–678.