# Proof-pattern Search Methods in Coq*

Jónathan Heras and Ekaterina Komendantskaya

School of Computing, University of Dundee, UK
{jonathanheras,katya}@computing.dundee.ac.uk

**Abstract.** ML4PG is a machine-learning extension designed to aid proof development in Coq. It collects statistics from the user interaction with the prover, compares the data with other existing libraries, and shows the user existing proof similarities. In this paper, we provide a thorough evaluation of ML4PG in three steps. First, we develop two challenging benchmarks: finding proof-patterns in mathematical proofs (across libraries for Linear Algebra, Combinatorics and Persistent Homology), and software verification (verification of the Java Virtual Machine). Next, we compare ML4PG's functionality with existing symbolic methods of pattern search in Coq. Finally, we introduce two new extensions to ML4PG: one allows ML4PG to handle big proofs by analysing small *proof-patches*, and the other uses sparse methods to analyse entire proofs.
**Keywords:** Interactive Theorem Proving, Coq, SSReflect, Machine Learning, Clustering.

## 1 Introduction

Development of interactive theorem provers (ITPs) has led to creation of big libraries and varied infrastructures for formal mathematical proofs. These frameworks usually involve thousands of definitions and theorems (for instance, there are approximately 4200 definitions and 15000 theorems in the formalisation of the Odd Order theorem [9]). It becomes difficult to trace them to find patterns which could be reused in the proof of a new theorem. Similarly, ITPs have been successfully applied in industry to verify the correctness of hardware and software systems. In this context, proofs usually have more regularity and involve many routine or similar cases. However, the proofs are often developed by a team, where users have their own definitions and notation. In both contexts, it would be extremely helpful to use a tool that could detect proof patterns across different users, notations and libraries.

As a solution, we propose ML4PG [12] – a machine-learning extension to the Proof General [2] interface for Coq; its basic architecture was introduced in [16]. Its main purpose is to assist interaction between the programmer and the ITP, helping the user to find similar proofs in the library. In this sense, ML4PG extends the functionality of other existing tools for intelligent proof search in Coq: e.g. `Search` [10], `SearchPattern`, `SearchAbout`, `SearchRewrite` [8] and `Whelp` [1].

* The work was supported by EPSRC grant EP/J014222/1.

Currently, ML4PG works with plain Coq and its dialect SSReflect. It has three main features:

**F1.** it works on the background of Proof General, and extracts some simple, low-level features from interactive proofs in Coq/SSReflect;
**F2.** it automatically sends the gathered statistics to a chosen machine-learning interface and triggers execution of a clustering algorithm of the user's choice;
**F3.** it does some gentle post-processing of the results given by the machine-learning tool, and displays families of related proofs to the user. On user's request, the clustering may be performed in relation to the current unfinished proof, or in a goal-independent way.

The purpose of this paper is twofold: to provide a first proper evaluation for ML4PG and to introduce two technical improvements to it. To achieve the former, we introduce challenging benchmarking libraries for ML4PG, and carefully compare the outcomes with existing symbolic methods of proof-pattern search in Coq. In Section 2, we introduce Benchmark-1 involving several libraries for Linear Algebra, Combinatorics and Persistent Homology; and Benchmark-2 – involving libraries from verification of the Java Virtual Machine [19]. In Section 3, we evaluate the ML4PG's functionality on these two benchmarks. For ML4PG's performance for Computer Algebra formalisations, see [13]. In Section 4, we analyse different possible approaches to proof-pattern recognition in an ITP framework. The first approach (as implemented in standard tools like `Search` and `Whelp`) finds local and low-level patterns in definition and lemma statements, and is based on unification/matching techniques; see Section 4.1. We discuss the new features that machine-learning brings into the intelligent search, as compared to the standard symbolic methods.

Based on this evaluation, we introduce two extensions for ML4PG, allowing us to approach the notion of a proof pattern on two different levels of an interactive proof. First extension is a *proof-patch* method. It finds proof patterns as given by small fragments of big proofs. As opposed to the symbolic approach, these patterns encompass several subgoals, and can capture proof strategies on a higher relational level; see Section 4.2. The second extension is an even more general method to consider proof patterns in ML4PG: we integrate sparse machine-learning methods to analyse proof features coming from proofs of varied sizes. Here, a proof pattern is given by an entire proof; see Section 4.3.

Section 5 discusses related work on machine-learning in theorem proving, and concludes the paper.

## 2 ML4PG Benchmarks

Determining a suitable set of benchmarking examples for ML4PG is no easy task. Unlike e.g. related work on using machine-learning for automated premise selection [17,18], we do not have a quantitative target when it comes to improving *interactive* proof building experience. No longer speed up in automated proof search or the number of automatically proven theorems are the main criteria of

success. We generally follow the intuition that ML4PG, being an interactive hint generator, must provide *interesting and non-trivial* hints on user's demand, and should be flexible enough to do so at *any stage* of the proof, and relative to any chosen proof library. In this section, we formulate two *meta* **proof strategies** to be the benchmarks: the strategies go beyond the simple combination of tactics, lemma name suggestions, or datatypes.

All libraries and proofs we mention here can be downloaded from [12].

### 2.1 Benchmark 1. Mathematical Libraries

The first case study concerns discovery of proof patterns in mathematical proofs across formalisations of apparently disjoint mathematical theories: Linear Algebra, Combinatorics and Persistent Homology. In this scenario, we use statistically discovered proof patterns to advance the proof of a given "problematic" lemma. In this case, a few initial steps in its proof are clustered against several mathematical libraries.

In this section, we deliberately take lemmas belonging to very different Coq libraries. Lemma 1 states a result about *nilpotent* matrices [6] (a square matrix $M$ is *nilpotent* if there exists an $n$ such that $M^n = 0$). Lemma 2 is a basic fact about summations. Finally, Lemma 3 is a generalisation of the *fundamental lemma of Persistent Homology* [11].

**Lemma 1** Let $M$ be a square matrix and $n$ be a natural number such that $M^n = 0$, then $(1 - M) \times \sum_{i=0}^{n-1} M^i = 1$.

**Lemma 2** If $g : \mathbb{N} \to \mathbb{Z}$, then

$$\sum_{0 \leq i \leq n} (g(i+1) - g(i)) = g(n+1) - g(0).$$

A Coq proof for Lemma 2 is given in Table 1.

**Lemma 3** Let $\beta_n^{k,l} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$, then

$$\beta_n^{k,l} - \beta_n^{k,m} = \sum_{0 \leq i \leq k} \sum_{l < j \leq m} (\beta_n^{i,j-1} - \beta_n^{i,j}) - (\beta_n^{i-1,j-1} - \beta_n^{i-1,j}).$$

When proving Lemma 1, it is difficult, even for the expert user, to get the intuition that he can reuse the proofs of Lemmas 2 and 3. There are several reasons for this. First of all, the formal proofs of these lemmas are in different libraries: the proof of Lemma 1 is in the library about matrices, the proof of Lemma 2 is in the library about basic results concerning summations, and the proof of Lemma 3 is in the library about Persistent Homology. Then, it is difficult to establish a conceptual connection among them. Moreover, although the three lemmas involve summations, the types of the terms of those summations

| Goals and Subgoals | Applied Tactics |
|---|---|
| $\forall\ n, \sum_{i=0}^{n}(g(i+1)-g(i)) = g(n+1)-g(0)$ | `elim : n => [|n _].` |
| $\sum_{i=0}^{0}(g(i+1)-g(i)) = g(1)-g(0)$ | `by rewrite big_nat1.` |
| $\sum_{i=0}^{n+1}(g(i+1)-g(i)) = g(n+2)-g(0)$ | `rewrite sumrB big_nat_recr big_nat_recl addrC addrC -subr_sub -!addrA addrA.` |
| $g(n+2)+\sum_{i=0}^{n} g(i+1)-$ $\sum_{i=0}^{n} g(i+1)-g(0) = g(n+2)-g(0)$ | `move : eq_refl; rewrite -subr_eq0; move/eqP => ->.` |
| $g(n+2)+0-g(0) = g(n+2)-g(0)$ | `by rewrite sub0r.` |
| $\square$ | `Qed.` |

**Table 1.** *Proof for Lemma 2 in SSReflect.*

are different. Therefore, search based on types or keywords would not help. Even search of all the lemmas involving summations does not provide a clear suggestion, since there are more than 250 lemmas – a considerable number for handling them manually.

However, if Lemmas 2 and 3 are suggested when proving Lemma 1, the expert would be able to spot the following common proof pattern.

**Proof Strategy 1**

1. *Apply case on $n$.*
   (a) *Prove the base case (a simple task).*
   (b) *Prove the case $0 < n$:*
      i. *expand the summation,*
      ii. *cancel the terms pairwise,*
      iii. *the only terms remaining after the cancellation are the first and the last one.*

We also include the following lemma. At first sight, the proof of this lemma does not seem to fit Proof Strategy 1, since the statement of the lemma does not involve summations. However, inspecting its proof, we can see that it uses $\sum_{i=0}^{n-1} M^i$ as witness for $N$ and then follows Proof Strategy 1.

**Lemma 4** Let $M$ be a nilpotent matrix, then there exists a matrix $N$ such that $N \times (1 - M) = 1$.

Discovering that, across 758 lemmas and 5 libraries, the four lemmas given above follow the Strategy 1 will be our first benchmarking task for ML4PG.

### 2.2 Benchmark 2. Verification of JVM

We now turn to industrial applications of Coq and ML4PG. For this purpose, we translate the ACL2 proofs of correctness of the Java Virtual Machine (JVM) [14] into Coq. JVM [19] is a stack-based abstract machine which can execute Java bytecode. We have modelled an interpreter for JVM programs in Coq. From now on, we refer to our machine as "CJVM" (for Coq JVM).

Industrial scenario of interactive theorem proving may differ significantly from the mathematical scenario above, as it may involve distribution of workload across a team, and a bigger proportion of routine or repetitive cases . Here, the inefficiency often arises when programmers use different notation to accomplish very similar tasks, and thus a lot of work gets duplicated, see also [7]. We tested ML4PG in exactly such scenario: we assumed that a programming team is collectively developing proofs of *a) soundness of specification, and b) correctness of implementation* of Java bytecode for a dozen of programs computing multiplication, powers, exponentiation, and other functions about natural numbers.

Given a specific Java method, we can translate it to Java bytecode using a tool such as `javac` of Sun Microsystems. Such a bytecode can be executed in CJVM provided a schedule, and the result will be the state of the JVM at the end of the schedule. Moreover, we can prove theorems about the CJVM model behaviour when interpreting that bytecode.

*Example 1.* The bytecode associated with the factorial program can be seen in Figure 1.

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

```
0  : iconst 1
1  : istore 1
2  : iload 0
3  : ifeq 13
4  : iload 1
5  : iload 0
6  : imul
7  : istore 1
8  : iload 0
9  : iconst 1
10 : isub
11 : istore 0
12 : goto 2
13 : iload 1
14 : ireturn
```

```
Fixpoint helper_fact (n a) :=
match n with
| 0 => a
| S p => helper_fact p (n * a)
end.

Definition fn_fact (n : nat) :=
     helper_fact n 1.
```

**Fig. 1.** ***Factorial function. Left:*** *Java program for computing the factorial of natural numbers.* ***Centre:*** *Java bytecode associated with the Java program.* ***Right:*** *tail recursive version of the factorial function in Coq.*

The state of the CJVM consists of 4 fields: a *program counter* (a natural number), a set of registers called *locals* (implemented as a list of natural numbers), an operand *stack* (a list of natural numbers), and the bytecode *program* of the method being evaluated.

Java bytecode, like the one presented in Figure 1, can be executed within CJVM. However, more interesting than mere executing Java bytecode, we can prove the correctness of the implementation of the Java bytecode programs using Coq. For instance, in the case of the factorial program, we can prove the following theorem, which states the correctness of the `factorial` bytecode.

**Lemma 5** Given a natural number $n$ and the factorial program with $n$ as an input, CJVM produces a state which contains $n!$ on top of the stack running the bytecode associated with the program.

The proof of theorems like the one above always follows the same methodology adapted from ACL2 proofs about Java Virtual Machines [14] and which consists of the following three steps.

(1) Write the specification of the function, write the algorithm, and prove that the algorithm satisfies the specification.
(2) Write the JVM program within Coq, define the function that schedules the program (this function will make CJVM run the program to completion as a function of the input to the program), and prove that the resulting code implements this algorithm.
(3) Prove total correctness of the Java bytecode.

Using this methodology, we have proven the correctness of several programs related to arithmetic (multiplication of natural numbers, exponentiation of natural numbers, and so on); see [12]. The proof of each theorem was done independently from others to model a distributed proof development.

Therefore, we simulated the following scenario. Suppose a new developer tackles for the first time the proof of Lemma 5, and he knows the general methodology to prove it and has access to the library of programs previously proven by other users. This situation is similar to the one presented in Subsection 2.1 with an additional problem: the different notation employed by different users obscure some common features. ML4PG would be a good alternative to the manual search for proof patterns.

Let us focus on the first step of the methodology – that is, the proof of the equivalence between the specification of the factorial function (which is already defined in SSReflect) and the algorithm. The Java factorial function is an iterative function; and the algorithm is written in Coq as a tail recursive function, see the right side of Figure 1. In general, all the tail recursive functions are defined using an auxiliary function, called the *helper*, and a wrapper for such a function. The suggestions provided by ML4PG in this case are the proofs of step (1) for three iterative programs: the multiplication, the exponentiation and the power of natural numbers. All of them follow the same proof strategy which can be also applied in the case of factorial:

**Proof Strategy 2** *Prove an auxiliary lemma about the helper considering the most general case. For example, if the helper function is defined with formal parameters n, m, and a, and the wrapper calls the helper initializing a at* 0*, the*

*helper theorem must be about* `(helper n m a)`*, not just about the special case* `(helper n m 0)`*. Subsequently, instantiate the lemma for the concrete case.*

Discovery of proofs following Proof Strategy 2 among 147 JVM library lemmas will be the second benchmarking task for ML4PG.

## 3 Finding Proof Strategies with ML4PG

In this section, we show how ML4PG discovers Proof Strategies 1 and 2, thereby giving a brief introduction to proof-pattern recognition with ML4PG; full details of implementation can be found in [12, 16].

The discovery of statistically significant features in data is a research area of its own in machine-learning, known as *feature extraction*, see [4]. Statistical machine-learning algorithms classify given examples seen as points in an $m$-dimensional space, where $m$ is the maximum number of features each example may be characterised by. Irrespective of the particular feature extraction algorithm used, most pattern recognition tools [4] will require that the number of selected features is limited and fixed. The exception to this is a special class of methods called "sparse" methods, where the number of features is either extremely large, or large enough to be considered unlimited; see [5]. In Section 4.3, we incorporate sparse learning in ML4PG and evaluate their efficiency; but for now, we stay with the classic assumption of the fixed-size feature vectors.

ML4PG has its own method of feature extraction, called the *proof-trace method*. It allows us to capture a potentially infinite variety of lemma shapes, by means of gathering a fixed number of statistical features.

*Example 2.* Table 1 shows the proof for Lemma 2, and Table 2 shows the statistical features gathered from its proof according to ML4PG's algorithm of proof-trace feature extraction [16].

With the *proof-trace method*, the lemma structure "shows itself" through some simple statistics of the proof steps it induces. Note that, using the two dimensions of Table 2, we gather statistics both *dynamically* (considering corelation of features within several proof steps) and *relationally* (tracking correlation of sub-goal shapes and user actions). An advantage of this method is that, due to the simplicity of every single statistical feature, it applies uniformly to any Coq library, irrespective of proof complexity. ML4PG extracts all proof features during Coq compilation.

Once all proof features are extracted, ML4PG is ready to communicate with *machine-learning interfaces*. ML4PG is built to be modular – that is, the feature extraction is first completed within the Proof General environment, where the data is gathered in the format of hash tables, and then these tables are converted to the format of the chosen machine-learning tool (in our case, MATLAB or Weka). Extending the list of machine-learning engines does not require any further modifications to the feature extraction algorithm, but just defining new *translators* for those engines. There is a synchronous communication between

| | *tactics* | *N tactics* | *arg type* | *arg is hyp?* | *top symbol* | *n subgoals* |
|---|---|---|---|---|---|---|
| *g1* | elim | 1 | nat | Hyp | forall | 2 |
| *g2* | rewrite | 1 | Prop | `big_nat1` | equal | 0 |
| *g3* | rewrite | 1 | 8×Prop | $EL'$ | equal | 1 |
| *g4* | move:; rewrite; move/ | 3 | 3×Prop | $EL''$ | equal | 1 |
| *g5* | rewrite | 1 | Prop | `sub0r` | equal | 0 |

**Table 2.** ***A Table illustrating the work of ML4PG's feature extraction algorithm for the proof in Table 1.*** *Parameters inside the double lines are the extracted features used to form the* ***feature vectors****. Notation* g1–g5 *is used to denote five consecutive subgoals in the derivation. Columns are the properties of subgoals ML4PG tracks: the names of applied tactics, their number, types of arguments, link of the proof step to a hypothesis (Hyp), inductive hypothesis or a library lemma; top symbol of the current goal, and the number of the generated subgoals. Where we use notation $EL'$, ML4PG gathers the lemma names (`sumrB`, `big_nat_recr` , `big_nat_recl`, ...); where we use $EL''$, ML4PG gathers the lemma names (`eq_refl`, `subr_eq0`, `eqP`).*

ML4PG and the machine-learning interfaces, which run in the background waiting for ML4PG calls. The user can chose additional proof libraries to be clustered against the current proof. ML4PG provides output instantly.

ML4PG offers a choice of *pattern-recognition algorithm*. We connected ML4PG only to *clustering algorithms* [4] – a family of *unsupervised learning methods*. Clustering techniques divide data into $n$ groups of similar objects (called clusters), where the value of $n$ is provided by the user. Unsupervised learning is chosen when no user guidance or class tags are given to the algorithm in advance: in our case, we do not expect the user to "tag" the library proofs in any way. There are several clustering algorithms available in MATLAB (*K-means* and *Gaussian*) and Weka (*K-means*, *FarthestFirst* and *Expectation Maximisation*, in short E.M.); we will use several of them in the coming examples.

*Example 3 (Benchmark 1).* ML4PG will show a window with Lemmas 2, 3 and 4 when proving Lemma 1, if we ask it to consider five SSReflect libraries for clustering: `bigop` (devoted to generic indexed big operations like $\sum_{i=0}^{n} f(i)$ or $\bigcap_{i \in I} f(i)$), `matrix`, `binomial` (which defines combinatorial notions and include several results involving summations), a small library about summations (which includes the proof of Lemma 2) and the library about Persistent Homology formalised in [11]. These libraries involve 758 lemmas for ML4PG to analyse.

Table 3 shows the results that ML4PG will obtain, if the user varies the clustering algorithms and the size of clusters when calling ML4PG within the proof environment; we also highlight the exact solution to Benchmark-1 task.

Various numbers of clusters can be useful: this may depend on the size of the data set, and on existing similarities between the proofs. ML4PG has its own algorithm that determines the optimal number of clusters interactively, and based on the library size. As a result, the user does not provide the value of $n$ directly, but just decides on *granularity* in the ML4PG menu, by selecting a value

| Algorithm: | $g = 1$ ($n = 75$) | $g = 2$ ($n = 84$) | $g = 3$ ($n = 94$) | $g = 4$ ($n = 108$) | $g = 5$ ($n = 126$) |
|---|---|---|---|---|---|
| Gaussian | $20^{a-d}$ | $15^{a-d}$ | $11^{a-d}$ | $2^{b,d}$ | 0 |
| K-means (Matlab) | $53^{a-d}$ | $23^{a-d}$ | $16^{a-d}$ | $\mathbf{4}^{a-d}$ | 0 |
| K-means (Weka) | $66^{a-d}$ | $42^{a-d}$ | $15^{a-d}$ | $\mathbf{4}^{a-d}$ | $2^{b,d}$ |
| E.M. | $25^{a-d}$ | $50^{a,b,d}$ | $40^{a,b,d}$ | $10^{b,c,d}$ | $30^{a,b,d}$ |
| FarthestFirst | $75^{a-d}$ | $40^{a-d}$ | $30^{a-d}$ | $24^{a-d}$ | $20^{a-d}$ |

**Table 3. Clustering experiments discovering Proof Strategy 1,** for a choice of algorithms. When $g$ is chosen by the user, ML4PG dynamically calculates the number of clusters $n$; the table shows the size of the single cluster that ML4PG displays to the user for every choice of $g$. We mark if the cluster contains: $a$) Lemma 1, $b$) Lemma 2, $c$) Lemma 3 and $d$) Lemma 4. The size of the data set is 758 lemmas, in bold is the cluster that contains exactly the three benchmark lemmas for ML4PG to display.

between 1 and 5, where 1 stands for a low granularity (producing fewer large clusters) and 5 stands for a high granularity (producing many smaller clusters).

*Example 4 (Proof Granularity).* As Table 3 shows, one can use a "bottom-up approach" starting with the granularity value of 1 and increasing that value in successive calls. Using the default value $g = 3$ and e.g. K-means (Weka) algorithm, ML4PG obtains 15 suggestions related to lemmas about summations including Lemmas 2 and 3. Increasing the granularity level to 4, ML4PG discovers that Lemma 1 is similar to Lemmas 2, 3, and 4. Finally, increasing the granularity level to the maximum level of 5, ML4PG just discovers that Lemmas 1 and 4 are similar.

*Example 5 (Benchmark 2).* ML4PG will correctly suggest similar lemmas to Lemma 5 in the libraries for multiplication, exponentiation and power. For this, we consider 15 libraries for clustering related to formalisations of arithmetic JVM programs. These libraries involve 147 lemmas for ML4PG to analyse. Table 4 shows the results for different choices of algorithms and parameters, and we highlight the exact match between the benchmark task and the ML4PG result. In case the user is unsure of the optimal machine-learning parameters, we recommend to use a "top-down approach". The highest granularity level does not produce any result. But, if we decrease the granularity level to 4, ML4PG spots some interesting similarities. If this is not enough to discover Proof Strategy 2; one can decrease the granularity level to 3, for which ML4PG discovers exactly the four Benchmark-2 lemmas.

In conclusion, we note that it is the nature of statistical methods to produce results with some "probability", and not being able to provide "guarantees" that a certain cluster will be found for a certain library. However, ML4PG ensures quality of the output in several different ways. First of all, the results shown in Tables 3 and 4 are not taken from one random run of a clustering algorithm – instead, ML4PG output shows the digest of clustering results coming from 200

| Algorithm: | $g=1$ $(n=14)$ | $g=2$ $(n=16)$ | $g=3$ $(n=18)$ | $g=4$ $(n=21)$ | $g=5$ $(n=24)$ |
|---|---|---|---|---|---|
| Gaussian | $10^{a-d}$ | $10^{a-d}$ | 0 | 0 | 0 |
| K-means (Matlab) | $21^{a-d}$ | $\mathbf{4}^{a-d}$ | $\mathbf{4}^{a-d}$ | $2^{c,d}$ | 0 |
| K-means (Weka) | $30^{a,b,d}$ | $11^{a,b,d}$ | $3^{a,b,d}$ | 0 | 0 |
| E.M. | $21^{a-d}$ | $7^{a-d}$ | $7^{a-d}$ | 0 | 0 |
| FarthestFirst | $28^{a-d}$ | $25^{a-d}$ | 0 | 0 | 0 |

**Table 4. A series of clustering experiments discovering Proof Strategy 2.** The table shows the sizes of clusters containing: *a*) Lemma about JVM multiplication program, *b*) Lemma about JVM power program, *c*) Lemma about JVM exponentiation program, and *d*) Lemma about JVM factorial program. The size of the data set is 147 lemmas, in bold is the cluster that finds exactly the four benchmark examples.

runs of the clustering algorithm. Only clusters that appear frequently in 200 runs of the algorithm are displayed to the user. There is a way to manipulate the frequency threshold within ML4PG. Another measure – relative proximity of examples in a cluster – is also taken into account by ML4PG before the results are shown, see [12,16]. If a lemma is contained in several clusters, proximity and frequency values are used to determine one "most reliable" cluster to display.

It is very encouraging that, with all variations of the learning algorithms and parameters shown in Tables 3 and 4, ML4PG is consistently grouping the correct lemmas into clusters, albeit with varied degree of precision. Judging by the experiments, Gaussian and K-means algorithms are the most reliable; they show very stable results; asking the user the very minimum effort of adjusting the granularity parameter to obtain the result of required precision. ML4PG is very fast and give instant outputs allowing the user to have quick search/evaluation.

## 4 Search Patterns in Coq/SSReflect

In this section, we look broader at ML4PG's functionality relative to the range of proof-pattern search methods already available in Coq/SSReflect. We then produce two original extensions to ML4PG that allow us to analyse proof patterns of varied scale: from small patches in a big proof, to proofs in their entirety.

### 4.1 Symbolic Methods

Coq provides comprehensive symbolic search mechanisms to browse the corpus of results available in its libraries. There are 4 commands in Coq to find definitions and theorems: `Search` , `SearchAbout`, `SearchPattern` and `SearchRewrite` [8]. In addition, SSReflect implements its own version of the `Search` command [10] – SSReflect's `Search` subsumes the four Coq's search mechanisms. Related is the Whelp platform [1] – a web search engine for mathematical knowledge formalised in Coq, which features 3 functionalities: `Match` (similar to Coq's `Search` command), `Hint` (that finds all the theorems which can be applied to derive the current goal) and `Elim` (that retrieves all the eliminators of a given type).

In all these tools, the search focuses on lemma/definition statements only, as opposed to searching proof patterns in ML4PG. In the lemma statements, they search for "local" patterns, such as names, types, operators or combination of them. The main methods behind them are algorithms of unification and matching, which have a more deterministic outcome than ML4PG: the searching engine will always find all lemmas that can be exactly matched against the user-defined searching pattern. The results produced by these tools are most useful in situations when the user knows what kind of a pattern he is searching for.

*Example 6.* In the proof of Lemma 1, we may want to introduce the term $(1-M)$ inside the sum $\sum_{i=0}^{n-1} M^i$, but if we have no experience with the bigop library of SSReflect, it is likely that we do not know the name of the concrete lemma. The lemma which carries out this task can be found using, for instance, the commands `Search "distr"in bigop`, that finds the lemmas of the `bigop` library containing `"distr"` in their name, or `Search (_ * (\big[_/_]_(_ <- _)_))`, that finds a particular statement pattern.

However, these tools are less useful when the user is unsure about how to continue the proof. Even if they can be used, for instance, to find all the theorems applicable to derive the current goal (`Hint` mechanism of Whelp), they cannot find a "global" proof pattern (like e.g. Proof Strategy 1) to tackle the proof of a concrete goal.

*Example 7.* In Benchmark 1, a search of all the lemmas involving big operations produce more than 300 lemmas. If we refine the search to lemmas involving $\sum$, we can find 250 lemmas. If we search lemmas about the multiplication of matrices, the Search command returns more than 120 results. The command `Search "distr"` obtains almost 100 lemmas.

Although the symbolic search is deterministic, it can be helpless for discovering more general or less trivial patterns. Refinement of the results produced by the symbolic search mechanisms involves some kind of creativity in the input of these tools. In contrast, ML4PG can help to find those "meta" proof patterns but cannot be guaranteed to find the smaller, "local" patterns as well as the standard tools do.

### 4.2 ML4PG and Proof Patch Method

To access the "meta" proof patterns, that is, patterns that cannot be found by unification and matching only, ML4PG analyses proofs in addition to lemma statements, cf. proof-trace method of Section 3. Notably, some information used in the standard searching mechanisms is also implicitly considered (for instance, lemma names, types and top-level operators). Section 3 illustrated that this method can be successful. However, there is still room for improvements. We designed and implemented two new techniques to obtain better results: the *proof-patch method* and *dynamic lemma numbering*, now available at [13].

Most pattern recognition tools require the numbers of features to be limited and fixed. However, Coq proofs may have varied length. The first ML4PG prototype [16] considered just the first five proof steps, cf. Table 2, inevitably losing some information. The *proof-patch* method tackles this problem. The underlying idea of this technique is that one small proof may potentially resemble a fragment of a bigger proof; also, various small "patches" of different big proofs may resemble. The proof-patch method automatically analyses many fragments of a big proof, using the feature extraction mechanism of Table 2. If the proof is larger than 5 steps, it will form separate feature vectors for proof steps 6–10, 11–15 and so on. Additionally, ML4PG always extracts features from the last five proof steps the user makes, whether the proof is finished or not.

*Example 8 (Benchmark 1).* Repeating the experiments of Example 3 with the proof-patch method, we obtain the results presented in Table 5. It turns out that Proof Strategy 1 can be applied twice in the proof of Lemma 3, firstly in the inner summation and subsequently in the remaining proof. Then, two of the suggestions provided by ML4PG in this case come from the proof of Lemma 3.

| Algorithms: | $g = 1$ | $g = 2$ | $g = 3$ | $g = 4$ | $g = 5$ |
|---|---|---|---|---|---|
| Gaussian | $21^{a-e}$ | $16^{a-e}$ | $12^{a-e}$ | $2^{b,e}$ | 0 |
| K-means (Matlab) | $83^{a-e}$ | $31^{a-e}$ | $21^{a-e}$ | $\mathbf{5}^{a-e}$ | 0 |
| K-means (Weka) | $76^{a-e}$ | $51^{a-e}$ | $16^{a-e}$ | $4^{a,b,c,e}$ | $2^{b,e}$ |
| Expectation Maximisation | $26^{a-e}$ | $51^{a,b,d,e}$ | $41^{a,b,e}$ | $11^{b,c,d,e}$ | $31^{a,b,e}$ |
| FarthestFirst | $81^{a-e}$ | $48^{a-e}$ | $31^{a-e}$ | $25^{a-e}$ | $21^{a-e}$ |

**Table 5. A series of clustering experiments discovering Proof Strategy 1 using the proof-patch method.** The table shows sizes of clusters containing: *a*) Lemma 1, *b*) Lemma 2, *c*) Lemma 3 inner sum, *d*) Lemma 3 outer sum and *e*) Lemma 4.

This technique can be even more useful when rare patterns of interest appear for the first time in late stages of a big proof.

The second improvement is *dynamic lemma numbering*. This technique is related to how the information about external lemmas is recorded in the feature vectors. Machine learning algorithms expect numerical feature vectors as inputs; therefore, ML4PG converts all features into numbers, see [16]. In [16], the conversion of lemma names was blind, assigning unique integers to lemmas in order of their appearance in the library. This incremental lemma numbering had an effect on clustering results, specially if ML4PG worked with big proof libraries, where the lemma numbering gave a big value spread.

To solve this problem, ML4PG now assigns closer numbers to similar lemmas. During the Coq compilation and feature extraction, ML4PG starts with numbering first two lemmas Coq compiles, and then continues inductively to cluster more lemmas one by one, and renumber them according to the cluster proximity of one proof to another.

*Example 9 (Dynamic Lemma Numbering).* For the JVM benchmark, Table 4 already incorporates dynamic lemma numbering. However, without it, the Lemma *c)* (proof for exponentiation) disappears from certain clusters. This happens because all clustered lemmas used different auxiliary lemmas. With blind lemma numbering, the auxiliary lemmas appearing later in the development were represented by numbers distant from similar lemmas in e.g. the first library processed by ML4PG. Dynamic lemma numbering addresses this problem.

### 4.3 Sparse ML4PG

Moving from specific and lower-level patterns to global patterns, we introduce the last extension to ML4PG – a new *sparse* method to deal with whole proofs without analysing the patches. This method extends the rows of Table 2 to include all goal steps in a proof. Moreover, if a cell contains information about several features (cf. tactics for row *g4* in Table 2), this feature is split into many separate features, see Table 6.

| | ... | *tactic 1* | *tactic 2* | *tactic 3* | *N tactics* | *arg type 1* | *arg type 2* | *arg type 3* | ... |
|---|---|---|---|---|---|---|---|---|---|
| *g1* | ... | elim | - | - | 1 | nat | - | - | ... |
| *g2* | ... | rewrite | - | - | 1 | Prop | - | - | ... |
| *g3* | ... | rewrite | - | - | 1 | Prop | Prop | Prop | ... |
| *g4* | ... | move: | rewrite | move/ | 3 | Prop | Prop | Prop | ... |
| *g5* | ... | rewrite | - | - | 1 | Prop | - | - | ... |

**Table 6.** ***A fragment of the table illustrating the work of ML4PG's sparse feature extraction algorithm for the proof in Table 1.*** *The empty cells of the table are indicated with the symbol -.*

Sparse feature vectors may contain many 0-features, especially as big proofs determine the size of feature vectors for all (even short) proofs. The rest of the functionality of ML4PG presented in the introduction remains unchanged; however, the only available clustering method that can handle the sparse feature vectors is now K-means algorithm implemented in MATLAB.

*Example 10 (Benchmarks 1 and 2).* Table 7 shows the results obtained by ML4PG using the sparse method. For Benchmark 2, the results are similar to the best results obtained using the proof-trace method, see Table 4, so it actually adds precision. For Benchmark 1, it fails completely.

The main difference in the sparse clustering for Benchmark 1 and 2 is the extent to which sparsity increases when we apply the new algorithm of feature extraction. As proofs in the Benchmark 2 examples have very regular shapes and sizes, the resulting "sparse" feature vectors (size 66) do not differ much from the initial dense vectors implemented in ML4PG (size 30), but marginally improve representation of external auxiliary lemmas. For Benchmark 2, where the proofs

| | $g=1$ | $g=2$ | $g=3$ | $g=4$ | $g=5$ |
|---|---|---|---|---|---|
| Benchmark 1 | 0 | 0 | 0 | 0 | 0 |
| Benchmark 2 | $11^{a-d}$ | $4^{a-d}$ | $4^{a-d}$ | 0 | 0 |

**Table 7. Sparse clustering for Benchmarks 1 and 2, using K-means algorithm and dynamic lemma numbering.** Notation (a-d) follows Table 4.

are much less regular, sparsity has the opposite effect: the sparse algorithm of feature extraction increases the number of features from 30 to 355 – and this translates into 12-fold increase in the dimensions of the feature space over which classification is performed; this decreases classification precision drastically.

From the efficiency point of view, the sparse method is slower than the other two methods available in ML4PG (proof-trace and proof-patch). This is due to the fact that the sparse mechanism must standardise the size of all the feature vectors (this can involve hundreds of vectors); on the contrary, this step is not necessary in the other two methods. In addition, clustering algorithms get slower when increasing the number of features and the sparse method produces vectors with hundreds of features.

As a summary, sparse methods have the following disadvantages in ML4PG setting: they will require either regular (as in Benchmark 2) or very large data sets to be accurate, whereas we wish ML4PG to be versatile, and robust with small as well as big libraries, without any restrictions on proof regularity. The substantial slow-down of the sparse methods defies the idea of fast interactive communication between the user, ML4PG, and Coq compiler. Finally, not all clustering algorithms can in principle be adapted to sparsity, e.g. a very accurate Gaussian algorithm requires a certain ratio between the size of the feature space and the number of examples.

## 5 Conclusions and Related work

We have evaluated ML4PG proof-pattern recognition ability. The main conceptual conclusion of our evaluation is that statistical methods offer a very different perspective on proof pattern recognition compared to the standard symbolic methods of proof-pattern search. Notably, they introduce a trade-off between determinism of the output and its conceptual novelty. That is, as our experience shows, ML4PG can find surprisingly complex and interesting patterns across several libraries – patterns that one would not find by exact matching of lemma shapes, operators, or types. On the other hand, ML4PG comes with no guarantees as to which patterns it will find and under which settings.

Currently, we see the speed and interactive nature of ML4PG as the main answer to this problem. It takes seconds for the user to query ML4PG, adjust its parameters; and decide whether the hint is useful. We finish with a brief survey of alternative machine-learning methods used in automated and interactive proofs.

**Machine-learning for Computer Proofs: related work.** Two styles of machine-learning can be useful in proof analysis, coming from *symbolic* AI and *statistical* AI.

*Symbolic machine-learning* (e.g. Inductive Logic Programming and *Rippling* techniques [3]) uses specification languages close to the prover language. This approach includes techniques such as lemma generation [15] and proof strategy discovery [3]. The systems following this trend are more deterministic and "algorithmic" than statistical AI, in the sense that they are built to automatically formulate new lemma statements or hypotheses. However, similarly to the Halting Problem argument, the algorithmic approach has its limits: the algorithmic tools are often tailored for certain fragments of first order language or a certain library, or a certain proof shape, but not for all envisaged proofs in general.

*Statistical machine-learning* methods can discover data regularities on the basis of numeric proof features, but they have very weak power for generalisation of this knowledge. For instance, ML4PG only "shows" the families of related proofs to the user, but does not formulate new theorems.

ML4PG could be the starting point of a future "larger" system that incorporates the symbolic style on top of statistical style – for instance, the proof families obtained with ML4PG could be then conceptualised into new lemmas or strategies like Proof Strategy 1 and 2. Nevertheless, up to now, in the trade-off between unlimited generality, variety and size of proofs, and the symbolic conceptualisation of the output, we have chosen the former. As a result, ML4PG interactively shows lemma similarities for any imaginable set of higher-order proofs in Coq/SSReflect, but stops there.

Among other successful statistical tools is the method of statistical proof-premise selection [17, 18]. Applied in several automated provers, it provides statistical ratings of existing lemmas; and this makes automated rewriting more efficient. There are several differences between premise-selection tools [17, 18] and ML4PG:

- *Aim.* Premise-selection tools aim to increase the number of goals which are proven automatically by automated provers. On the contrary, ML4PG assists the user, rather than the prover: the user may treat the suggested similar lemmas as proof hints.
- *Features.* Premise-selection tools extract features from the statements of first-order formulas. In the case of ML4PG, features are extracted from higher-order formulas and proofs.
- *Feature vectors representation.* Feature vectors of premise-selection tools capture statistics relative to every symbol in the library; as a result – they are sparse, including up to $10^6$ features. ML4PG can deal with sparse representations, but it performs best with standard, dense machine-learning methods; as proof-trace and proof-patch methods provide an intelligent scaling for big libraries. In [17, 18], scaling is done at the level of machine-learning algorithms.
- *Machine-learning methods.* Premise-selection tools work with *supervised learning* methods. 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 $< B >$. After such training, the classifier $< B >$ can predict whether a new lemma $C$ requires the lemma $B$ in its proof.
ML4PG does not use any a priori given training labels; instead, it uses unsupervised learning (clustering).

## References


1. A. Asperti, F. Guidi, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. A Content Based Mathematical Search Engine: Whelp. In *Post-Proceedings of the TYPES'04 International Conference*, volume 3839 of *LNCS*, pages 17–32, 2006.
2. D. Aspinall. Proof General: A Generic Tool for Proof Development. In *Proceedings of TACAS'00*, volume 1785 of *LNCS*, pages 38–43, 2000.
3. D. Basin, A. Bundy, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
4. C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
5. A. Blum. Learning boolean functions in an infinite attribute space. *Machine Learning*, 9(4):373–386, 1992.
6. R. Brualdi and H. Ryser. *Combinatorial Matrix theory*. Cambridge University Press, 1991.
7. A. Bundy, D. Hutter, C. Jones, and J S. Moore. AI meets Formal Software Development (Dagstuhl Seminar 12271). *Dagstuhl Reports*, 2(7):1–29, 2012.
8. Coq development team. The Coq Proof Assistant Reference Manual, version 8.4. Technical report, 2012.
9. G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *Proceedings 4th Conference on Interactive Theorem Proving (ITP'13)*, LNCS, 2013.
10. G. Gonthier and A. Mahboubi. An introduction to small scale reflection. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
11. J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect. *To appear on Transactions on Computational Logic*, 2013.
12. J. Heras and E. Komendantskaya. ML4PG: downloadable programs, manual, examples, 2012–2013. `www.computing.dundee.ac.uk/staff/katya/ML4PG/`.
13. J. Heras and E. Komendantskaya. ML4PG in Computer Algebra Formalisations. In *Proceedings of CICM 2013*, LNCS, 2013. `http://arxiv.org/abs/1302.6421`.
14. J S. Moore. *Models, Algebras and Logic of Engineering Software*, chapter Proving Theorems about Java and the JVM with ACL2, pages 227–290. IOS Press, 2004.
15. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
16. E. Komendantskaya, J. Heras, and G. Grov. Machine Learning for Proof General: interfacing interfaces, 2012. 25 pages. Accepted for post-proceedings of User Interfaces for Interactive Theorem Proving (UITP'12), arXiv:1212.3618.
17. D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban. MaSh: Machine Learning for Sledgehammer. In *Proceedings of ITP'13*, LNCS, 2013.
18. D. Kühlwein et al. Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics. In *Proceedings of IJCAR'12*, volume 7364 of *LNCS*, pages 378–392, 2012.
19. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification: Java SE 7 Edition, 2012.