Machine Learning of Coq Proof Guidance: First Experiments

Cezary Kaliszyk[1], Lionel Mamane[2], and Josef Urban[3]

[1] University of Innsbruck, Austria
cezary.kaliszyk@uibk.ac.at
[2] Luxembourg
lionel@mamane.lu
[3] Radboud Uniersity Nijmegen, Netherlands
josef.urban@gmail.com

Abstract

We report the results of the first experiments with learning proof dependencies from the formalizations done with the Coq system. We explain the process of obtaining the dependencies from the Coq proofs, the characterization of formulas that is used for the learning, and the evaluation method. Various machine learning methods are compared on a dataset of 5021 toplevel Coq proofs coming from the CoRN repository. The best resulting method covers on average 75% of the needed proof dependencies among the first 100 predictions, which is a comparable performance of such initial experiments on other large-theory corpora.

1 Introduction

In the last decade, a number of bridges between interactive theorem provers (ITPs) and various external proof-advising systems have been built. The best-known examples of such systems are today the bridges between ITPs such as Isabelle, Mizar and HOL Light and automated theorem provers (ATPs) such as Vampire, E, and Z3. The success rate of such bridges for Mizar and HOL Light has recently reached 40% in a scenario that assumes no assistance from the users, only the ability to learn from the previous proofs that are already available in the large libraries of these ITP systems. Important components of these systems are the premise-selection algorithms, i.e., algorithms that for a given conjecture choose the most relevant theorems and definitions that are already available in the large libraries. Such algorithms can be actually used separately as a search engine available to the ITP users in cases when the full automated proof is too hard to find. Such functionality has been implemented early for the Mizar system and integrated in its Emacs authoring environment in the form of the Mizar Proof Advisor [14]. Good premise selection is thus a prerequisite for the fully automated AI/ATP systems, and also a functionality that can be immediately given to the ITP users.

In this work, we develop premise selection techniques for the Coq proof assistant and evaluate their performance on the Constructive Coq Repository at Nijmegen (CoRN) [2], containing 5021 toplevel proofs of theorems such as the Fundamental Theorem of Algebra [3]. The necessary work involves extraction of proof dependencies, for which we re-use previous work targeted at Coq user interfaces by Mamane, see Section 2. In the type-theoretical context used by Coq there is no clear distinction between theorems and definitions on one side, and formula symbols and terms on the other side. This makes it difficult to see from the extracted dependency data which of the dependencies characterize the conjecture to be proved, and which of them are the "proper" proof dependencies that we would like to recommend to the user (and ultimately to an ATP system). Section 3 explains how we heuristically differentiate between these two kinds of data. Section 4 briefly explains how we learn premise selection from such data, and Section 5 presents the first experimental results. It is shown that the best resulting method covers on average 75% of the needed proof dependencies among the first 100 predictions, which is a comparable performance of such initial experiments on other large-theory corpora. Section 6 discusses future work and concludes.

2 Extracting Coq Dependencies

The processs of extracting proof dependencies from Coq developments that we use in this work is described in detail in [1]. Here we give a brief overview of this process and related issues.

Coq is based on the Curry-Howard isomorphism, which means that statements (formulas) are encoded as types, and at the logical level there is no essential difference between a definition and a theorem: they are both just the binding (in the environment) of a name to a type (type of the definition, statement of the theorem) and a term (body of the definition, proof of the theorem). Similarly, there is no essential difference between an axiom and a parameter: they are both just the binding (in the environment) of a name to a type (statement of the axiom, type of the parameter, e.g. "natural number").

As part of the development of tmEgg [11], Mamane added to Coq a partial facility to communicate proof dependencies to the user interface. This facility has been already integrated in the official release of Coq. Since then this facility was extended to be able to treat the whole of the CoRN library, however these changes are not yet included in the official release of Coq. This facility works by tracking dependencies for various Coq commands. There are essentially three groups of Coq commands that need to be treated:

1. Commands that register a new logical construct (definition or axiom), either
   - From scratch. That is, commands that take as arguments a name and a type and/or a body, and that add the definition binding this name to this type and/or body. The canonical examples are

     Definition Name : type := body

     and

     Axiom Name : type

     The type can also be given implicitly as the inferred type of the body, as in

     Definition Name := body
   - Saving the current (completely proven) theorem in the environment. These are the "end of proof" commands, such as `Qed`, `Save`, `Defined`.
2. Commands that make progress in the current proof, which is necessarily made in several steps:

   (a) Opening a new theorem, as in

   Theorem Name : type

   or

   Definition Name : type

   (b) An arbitrary strictly positive amount of proof steps.

   (c) Saving that theorem in the environment.

   These commands update (by adding exactly *one* node) the internal Coq structure called "proof tree".

3. Commands that open a new theorem, that will be proven in multiple steps.

The dependency tracking is implemented as suitable hooks in the Coq functions that the three kinds of commands eventually call. When a new construct is registered in the environment, the dependency tracking walks over the type and body (if present) of the new construct and collects all constructs that are referenced. When a proof tree is updated, the dependency tracking examines the top node of the new proof tree (note that this is always the only change with regards to the previous proof tree). The commands that update the proof tree (that is, make a step in the current proof) are called `tactics`. Coq's tactic interpretation goes through three main phases:

1. parsing;
2. Ltac[1] expansion;
3. evaluation.

The tactic structure after each of these phases is stored in the proof tree. This allows to collect all construct references mentioned at any of these tree levels. For example, if tactic `Foo T` is defined as

```
try apply BolzanoWeierstrass;
solve [ T | auto ]
```

and the user invokes the tactic as `Foo FeitThompson`, then the first level will contain (in parsed form) `Foo FeitThompson`, the second level will contain (in parsed form)

```
try apply BolzanoWeierstrass;
solve [ FeitThompson | auto ].
```

and the third level can contain any of:

- `refine (BolzanoWeierstrass ...)`,
- `refine (FeitThompson ...)`,
- something else, if the proof was found by `auto`.

The third level typically contains only a few of the basic atomic fundamental rules (tactics) applications, such as `refine`, `intro`, `rename` or `convert`, and combinations thereof.

Dependency tracking is available in the program implementing the `coq-interface` protocol which is designed for machine interaction between Coq and its user interfaces. The dependency information is printed in a special message for each progress-making Coq command that can give rise to a dependency. After some postprocessing, the dependency data obtained from such commands will look as follows (first comes the defined/proved construct, then a list of its dependencies):

```
"CoRN.algebra.Basics.NEG_anti_convert"
  ("Coq.Init.Datatypes.S" "Coq.Init.Datatypes.nat" "Coq.Init.Datatypes.nat_ind"
   "Coq.Init.Logic.eq" "Coq.Init.Logic.refl_equal" "Coq.NArith.BinPos.P_of_succ_nat"
   "Coq.NArith.BinPos.Psucc" "Coq.NArith.BinPos.xH" "Coq.ZArith.BinInt.Z"
   "Coq.ZArith.Bi nInt.Z_of_nat" "Coq.ZArith.BinInt.Zneg" "Coq.ZArith.BinInt.Zopp")
```

[1]Ltac is the Coq's tactical language, used to combine tactics and add new user-defined tactics.

3 Learning Data

In order to use machine learning for guessing proof dependencies, two sets of data need to be extracted from a proof assistant library: suitable features of the theorems and their proof dependencies. Machine learning for premise selection (i.e., for guessing the necessary proof dependencies) has so far been successfully applied and evaluated for systems based on set theory (MizAR [15]) and higher order logic (Sledgehammer:MaSh [10], HOL(y)Hammer [9, 8]). In both kinds of systems, the available concepts are clearly divided in different categories by the implementation: In HOL-based systems types, type-classes and function symbols are all distinct from theorems. In particular, theorems can not appear inside terms. Similarly in Mizar, the meta-logic functions, predicates, modes, and registrations are separate from the defined set-theory objects.

This is no longer true in type-theory based systems which implement the Curry-Howard correspondence treating propositions as types, and proofs as terms of such (proposition) types. Each time a named theorem T is used in a proof of theorem S, we are actually using (a reference to) the (named) proof term of T. Such named proof terms are in `Coq` treated the same way as any other defined constants. It is not possible to distinguish named theorems from other defined constants without checking whether the type of the constant is of a propositional type. This is however not sufficient for us: the user may state (as a goal) and prove the existence of values in any type (such as `nat->nat`), and premise selection should also help in such non-propositional proofs. The users may even define their own proposition type (different from the default type `Prop`) and this is actually done in some `Coq` developments, for example in the case of the `CProp` used pervasively in `CoRN`.

The most straightforward way of adapting the machine learning infrastructure to `Coq`, could be as follows. For each defined constant T, the features $F(T)$ of T would be the set of defined constants present in the type (statement) of T, and the dependencies $D(T)$ of T would be the set of defined constants used in the proof term of T. This has however one obvious disadvantage: all theorems that talk about natural numbers would include `nat`, `0`, and `Suc` as their dependencies. Learning such constants would also predict them, however they are currently useless in automated techniques. To avoid learning such constants as dependencies, we first globally (for a given corpus C, such as `CoRN`) partition the set of all defined constants into those that may be used as features (F_C), and those that may be used as dependencies (D_C), using the following heuristic. The set F_C (allowed features) will consist of all the defined constants that appear in the types of theorems (and definitions, etc.) of C. The set D_C (allowed dependencies) will consist of all the defined constants that appear in all the proof terms, minus the set F_C.

Given a named theorem T, $F(T)$ is then defined as the elements of F_C present in the type (statement) of T, while $D(T)$ are the elements of D_C used in the proof term of T. This heuristic might not work well in the case where necessary propositional constants also appear in the types (statements) of theorems; however it works well with most `Coq` developments, in particular we have manually checked (on a random subset) that it behaves well with the `CoRN` development, which we focus on in this paper.

Table 1 shows some statistics for the machine learning data obtained. These are comparable to the data for the initial experiments done for other systems [5, 4]. Note that we do not yet work with more advanced features such as terms, subterms, patterns, etc.

Accessible part of Coq Standard Library:	5099 available facts
Evaluated CoRN proofs	5021 defined (proved) theorems
Distinct features	2683
Average number of features in a CoRN theorem	9.82
Average number of dependencies for a CoRN theorem	11.27

Table 1: Statistics for the machine learning data

4 Machine Learning

In the evaluations we use our custom implementations of three machine learning algorithms and their combinations. These are k-Nearest Neighbours [7], sparse Naive Bayes [10], and our modified version of the Meng-Paulson (MePo) relevance filter [12]. In order to combine the results of the algorithms we consider the Ensemble methods [13], in particular the weighted harmonic average of the classifications [6]. All these methods are implemented as fast tools (written either in OCaml or in C++) that are external to the Coq implementation.

k-Nearest Neighbours: The (distance-weighted) k-Nearest Neighbours learning algorithm first finds a fixed number (k) of proved facts nearest (in its feature representation) to the conjecture c, and then weights the dependencies of each such fact f by the distance between f and c. The final ranking (relevance) of dependencies is obtained by summing their weights across the k nearest neighbors.

Naive Bayes: Naive Bayes is a statistical learning method based on Bayes theorem with a strong (or naive) independence assumption. Given a conjecture c and a fact f, naive Bayes computes the probability of f being needed to prove c, based on the previous use of f in proving conjectures that are similar to c. The similarity is in our case expressed using the features F of the formulas. The independence assumption says that the (non-)occurrence of a feature is not related to the (non-)occurrence of every other features. The predicted relevance of a fact f with the set of features $F(f)$ is estimated by the conditional probability $P(f \text{ is relevant}|F(c))$.

MePo: The MePo (*Me*ng–*P*auls*o*n) filter keeps track of a set of *relevant features* initially consisting of all the conjecture's features. It performs the following steps iteratively, until n facts have been selected: (i) Compute each fact's score, as roughly given by $r/(r+i)$, where r is the number of relevant features and i the number of irrelevant features occurring in the fact. (ii) Select all facts with perfect scores as well as some of the remaining top-scoring facts, and add all their features to the set of relevant features.

Each of the above algorithms orders the available (accessible) premises w.r.t. the likelihood that it is useful in proving the goal. In order to evaluate such orderings, we compute the first 1024 advised premises and compute five machine learning statistics for each of them: cover, precision, full recall, AUC, and the average rank. Below we provide the definitions of these concepts:

Definition 1 (100Cover)**.** *The average coverage of the set of proof dependencies by the first 100 suggestions is called the (100-)Cover; it is usually expressed as a percentage of the covered proof dependencies.*

Definition 2 (100Precision). *The ratio of the correctly predicted proof dependencies in the first 100 suggestions is called (100-)Precision.*

Definition 3 (Recall). *Full Recall (also called 100% Recall) is the minimum number of predictions needed to include the whole set of proof dependencies; or the number of predictions plus 1 if the considered predictions set is not large enough.*

Definition 4 (Rank). *(Average) Rank is the average position of a proof dependency in the sequence of suggestions ordered by their expected relevance.*

The AUC (Area under the ROC Curve) is the probability that, given a randomly drawn used premise and a randomly drawn unused premise, the used premise is ranked higher than the unused premise. Values closer to 1 show better performance.

Definition 5. *Let $x_1, .., x_n$ be the ranks of the used premises and $y_1, .., y_m$ be the ranks of the unused premises. Then, the AUC is defined as*

$$AUC = \frac{\sum_i^n \sum_j^m 1_{x_i > y_j}}{mn}$$

where $1_{x_i > y_j} = 1$ iff $x_i > y_j$ and zero otherwise.

5 Results

For the evaluation we first topologically sort all the available CoRN facts using their proof dependencies, and then for each of these facts we try to predict its proof dependencies by learning on all the previous (in the topological sorting) facts and their proofs. As usual in such large-theory evaluations, we thus emulate the standard situation of using AI/ATP assistance in interactive theorem proving: For each conjecture C we assume that all formulas stated earlier in the development can be used to prove C, and that all the proofs of previously stated theorems are available for learning.[2]

Table 2 shows the results. While the performance of k-NN and naive Bayes is quite comparable to similar early evaluations done for other systems and libraries, we get comparatively low performance of MePo, which does not correspond to its good behavior in our experiments with other systems. This could mean that the relations between symbols in the Coq propositions differ quite significantly from such relations in the other systems, and MePo-style heuristics need some modifications for Coq. The best (Ensemble) performance means that on average 75% of the dependencies needed in the actual Coq proof will be present among the first 100 hints recommended by the Ensemble predictor trained on the previous proofs.

6 Conclusion

As far as we know, this is the first large-scale evaluation of learning proof dependencies done on a larger library written in the Coq system. The overall results are quite encouraging, showing that guessing of the relevant parts of the library necessary for proving a new Coq conjecture is not significantly harder than in other ITP systems.

Future work includes better features for learning as already mentioned above, and obviously also research on good automated proof techniques for the Coq logic which could use the guessed

[2]The standard (randomized) 10-fold cross-validation would in general allow to prove a conjecture using a theorem that was only proved later, resulting in cyclic proofs.

-	100Cover (%)	100Precision	Recall	Auc	Rank
MePo	17.0	0.01004	954.45	0.2020	817.21
k-NN	72.3	0.07466	451.28	0.8341	173.45
NBayes	72.6	0.08089	448.80	0.8366	171.06
Ensemble	74.9	0.08093	428.70	0.8490	158.37

Table 2: Experimental Results

premises for finishing off smaller proofs automatically. These could be either translations into the formats and logics used by the currently strongest ATP systems, or direct implementations of custom proof-search methods working directly in the Coq logic.

7 Acknowledgments

Thanks to Herman Geuvers for his help with clarifying the Coq terminology. This research was supported by Austrian Science Fund (FWF):P26201, and by NWO grant *Knowledge-based Automated Reasoning.*

References

[1] Jesse Alama, Lionel Mamane, and Josef Urban. Dependencies in formal mathematics: Applications and extraction for Coq and Mizar. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *AISC/MKM/Calculemus*, volume 7362 of *LNCS*, pages 1–16. Springer, 2012.

[2] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *LNCS*, pages 88–103. Springer, 2004.

[3] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *LNCS*, pages 96–111. Springer Berlin Heidelberg, 2002.

[4] Sebastiaan Joosten, Cezary Kaliszyk, and Josef Urban. Initial experiments with TPTP-style automated theorem provers on ACL2 problems. In Freek Verbeek and Julien Schmaltz, editors, *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, volume 152 of *EPTCS*, pages 77–85, 2014.

[5] Cezary Kaliszyk and Josef Urban. Initial experiments with external provers and premise selection on HOL Light corpora. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *PAAR-2012*, volume 21 of *EPiC Series*, pages 72–81. EasyChair, 2013.

[6] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *CoRR*, abs/1310.2805, 2013.

[7] Cezary Kaliszyk and Josef Urban. Stronger automation for Flyspeck by feature weighting and strategy evolution. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013*, volume 14 of *EPiC Series*, pages 87–95. EasyChair, 2013.

[8] Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 2014. `http://dx.doi.org/10.1007/s11786-014-0182-0`.

[9] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[10] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013.

[11] Lionel Mamane and Herman Geuvers. A document-oriented Coq plugin for TeXmacs. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *MKM 2007 - Work in Progress*, volume 07-06 of *RISC Report*, pages 47–60. University of Linz, Austria, 2007.

[12] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[13] Robi Polikar. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45, 2006.

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

[15] Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 50:229–241, 2013.