Machine-Learned Premise Selection for Lean*

Bartosz Piotrowski[1], Ramon Fernández Mir[2], and Edward Ayers[3]

[1] IDEAS NCBR
[2] University of Edinburgh
[3] Carnegie Mellon University

One of the major challenges in formalizing mathematics in proof assistants is the prerequisite of having a good familiarity with the structure and contents of the library. In the case of Lean [?], users may look for relevant lemmas in its formal library, mathlib [?], either by using general textual search tools and keywords, or using mathlib's `suggest` or `library_search` tactics. The first approach is often slow and tedious. The limitation of the second approach is the fact that they propose premises that strictly match the goal at the current proof state.

This project aims to help Lean users find relevant lemmas when building a proof. We develop a new tool that efficiently computes a ranking of useful lemmas selected by a machine learning (ML) model. The ranking can be accessed interactively via the `suggest_premises` tactic. The tool is fully implemented in Lean 4 and uses mathlib3port[1] as the main data source.

This project is related to other ML-based premise selection works like [?, ?, ?, ?].

Data A crucial requirement of a useful ML model is a high-quality training dataset. In this work, we use simple ML architectures that cannot process raw theorem statements and require *featurization* as a preprocessing step. The features are similar to those used in [?, ?], in particular: `names` (all names of constants used in a statement), `bigrams` (all paths of length 2 from the syntactic tree of a statement), and `trigrams` (paths of length 3).

Traversing the generated proof term and keeping track of all the premises results in a large number of premises that a user would rarely write explicitly. Three different filters are applied to mitigate this issue. The `all` filter preserves almost all premises from the original, raw list, removing those that were generated automatically by Lean, e.g., `MatrixAlgebra._auxLemma.1`. The `source` filter leaves only those premises that appear in the proof's source code. Finally, the `math` filter preserves only lemmas that are clearly of mathematical nature. Lemmas from mathlib are used as a *white list*, filtering out lemmas from the Lean's core library (such as `rfl`).

ML models In this task, both the number of features and labels (premises) are large, and the training examples are sparse in the feature space. Thus, traditional implementations of ML algorithms were not well-suited, so we developed custom versions using Lean 4.

k-nearest neighbours (k-NN) is a classical ML algorithm [?], which has already been used multiple times for premise selection [?, ?, ?]. One disadvantage is the need to traverse the whole training data set in order to produce a single prediction (a ranking), which may be inefficient.

Random forest (RF) is an alternative approach that uses a collection of decision trees. In our case, the labels are sets of premises, and the rules are simple tests that check if a given feature appears in an example. When predicting, unlabeled examples are passed down the trees to the leaves, the reached labels are recorded, and the final prediction is averaged across the trees via voting. Our version of RF is adapted to deal with sparse binary features and a large number of labels. It is similar to the one used in [?]. It is trained in an *online* manner, i.e., updated sequentially with single training examples.

*The results were supported by the Hoskinson Center for Formal Mathematics, Kościuszko Fundation, and Principal's Career Development Scholarship of University of Edinburgh

[1] https://github.com/leanprover-community/mathlib3port (commit `f4e5dfe`)

Evaluation and results Our data is split into a training and a test set. The test set consists of the modules that are *not* dependencies of any other modules (592 of them), simulating a user of the tool writing a new module. The rest of the modules (2436) are used for training.

The quality of the ranking produced by the ML model is given by the Cover metric. Assuming a theorem T depends on the set of premises P of size n, and R is the ranking of premises predicted by the ML advisor for T, $\text{Cover}(T) = \frac{1}{n}|P \cap \{R[0], \ldots, R[n-1]\}|$. Cover_+ is defined in a similar way, taking a set of the first $n + 10$ premises from R instead. The rationale for Cover_+ is that the user, in practice, may look through 10 or more suggested premises.

Both k-NN and RF are evaluated on data subject to all three premise filters. Three combinations of features are tested: `names` only (`n`), `names` and `bigrams` (`n+b`), and `names`, `bigrams`, and `trigrams` (`n+b+t`). The hyper-parameters were selected by running experiments on a smaller data set. For k-NN, the number of neighbors is set to 100. For RF, the number of trees is set to 300, each example was used for training a particular decision tree with probability 0.3, and we passed through the whole training data 3 times. Table **??** shows the results.

RF performed better than k-NN for all data configurations, with the exception of Cover_+ for the `math` filter. The best feature configuration `n+b`, which likely means that `trigrams` caused *over-fitting*. The results for the `all` filter are higher because of theorems with trivial premises.

An additional practical advantage of this model over k-NN is the speed of outputting predictions. For instance, for the `source` filter and `n+b` features, the average times of predicting a ranking of premises per theorem were 0.28 s and 5.65 s for RF and k-NN, respectively.

	$\text{RF}_{\texttt{n}}$	$\text{RF}_{\texttt{n+b}}$	$\text{RF}_{\texttt{n+b+t}}$	$k\text{-NN}_{\texttt{n}}$	$k\text{-NN}_{\texttt{n+b}}$	$k\text{-NN}_{\texttt{n+b+t}}$
`all`	0.56 (0.67)	**0.57** (0.67)	0.47 (0.58)	0.51 (0.65)	0.52 (0.66)	0.51 (0.62)
`source`	0.28 (0.36)	**0.29** (0.36)	0.28 (0.36)	0.25 (0.35)	0.25 (0.36)	0.26 (0.35)
`math`	0.25 (0.32)	**0.26** (0.33)	0.16 (0.24)	0.22 (0.34)	0.23 (0.34)	0.16 (0.26)

Table 1: Average results of RF and k-NN on testing data with different features. Cover and Cover+ (in brackets) measures are reported. In each row, the best Cover result is bolded.

Interactive tool The ML predictor is wrapped in an interactive tactic `suggest_premises` that users can type into their proof script. The list of suggestions is displayed in the 'infoview,' a panel in Lean that displays goal states and other information about the prover's state. The display makes use of the new remote-procedure-call (RPC) feature in Lean 4 [**?**], to then asynchronously run various tactics for each suggestion. Given a suggested premise p, the system will attempt to run tactics `apply` p, `rw [`p`]`, `simp only [`p`]` and return the first successful tactic application that advances the state. This will then be displayed to the user who can select the resulting tactic to insert into the proof script. By using an asynchronous approach, we can display results rapidly without needing to wait for a slow tactic search to complete.

Future work The results may be improved by augmenting the dataset, as well as developing better features, utilizing the well-defined structure of Lean expressions. Although it would depart from our philosophy of a lightweight, self-contained tool, applying modern neural architectures in place of the simpler algorithms used here is a promising path [**?**]. Finally, premise selection is an important component of ITP *hammer* systems [**?**], and our tool can be readily used in a Lean hammer, which has not yet been developed.

Source code https://github.com/BartoszPiotrowski/lean-premise-selection