CONSTRUCTOR UNIVERSITY BREMEN

Master of Science
Computer Science

Andrei Kozyrev

# Similarity-driven Retrieval Mechanism for Rocq theorems

Master's Thesis

First reviewer:
Prof. Dr. Anton Podkopaev

Second reviewer:
Dr. Daniil Berezun

Bremen
2025

# Statutory Declaration

| | |
|---|---|
| Family Name, First Name | Kozyrev, Andrei |
| Matriculation number | 30006595 |
| Kind of thesis submitted | Master Thesis |

## English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

## German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

Bremen, May 20th, 2025 Andrei Kozyrev

# Abstract

Interactive theorem proving has repeatedly shown to be fruitful in various domains with critical infrastructure; however, writing formal proofs remains a challenging task, requiring complex reasoning. In this work, we utilize Large Language models to solve the problem of generating formal proofs and explore currently existing approaches for Retrieval Augmented Generation in this domain. We propose a novel approach for extracting relevant premises from a large corpus of Rocq proofs, leveraging retrieval via a self-attentive embedder model. We evaluate the performance of our proposed solution and achieve a relative improvement of up to 28% over the baseline. This work also contains a detailed analysis of the results and the limitations of our approach.

# Acknowledgements

I would like to express my deepest gratitude to my Academic supervisor and colleague, Anton Podkopaev, for his invaluable guidance and support throughout the course of this thesis. I thank Daniil Berezun for reviewing this thesis and providing insightful feedback. I would also like to sincerely thank my supportive friends and colleagues, who have been there for me during this journey. Their encouragement and assistance with specific parts of this research are not to be overlooked. I am grateful to my family for their unwavering support and encouragement, which has been a constant source of motivation for me.

# Contents

# Introduction

Testing software has always been an essential part of Software Engineering. Creating reliable software is a challenging task, but it is a must in many domains with no space for mistakes. For some specific domains, such as Aviation, Medical Software, Banking, and others, reliable software is a matter of people's lives. There were cases when software bugs in such domains led to catastrophic consequences [7, 22].

Formal software verification ensures the correctness of software under some given specifications. When this specification is precise, complete, and unambiguous, formal verification gives developers stronger guarantees than, for example, unit/integration testing. To date, there exists a number of Software Verification systems or Interactive Theorem Provers (ITPs), such as Rocq (former Coq) [3], Agda [17], Lean [6], Isabelle [24], and others. Such a system helps users specify properties of their programs and then prove that these properties hold using formal logic.

Rocq is a mature ITP that has existed for more than 30 years. Rocq has proven its usability through a number of projects, both in Academia and Industry. For instance, CompCert [20] is a formally verified C compiler written in Rocq. CompCert was the only C compiler where a sufficient study found no bugs. Rocq is also used to develop CertiKOS [14], a formally verified operating system kernel. In Academia, Rocq is used in several projects, such as the first proof of the Four Color Theorem, accepted by the mathematical community [13], a recently published new work on a Relaxed Memory Model [23], and many more [15, 26].

A typical pipeline of writing proofs in Rocq is to state the theorem and then prove it step-by-step. At each step, Rocq's system allows the user to observe the current state of the proof. It contains a goal we want to show is correct and a list of hypotheses under which the goal is to be proven. At each step, the user applies so-called *tactics* that transform the current goal. Tactics are elementary building blocks for the proof, that can simplify the goal, apply some domain-specific reasoning, destruct the goal into smaller

subgoals, and so on.

Software verification has proven to be fruitful, but it is a time-consuming process. Writing proofs in Coq is a challenging task and requires considerable experience from the programmer [28]. This explains the high demand for tools that help automate the process of writing proofs.

Some generative tools for Rocq and other ITPs utilize classical reasoning methods, some — machine learning techniques. CoqHammer [5] translates Rocq's logic into first-order logic and performs the proof search. Tactician [4] contributes a tactic for different proof search strategies and implements the $K$-Nearest Neighbours ($K$-NN) algorithm for proof search. Recently, studies [18, 10, 33, 29] have shown that machine learning could also be successfully applied to the problem of proof synthesis and results in a fascinating symbiosis with Formal Methods.

CoqPilot [18] — one of the remarkable works in the domain, is a VSCode plugin allowing users to generate Rocq proofs seamlessly. CoqPilot utilizes the power of Rocq's system and its ability to automatically validate generated code and combines it with the generational capabilities of modern LLMs. CoqPilot was designed to serve as a framework for combining multiple proof generation methods and seamlessly incorporating them into a single user-friendly pipeline for proof synthesis.

CoqPilot's generation engine is a powerful combination of valuable tools that can be used in a single pipeline. In addition, CoqPilot serves as a platform for experiments and allows for easy benchmarking of different approaches. Therefore, CoqPilot is a great choice when one wants to implement and test new proof generation methods.

Many approaches call attention to the use of premise selection, *i.e.,* retrieving relevant information from the context to advance generation. LeanDojo [34] is a retrieval-based generation approach for Lean, which generates proofs for theorems step-by-step. On each step, it retrieves relevant premises from the large corpus of Lean proofs, which is prepared

in advance. LeanDojo implements a beam-search algorithm on top of the tactic generator to navigate the proof space, which is a common approach in the field of proof synthesis. LeanDojo has shown that the retrieval-based generation can outperform strong baselines, highlighting the importance of premise selection. Another vital research in the context of this work is Rango [30]. Rango's authors have measured how different retrieval methods of premise selection affect the generational capabilities of the model. *hint selection* is formulated as follows: given a proof state $S$, we suppose that to proceed with the proof, we want to apply some tactic with an unknown positional argument.

$$\texttt{apply}\ \tau_1. \quad \text{or} \quad \texttt{destruct}\ \tau_2.$$

The problem of *hint selection* is to yield such potential premises $\tau$ that might be used in the tactic. Potential premises might be auxiliary lemmas, definitions, objects, etc. The result is afterwards fed into the tactic-generation model/mechanism.

Most works on premise selection in ITPs focus on lemma selection. However, authors of Rango show that on their CoqStocq dataset (introduced in the same paper), hint selection provides minimal improvement over the baseline. On the contrary, works [18, 30] present the mechanism called *proof selection*. Problem of *proof selection* is formulated as follows; given a target theorem statement $S$, a generator model $m$, and a large dataset of already proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$, we want to choose a subset of those theorems $\{[s_i, p_i]\}$, so that we maximize the possibility, that model $m$ will correctly solve statement $S$, considering $\{[s_i, p_i]\}$ as the context. Evaluation in both CoqPilot and Rango works shows that *proof selection* brings invaluable profit to the generator. However, both approaches limit their proof selection mechanisms to a simple baseline, that orders the candidate theorems by decreasing lexical similarity of statements.

The main topic of discussion in this work is, "How do we choose the relevant theorems from the reference set to maximize the probability of proving a theorem?"

This work builds on the existing research [18, 30], and introduces a novel approach to *proof selection* in Rocq. We aim to train a self-attentive embedder model to represent theorems in a vector space. The model is supposed to be trained on a large dataset of Rocq statements with respective proofs in such a way that the cosine similarity between the vector representations in the latent space represents the similarity of the statements' proofs.

During the process of collecting the dataset for our model, we encounter a data scarcity problem in Rocq. There were attempts to create datasets for Rocq, such as CoqGym [33], but the main problem is that Rocq is not backward compatible. This means the proofs written in one version of Rocq may not be valid in another. Rocq is extremely hard to build in different versions, making the datasets hard to maintain and use. Alongside, there is not that much Rocq data available online in general. In absolute numbers, TheStack [16] dataset contains roughly 100 million tokens of Rocq code, compared to more than 10 million files of Python code. One Rocq file could be estimated to contain around 300 lines of code, each containing around 10 tokens.

In this work, we additionally aim to tackle Rocq's data-scarcity problem by introducing a data-augmentation tool that will run on a project and generate additional *dataset samples*, each comprising a theorem and its proof for every existing theorem. The tool is to convert proofs into tree-structured representations and interact with the Rocq system via the Language Server Protocol (LSP) [1] to synthesise new proofs.

## 1.1 Objectives

**O1 Data-augmentation tool**. Develop a CLI utility that

1. extracts every proof as a tree,
2. generates viewers for the proof trees,
3. generates the dataset in a format, compatible with model training

pipeline,

4. generates augmented `.v` files with freshly generated lemmas.

**O2 Augmented Rocq corpus**. Run BigRocq over public projects, gather the original + synthetic theorem–proof pairs, and collect an extensive dataset for training the model.

**O3 Self-attentive premise ranker**. Train a Transformer encoder whose cosine similarity correlates with proof-level similarity.

**O4 Evaluation**. Evaluate the model on the CoqPilot benchmark and show statistically significant improvements over the baseline.

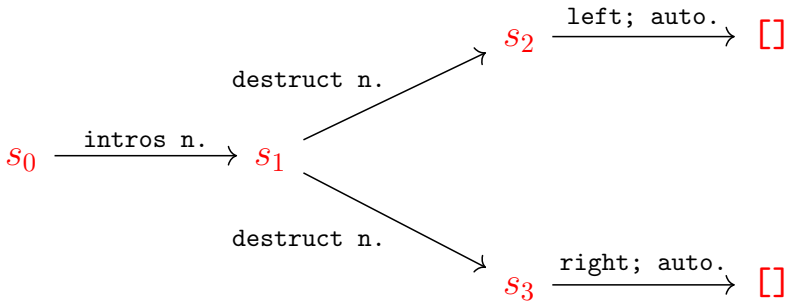
## 1.2 Proposed approach

This work aims to create a Rocq data augmentation tool. We propose an approach to generate new dataset samples for every theorem in a Rocq project. Given a theorem $T$ and its corresponding proof $P$, we can extract a sequence of tactics $\tau_1, \tau_2, \ldots, \tau_n$, used in the proof. In a simplified setting, each tactic is applied to some proof state $S_i$ and transforms it into several new states (possibly zero, meaning this proof branch is closed). We aim to transform the linear structure of the proof into a tree-like structure, with nodes representing proof states and edges representing tactics. In such a tree, each subtree would represent a valid and independent new lemma, and a depth-first search (dfs) traversal of the subtree would allow us to collect a sequence of tactics that would lead to the proof of the lemma.

As an example of such conversion, consider the following simple theorem, stating that any number is either equal to zero or not equal to zero:

```
Theorem test2nat1 : forall n : nat, n = 0 \/ n <> 0.
Proof.
 intros n.
 destruct n.
 - left; auto.
 - right; auto.
Qed.
```

The proposed solution will transform this linear structure into the following tree:




For each theorem, such a procedure shall produce up to $n$ new dataset samples, where $n$ is approximately equal to the number of tactics used in the proof.

To effectively gather information about intermediate proof states, traverse, and parse proofs, in this work, we utilize Rocq's Language Server Protocol (`Coq-LSP`) [1]. Language Server Protocol [1] is a protocol developed by Microsoft that allows for a unified way of communication between a language server and a client. A client is typically an Integrated Development Environment (IDE). A Language Server operates on a given project, stores states of open files, and provides features such as code diagnostics, code completion, go-to-definition, and others. In order to efficiently collect information about the Rocq project and its theorems, this work aims to implement a higher-level module, wrapping `Coq-LSP` and providing required abstractions for the data augmentation tool.

Using the developed tool, we mine a dataset of theorems with respective proofs from existing open-source Rocq projects. We train an embedder model on the produced dataset in the following way: we encourage the model to learn the dependency between the similarity of the theorems and the similarity of their proofs. We define proofs' similarity as the modification of Levenshtein edit distance (discussed in § 4), and train the model to approximate this function, but using statements, instead of their proofs. Given such a function, we can predict whether two statements have similar

[1]Language Server Protocol: https://microsoft.github.io/language-server-protocol/

proofs or not. In case of a positive prediction, that would mean that one is a relevant premise for the other.

# 2 Related Work

In this section, we will discuss existing tools and approaches to proof generation, including CoqPilot, describe existing Rocq datasets, briefly introduce the reader to the Rocq proof assistant, and explain the basics of Machine Learning (ML), Natural Language Processing (NLP), and Retrieval Augmented Generation (RAG).

## 2.1 Rocq datasets

The largest attempt to create and use a Rocq dataset for machine learning is CoqGym [33]. CoqGym includes 71k human-written proofs from 123 Rocq projects. CoqGym possesses several problems. Firstly, it relies on the deprecated SerAPI[2] library. SerAPI is a library for machine-to-machine interaction with the Rocq proof assistant, with particular emphasis on applications in IDEs. To use CoqGym, one could download the preprocessed dataset (JSON files, no possibility to interact with Rocq files/Rocq's System, check new proofs in context, etc.) or use the scripts provided by the authors to extract the dataset. This comes with a number of problems. Firstly, SerAPI has been deprecated for a while, and the release 8.20 (dated July 2024) was officially the last release managed by the SerAPI team. Secondly, the scripts for dataset extraction are not generalized enough and are poorly maintained.

CoqGym was not the only attempt to collect a Rocq dataset. Work [11] by Andreas Florath describes a dataset collected implicitly to enhance LLMs' proficiency in interpreting and generating Rocq code. The dataset consists of around 10K Rocq source files. Charles Norton has also recently addressed an issue of Rocq datasets and proposed `Coq-MetaCoq`[3] and `Coq-MetaCoq-QA`[4] datasets. Nevertheless, the main issue persists. Coq is highly version-dependent and not backward-compatible. This makes it hard to maintain a dataset. We aim to close this gap by providing an easy-to-use

[2]SerAPI https://github.com/rocq-archive/coq-serapi
[3]Coq-MetaCoq https://huggingface.co/datasets/phanerozoic/Coq-MetaCoq
[4]Coq-MetaCoq-QA https://huggingface.co/datasets/phanerozoic/Coq-MetaCoq-QA

and adaptable utility for Coq dataset augmentation.

## 2.2 Rocq's system

At the moment, one of the widely used Integrated Development Environments (IDEs) for Rocq is `VSCode`. `VSCode` implements support for programming languages through implementing a Language Server Protocol (LSP). There are two commonly used LSPs for Rocq: `vscoq` [8] and `Coq-LSP` [1]. The latter one has been developed earlier and is used in our work. From now on, we will refer to it as `Coq-LSP`.

To prove a theorem in Rocq, the user shall start by constructing a specification, or theorem statement. We will proceed with a trivial example, stating that for any natural number $n$, it is either equal to 0 or not equal to 0, which we will denote as `test2nat1`. The user can start by typing the following line in the editor:

```
Theorem test2nat1 : forall n : nat, n = 0 \/ n <> 0.
Proof.
```

Assuming that the `Coq-LSP` VSCode extension is installed, placing the cursor at the end of the line, and navigating to the right panel of the editor, users will see the following:

```
Goal (1)
============================
forall n : nat, n = 0 \/ n <> 0
```

This is the goal that is to be proven. Typically, hypotheses are located above the line, and the conclusion of the goal is below. If we apply the `intros.` tactic, that automatically introduces all hypotheses from the left side of the conclusion to the context, the goal will be updated to:

```
Goal (1)
n: nat
============================
n = 0 \/ n <> 0
```

As with most proofs about natural numbers, we either proceed with induction or with case analysis. In this case, as the theorem is trivial, we can use the

`destruct`, and cover two cases: (i) $n = 0$ and (ii) $n \neq 0$, this is reflected in the following code:

```
destruct n. (* Two goals *)
- left; auto. (* No more goals. Focus next goal with bullet -. *)
- right; auto. (* No more goals. *)
```

The proof is complete when the conclusion is empty, as at the end of this code. The proof environment could be exited by writing `Qed.` or `Defined.` in the case of definitions.

By default, tactic expressions are applied only to the first goal. However, Rocq provides a special syntax called *goal selectors*[5], that allows to apply tactics to any goal, goal sequence, goal range, or even to all goals. We could rewrite the proof from our example as follows:

```
Theorem test2nat1 : forall n : nat, n = 0 \/ n <> 0.
Proof.
 intros n.
 destruct n as [|n'].
 all: try (left; auto) || (right; auto).
Qed.
```

The prefix `all:` indicates that the tactic should be applied to all goals.

## 2.3 Retrieval Augmented Generation

Premise selection has been a long-standing challenge in the field of Automated Theorem Proving. It was sufficiently covered in literature [19, 2, 31]; however, *proof selection*, as described in the Introduction, was barely researched. The main reason for this is that applying Deep Learning transformer models to ITPs is a relatively new field. Before the existence of transformers, it was unclear how we could utilize similar proofs to generate the target one. CoqPilot [18] and Rango [30] works have successfully implemented a baseline proof selection into their generation pipeline. This baseline works as follows. Given a target theorem statement $S$, a generator model $m$, and a large dataset of already proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$, assign a score to each potential premise $[s_i, p_i]$, according to their lexical state

[5] https://coq.inria.fr/doc/V8.18.0/refman/proof-engine/ltac.html#goal-selectors

similarity similarity$(S, s_i)$, and then select the top $k$ most similar premises. As shown in § 4, this approach is not optimal, as similar statements do not necessarily lead to similar proofs. Authors of Rango use a similarity-based metric, BM-25, while CoqPilot uses the Jaccard-similarity index to tackle the same problem. The two metrics are semantically and statistically the same. In § 5, we will show how our approach outperforms the Jaccard-similarity baseline by 28% in terms of the number of generated proofs, on a dataset of 300 theorems.

## 2.4 Evaluation dataset

For hypothesis testing and evaluation of our proposed approach we decided to reuse the dataset from CoqPilot. It is limited to 300 theorems from the IMM project [27], such limilation is suitable for us in terms of computational and financial costs. Theorems are divided into three buckets by difficulty, where the number of tactics in their human-written reference proofs serves as a proxy for difficulty. Dataset is limited to proofs containing no more than 20 tactics, reflecting CoqPilot's original focus on subgoals and shorter lemmas. Proofs of this length account for 83% of all proofs in the IMM project. Bucket boundaries and sizes were chosen based on the original distribution of proof lengths in the project; the resulting length ranges are $[1, 4]$, $[5, 8]$, and $[9, 20]$ tactics. Resulting group sizes are $[131, 98, 71]$ theorems. We call this collection of 300 theorems the *IMM-300* dataset. None of the theorems appear in the embedder's training data (which only contained partial goals, not full statements).

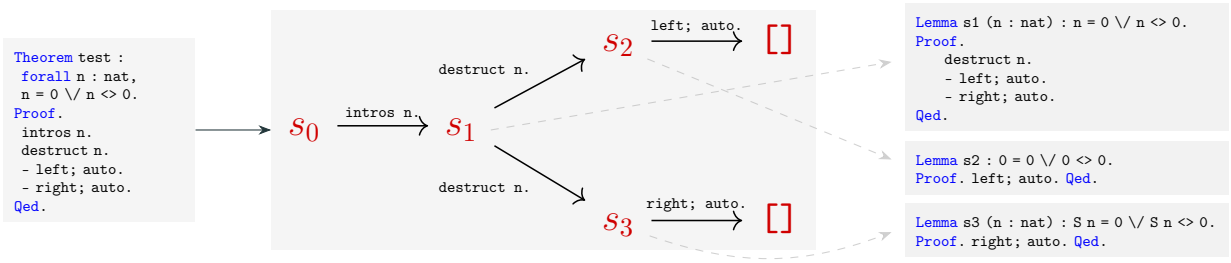



Figure 1: Processing theorems into trees. $s_i$ denotes a state

# 3 BigRocq

If we recall the example from § 2.2, we could notice that even though the proof is linear, it rather has a tree rather than a sequential structure. On an example from § 2.2:

```
Theorem test :
 forall n : nat,
 n = 0 \/ n <> 0.
Proof.
 intros n.
 destruct n.
 - left; auto.
 - right; auto.
Qed.
```

When we applied the `destruct` tactic, we created two branches of the proof. Proof state at that moment is as follows:

```
Goal (1)

============================
0 = 0 \/ 0 <> 0

Goal (2)
n nat
============================
S n = 0 \/ S n <> 0
```

Due to the nature of Rocq proofs, we could transform most of them (limitations discussed in § 3.1) into trees, where nodes will be states and edges — transitions between them. If we fix a node and traverse its subtree, we will obtain a proof for the node. From a single theorem with a proof of length $l$, we could extract $l$ new auxiliary lemmas with respective proofs. This procedure is depicted in Figure 1. We call the proposed tool *BigRocq*

and make it publicly available as a standalone component of our system. It is located in the mono-repo of the project at https://anonymous.4open.science/r/E60E/big-rocq/README.md. BigRocq takes a Rocq project as input, iterates over proven theorems in it, and generates a set of auxiliary lemmas with respective proofs. It produces new Rocq files containing the freshly generated lemmas listed above, source theorems so that the user can quickly open the file inside the IDE and ensure that the generated lemmas correctly type-check. Additionally, BigRocq generates visualizations for all of the built trees, so that one can explore them in a more user-friendly way. Packed with statistics, such as elapsed time, success rate for tree building and success rate for verifying proofs, everything is stored to the desired location.

## 3.1 Implementation

BigRocq is implemented in `TypeScript` due to two reasons: (i) it is a natural choice for a language client, it possesses a rich set of libraries for automating client-server communication routine, and (ii) we wanted to re-use some parts of the CoqPilot project, which is implemented in `TypeScript`.

We use `Coq-LSP` as the backbone of our system. At the second level of abstraction, we wrap it into an interface, allowing us to conveniently communicate with Rocq's system. One of the difficulties we had to overcome is that `Coq-LSP`, as most LSPs, is designed to communicate with IDEs, not users nor CLIs; therefore, the set of provided API calls is not adapted for our specific use. We implement a set of heuristics on top of the server to adapt it to our needs. The resulting object contributes the following methods:

```
getGoalsAtPoint(
 position: Position,
 documentUri: Uri,
 version: number,
 command?: string
): Promise<Result<GoalConfig<PpString>, Error»;
```

```
getRocqDocument(uri: Uri): Promise<FlecheDocument>;

withTextDocument<T>(
 documentSpec: DocumentSpec,
 block: (
 openedDocDiagnostic: DiagnosticMessage
 ) => Promise<T>
): Promise<T>;
```

The first method allows us to obtain the goal at the cursor position. The second one is used to request the internal representation of the document, stored inside `Coq-LSP`, and is then used to parse it. The third method is a wrapper around the `Coq-LSP` API that allows working with a document under the decorator. It handles opening and closing the document, correctly working with its state, and waiting for the server to do type-checking.

BigRocq implements a simple cycle over files inside the project, automatically typechecks these files, and parses them into an internal representation. We attempt to construct a *proof tree* for each of the parsed theorems. We start from a tree with a single node — the initial proof state of the theorem. We iterate over tactics used in the proof, and with each new tactic, we extract the proof state before and after its application and add new nodes to the tree. It is done via the following naive algorithm:

```
(* Destruct the number of goals before and
after tactic application: (NUM_B, NUM_A) *)
match numbers_of_goals with
| case(NUM_B = NUM_A): create edge B_0 -> A_0; break
| case (NUM_B - 1 = NUM_A): create edge B_0 -> []; break
| case (NUM_B + k = NUM_A): (* Tactic creates new goals *)
 create edges for i in 0..k B_0 -> A_k
```

The nature of this algorithm is explained by the fact that we do not get much information from the `Coq-LSP`. We can only retrieve the list of goals at the given point. We do not possess goal IDs, etc.

Unfortunately, the described algorithm has limitations. The issue is that

when goal selectors (described in § 2.2) occur, we can no longer determine how to proceed with building our proof tree. We have a snapshot of the list of currently unsolved goals before and after the application of the tactic, but, in the current implementation of `Coq-LSP`, we are unable to figure out which goals have transformed into which; therefore, we are unable to build the tree. This problem was also described in CoqGym [33] work. This work possesses the same limitation. Fortunately, the portion of theorems that use goal selectors is small, and we can ignore them. Along with goal selectors, we cannot process theorems with existential quantifiers, as they break the hypothesis that two different goals are independent from one another.

For each theorem in the project, we either successfully produce a proof tree or reject the theorem with a reason, such as `goal selector` or `proof is unfinished`. Afterward, we will proceed with validating all newly generated theorems. By default, we assume that all of them should be valid, but, for example, existential quantifiers are only detected during these checks. This process is packed with heuristics and unintuitive procedures due to the nature of `Coq-LSP`. A pseudo-algorithm, illustrating the validation process, is presented in Algorithm 1. Then we proceed with generating visualizations for the trees in a collection of static HTML files. Packed with the serialized dataset, Rocq `.v` files, and run statistics, everything is stored in a requested location.

## 3.2 Usage

On an example of the IMM project, we will demonstrate the use of BigRocq. BigRocq provides a simple yet flexible `CLI` that lets you point it at your Coq sources, specify where to write out the augmented proofs, and select the `coq-lsp` server binary you wish to drive. After installing all dependencies, the minimal invocation looks like this:

```
npm run augment - -targetRootPath <path>
    -workspaceRootPath <path>
    -coqLspServerPath <path>
```

**Algorithm 1** Check Generated lemma for the theorem

```
1:  procedure CheckLemmas(Theorem)
2:      GlobalVars ← DetermineGlobals(Theorem)
3:      CreateFile ()
4:      InsertPrefix (Theorem, File)
5:      Results ← [ ]
6:      for all ℓ in GeneratedLemmas do
7:          U ← ℓ.Hyps \ GlobalVars
8:          ℓ′ ← SetHyps(ℓ, U)
9:          InsertAfterPrefix (ℓ′, File)
10:         Typecheck (File)
11:         Diags ← GetDiagnostics()
12:         Results.Append(⟨ℓ′, Diags⟩)
13:         RemoveLemma (ℓ′, File)
14:     end for
15:     return Results
16: end procedure
```

Under the hood, BigRocq will recursively scan your workspace, spin up a dedicated `coq-lsp` process, and automatically type-check and instrument every proof. All of this happens without further manual intervention, so you can leave the tool to run over hundreds of definitions and theorems unattended. Figure 2 shows an example of the produced statistics: the ratio of successfully augmented nodes, the ratio of successfully build proof trees, the avarage length of among newly generated proofs, and how did the

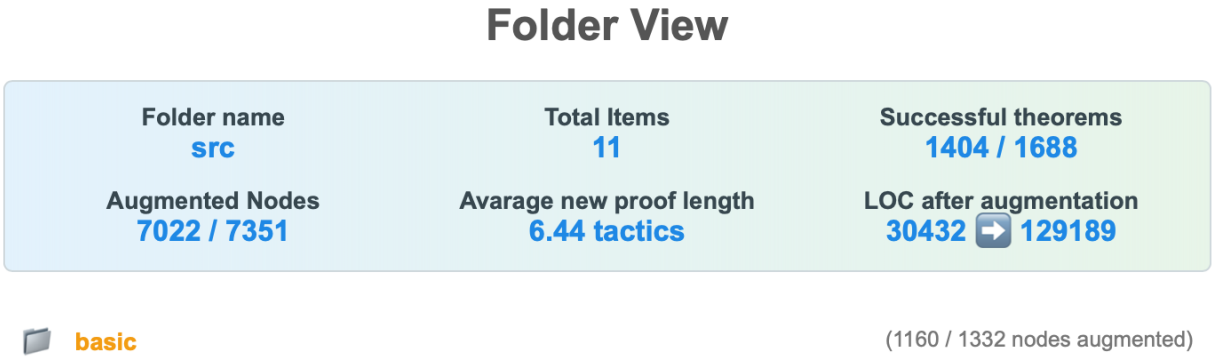



Figure 2: IMM BigRocq statistics

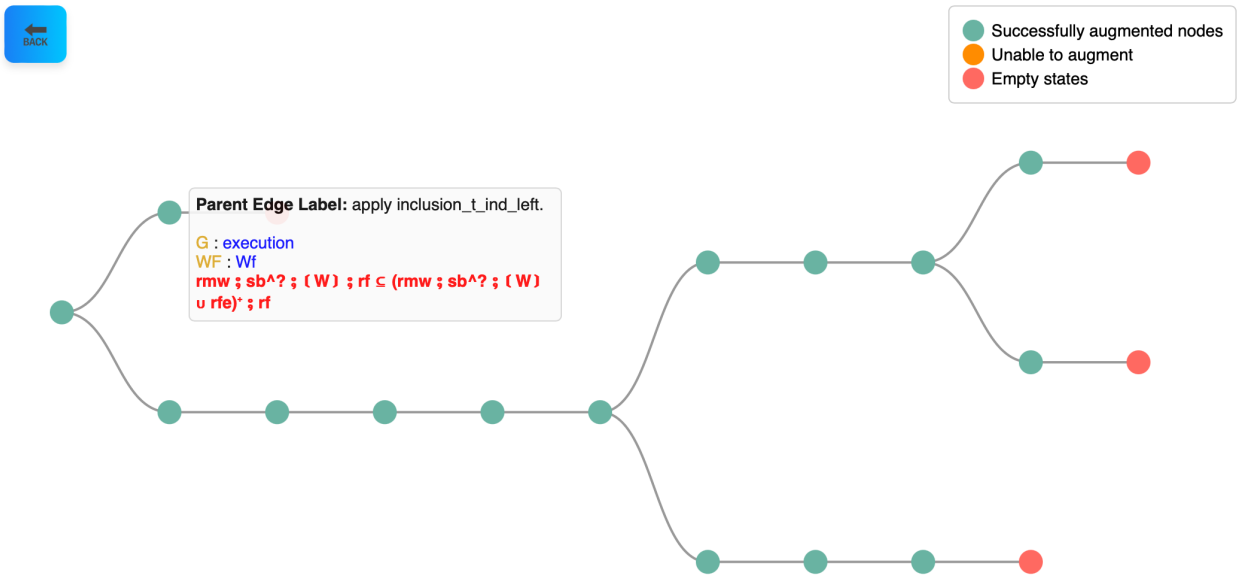



Figure 3: IMM BigRocq tree

number of lines of code change after augmentation.

With such output, one can quickly spot which parts of the development yield the richest augmentation opportunities, compare proof-length trends across modules, or even export the data for downstream machine-learning experiments. An example of the proof tree visualization is presented in Figure 3. The tree is interactive, and allows to expanding and collapsing nodes.

# 4 Embeddings

This section will cover the procedure of building the embeddings for the Rocq statements. We start with an extensive motivation for the task, then describe the process of mining the dataset of Rocq statements, and afterwards discuss the training process of a self-attention model to learn the embeddings of the Rocq statements. We call our proposed embedding model *RocqEta*. In § 5, we assess the effectiveness of the proposed approach.

## 4.1 Work justification

As already introduced in the Introduction, the primary goal of this work is to improve methods of automated *proof selection*. Proof selection is formulated as follows. Given theorems $\{T_i\}$ and statement $s^*$, it means choosing other statements with their respective proofs, so that their presence in the context of the generation request would help the model with the generation of the proof for statement $s^*$. *Assumption 1* is that the model benefits from seeing similar proofs to the one it is trying to generate. Proof selection methods in existing literature [18, 30] are based on the assumption that if statements $s_*$ and $s_i$ are similar, their respective proofs $p_*$ and $p_i$ are similar as well, we will call it *Assumption 2*:

$$\text{similarity}(s_*, s_i) \implies \text{similarity}(p_*, p_i) \tag{2}$$

Therefore, assuming that the model will benefit from seeing proofs of similar statements. However, we show that this assumption often **does not** hold. A trivial illustration of it is that a single statement can have multiple very diverse proofs.

To further illustrate the problem, we conduct an experiment where we calculate how statement similarity correlates with proof similarity on large-scale data. At first, we define statement similarity as a commonly used BM-25 similarity metric, and proof similarity as a modified version of the

Levenshtein edit distance:

$$D_L(p_i, p_j) = \frac{\text{Lev}(p_i, p_j)}{\max(l_i, l_j)}, \quad D_J(p_i, p_j) = 1 - \frac{|p_i \cap p_j|}{|p_i \cup p_j|},$$
$$\text{proof\_distance}(p_i, p_j) = \alpha\, D_L(p_i, p_j) + (1 - \alpha)\, D_J(p_i, p_j) + \gamma$$

where $p_i$ is a viewed as a list of tactic $p_i = [tac_{i_0}, \ldots, tac_{i_m}]$, $l_i$ is the length of the proof. $D_J$ is a Jaccard distance; it is added for robustness. $\alpha$ is a hyperparameter that controls the balance between the two metrics, and $\gamma$ is a small random noise added to the distance to avoid overfitting. Alternatively, we also measure the correlation with the BM-25 metric used to measure proof similarity.

Considering all 1927 theorems from the IMM project we get 1,855,701 pairs of theorem. We calculate the correlation between the statement similarity and proof similarity. BM25-based statement similarity is weakly and negatively associated with Levenshtein-based proof distance (Pearson $r = -0.154$, Spearman $\rho = -0.171$). BM25-based proof similarity yields an almost zero Pearson correlation ($r = 0.029$) and only a small positive Spearman correlation ($\rho = 0.240$), both effectively negligible. Pearson correlation quantifies the strength of a linear relationship between two numerical variables, whereas Spearman correlation evaluates the strength of a monotonic relationship based on the ranked values of the variables [12]. Results highlight that Assuption 2 does not generally hold.

As the main objective of this work, we aim to find such a function $f(s_i, s_j)$ that would correlate with similarity$(p_i, p_j)$ strongly than sim$(s_i, s_j)$. First of all, we conduct an experiment that determines whether such a function could theoretically help us improve proof selection. We do hypothesis testing for Assumption 0. We take the *IMM-300* dataset and as we already have proofs for all of them, we could perform so-called *oracle* premise selection. This means that we will iterate through the theorems in the dataset, and for each target theorem, we will take those theorems as premises that have the most similar proofs to the target one. This method is called *oracle*, because at test-time we don't have access to the reference proof of our target theorem.

| Ranker | Jaccard | Oracle |
|---|---|---|
| GPT-4o | $29\% \pm 3\%$ | $\mathbf{34\%} \pm 3\%$ |
| Claude 3.5 | $38\% \pm 3\%$ | $\mathbf{40\%} \pm 3\%$ |

Table 1: Oracle premise selection experiment.

We conducted the experiment to compare two methods of proof selection: (i) Jaccard-similarity index over the statements, and (ii) Oracle premise selection. The experiment was repeated 3 times to provide a 0.95 confidence interval. The results are shown in Table 1. We have done generation with two different models under the hood. Table 1 shows that the oracle premise selection method outperforms the Jaccard-based method with both models. This sets a valid ground for the next step of our work.

## 4.2 Modeling

We propose to use a self-attention encoder model [21] to learn the embeddings of the statements. We assume that such a model could learn to determine whether two statements have similar proofs or not.

To collect the dataset of Rocq statements, we take four large open-sourced Rocq projects and augment them. These projects are: CompCert[6], IMM[7], PromisingToIMM[8], and XMM[9]. CompCert is one of the largest open-sorced Rocq projects, providing a good amount of data, other three projects are of particular interest to our research lab. These four projects have a total of 10,314 theorems. After augmenting them with BigRocq, we get a dataset of 76,524 theorems, more than 7 times the original dataset.

As already mentioned, Assumption 2 does not hold in general; however, it holds in some cases, creating a strong baseline for our model. Due to that and the restrictions of computational resources, we decided not to do full training on the model but rather to fine-tune an existing encoder, which is

[6]CompCert: https://github.com/AbsInt/CompCert
[7]IMM: https://github.com/weakmemory/imm
[8]PromisingToIMM: https://github.com/weakmemory/promising2ToImm
[9]XMM: https://github.com/weakmemory/xmm

already trained on a large corpus of code. As such, we decide to take the `codebert` [9] model.

CodeBERT is a Transformer-based model that learns to understand both programming code and natural language together. Built on the same architecture as RoBERTa-base (about 125 million parameters), it is trained by masking out tokens in paired code and documentation, then teaching the model to predict them. It also learns by spotting replaced tokens in extensive, unlabeled code and text collections. This mixed training on millions of code–text pairs and standalone code examples helps CodeBERT capture the meaning of code and its comments, making it a strong starting point for tasks like searching for code snippets or generating documentation.

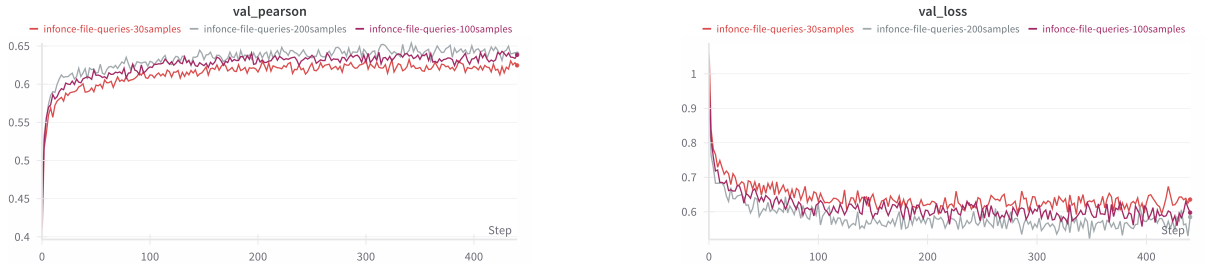
On a tiny extra test dataset, consisting of 50 theorems with corresponding hand-picked premises, raw `CodeBert` achieves an accuracy of 48%. This corresponds to roughly the same accuracy for ranking performed using the Jaccard-similarity metric on statements.

Given a dataset of pairs $\{(\mathit{statement}_i,\ \mathit{proof}_i)\}$ and a proof-similarity function $f(\mathit{proof}_i, \mathit{proof}_j)$, we aim to learn a ranker

$$r : \mathit{statement} \times \mathit{statement} \ \to\ \mathbb{R}$$

That, given two Coq statements, predicts their proof similarity as closely as possible to $f$. In
Crefsec:evaluation, we evaluate the proposed model on the following task: for a target statement $s_*$ and a set of already proven theorems, select the top $k$ premises and use them as context to generate a proof for $s_*$.

$$\begin{aligned}
\mathcal{T} = \{(p_i, s_i)\}\,,\ \mathcal{S} &= \{s_i\} \\
\mathrm{Top}_k(r, s_*) &= \operatorname*{arg\,top}_{k\ (p_i,s_i)\in\mathcal{T}} g(s_i, s_*) \\
\mathrm{Solve}(r, s_*) &= \mathrm{Solve}\big(\mathrm{Top}_k(r, s_*), s_*\big) \in \{0, 1\} \\
\mathcal{Q}(r) &= \mathbb{E}_{s_*\sim\mathcal{D}}\big[\mathrm{Solve}(r, s_*)\big]
\end{aligned}$$




(a) Correlation

(b) Validation loss

Figure 4: Train metrics for different values of $k_{\text{samples}}$.

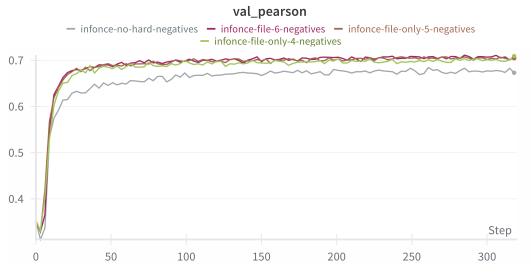
We optimize the model with the InfoNCE [25] loss. Concretely, for each statement $s$, we compute proof-to-proof distances against other examples during post-processing. One relevant question is, to which other examples should we calculate distances? We could use all examples, resulting in a quadratic number of pairs. However, it is unclear whether it is optimal. Let us denote this hyperparameter as $k_{\text{samples}}$. We compare different values of $k_{\text{samples}}$ in Fig. 4. Considering the depicted results, we consider that $k_{\text{samples}} = 200$ to produce the best results. Moreover, we decided to always compute paired distances between statements from the same file, as they are the most relevant ones. As 50 is a rough estimation of the number of theorems in one file, we choose the default value of $k_{\text{samples}} = 150$, plus always compute distances to the statements from the same file.

We label a pair as positive if its distance falls below a threshold $\tau_{\text{pos}}$, and negative if it exceeds $\tau_{\text{neg}}$. Let $P_s^+$ and $P_s^-$ be the resulting sets of positive and negative pairs, and $k_{\text{neg}}$ the number of hard negatives (will be explained in the next paragraph); then the per-statement loss term $\mathcal{L}_s$ is given by:

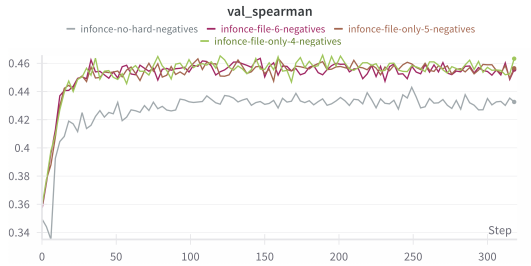
$$\mathcal{L}_s = -\log \frac{\exp\big(\varphi(z_s, z_p)/T\big)}{\exp\big(\varphi(z_s, z_p)/T\big) + \sum_{j=1}^{k_{\text{neg}}} \exp\big(\varphi(z_s, z_{n_j})/T\big)} \qquad \big(p \in P_s^+,\ n_j \in P_s^-\big)$$

where $\varphi$ is a cosine similarity between $\ell_2$-normalized embeddings of statements.

Because of the shape of the proof-distance distribution, training tended




(a) Pearson Correlation




(b) Spearman Correlation

Figure 5: Correlation metrics for different $k_{\text{neg}}$ and w/o hard negatives.

to focus too quickly on *easy* negatives — pairs whose proofs (and typically their statements) are already far apart. To preserve informative gradients, we inject hard negatives: with some probability, we label $(s_a, s_b)$ as negative whenever

$$\tau_{\text{hardneg}} \;\leq\; \text{sim}(\text{proof}_a, \text{proof}_b) \;\leq\; \tau_{\text{neg}}.$$

Adding these harder negatives stabilizes training, yielding a gentler loss curve and better overall generalization. We experiment with different values of $k_{\text{neg}}$, as well as the $k_{\text{neg}}$ parameter. Figure 5 shows the results of the experiments. Figure 5 highlights the importance of hard negatives. Moreover, it provides insights into the impact of the number of hard negatives on the model's performance. Here we obtain little fluctuation, but observe slightly better results with $k_{\text{neg}} = 4$, which aligns well with existing research [32].

## 4.3 Training hyperparameters

Table 2 summarizes the core model and data-processing choices we use throughout training. We fine-tune the pretrained `microsoft/codebert-base` encoder, projecting its hidden states down to a 768-dimensional embedding and capping input sequences at 128 tokens. To create our splits, we randomly assign 70% of examples to training, 20% to validation, and 10% to testing. What is important is that we do the split by files, as we mainly compute pairwise distances between statements from the same file.

During the InfoNCE loss post–processing, we mark proof-pairs as positive when their distance falls below 0.3, negative when it exceeds 0.65, and treat

| Parameter | Value |
|---|---|
| base model | `microsoft/codebert-base` |
| embedding dim | 768 |
| max sequence length | 128 |
| splitting strategy | 70% train, 20% valid, 10% test |
| threshold pos. | 0.3 |
| threshold neg. | 0.65 |
| threshold hard neg. | 0.45 |
| hard negatives prob. | 30% |
| $k_{\text{negatives}}$ in loss | 4 |

Table 2: Model and Dataset hyperparameters

| Parameter | Value |
|---|---|
| algorithm | AdamW ($\beta_1 = 0.9,\ \beta_2 = 0.99,\ \lambda = 1\text{e}{-2}$) |
| schedule | linear warmup (10) – cosine decay (90) |
| lr | 4e−6 |
| batch size (statements) | 32 |
| dropout | 0.1 |

Table 3: Optimization hyperparameters

pairs with distance in [0.45, 0.65] as hard negatives with probability 30%. We found that sampling four negatives per positive yields the best balance between convergence speed and final performance.

Table 3 details the optimizer settings and regularization we employ. We use AdamW with $(\beta_1, \beta_2) = (0.9, 0.99)$ and a weight-decay factor of $10^{-2}$. The learning rate is warmed up linearly over the first 10% of updates to a peak of $4 \times 10^{-6}$. Training proceeds with batches of 32 statements, and we apply a dropout of 0.1 to the CLS embedding to guard against overfitting.

## 4.4 Training resources

During our final training run, the model consumed roughly 43 GB of GPU-process memory and only about 6% of the host's RAM. Over the course of 13.5 hours on a single NVIDIA H100 accelerator (with 20 CPU

cores and 160 GB of system memory), disk usage grew steadily from 28 GB to 76 GB as checkpoints and logs accumulated. GPU utilization stabilized above 85% shortly after the warmup phase and remained near saturation for the remainder of training, ensuring efficient use of the hardware. These measurements demonstrate that our setup runs comfortably on a single high-end GPU node with modest additional CPU and memory overhead.

| Group | $\leqslant$ **4** | | **5 – 8** | | **9 – 20** | |
|---|---|---|---|---|---|---|
| Ranker | Jaccard | RocqEta | Jaccard | RocqEta | Jaccard | RocqEta |
| GPT-4o | 48% ± 5% | **51%** ± 5% | 18% ± 4% | **25%** ± 3% | 11% ± 4% | 11% ± 5% |
| Claude 3.5 | 58% ± 5% | **61%** ± 4% | 28% ± 5% | **36%** ± 5% | 16% ± 5% | **21%** ± 5% |

Table 4: Model performance under different ablations across all evaluation sets.

# 5 Evaluation

To assess the efficiency of our proposed method, we conducted a series of experiments using the CoqPilot benchmarking framework. CoqPilot allows for easy setup and contributes to the reproducibility of our results.

We embed our retrieval mechanism *RocqEta* as a ranker in CoqPilot and, as said in § 2.4, evaluate it on the *IMM-300* dataset using different underlying models. We benchmark against a baseline that orders candidate theorems by decreasing Jaccard similarity: given a target statement $s_*$ and a pool of proven theorems $\{(s_i, p_i)\}$, we compute

$$J(s_*, s_i) = \frac{|S_{s_*} \cap S_{s_i}|}{|S_{s_*} \cup S_{s_i}|},$$

where each statement is tokenized using whitespace and punctuation, in practice, this yields identical scores to BM25. For each target theorem, we restrict candidates to the same file, rank them by either Jaccard or our trained embedding ranker *RocqEta*, select the top $k = 7$ premises, and feed them as a few-shot prompt to the model. We issue 12 generation attempts per theorem, and mark it solved if any generated proof is accepted by Rocq's system. Our evaluation metric is the fraction of theorems successfully proved. Table 4 demonstrates that RocqEta consistently outperforms the Jaccard-based baseline across nearly all difficulty levels. The most pronounced gains appear in the medium-difficulty group (proofs of 5–8 tactics), where syntactic similarity signals begin to break down and semantic embeddings capture deeper structural relationships. In the easiest group (proofs $\leqslant 4$ tactics), both methods achieve high success rates, since short proofs tend

to follow predictable patterns and the Jaccard heuristic remains effective. In the hardest category (proofs of 9–20 tactics), RocqEta continues to hold its own. Under GPT-4o, both rankers achieve an 11% solve rate, but more notably, with Claude 3.5, the RocqEta ranker improves from 16% to 21%, a 5% gain in an especially challenging setting. These results confirm that our ranker excels on medium-difficulty theorems and remains competitive when proofs grow longer and more varied.

Example in Figure 6 highlights the differences between different approaches to measure similarity between premises.

```
Lemma ext_sb_trans : transitive ext_sb.
Proof using.
  unfold ext_sb; red; ins.
  destruct x,y,z; ins; desf; splits; eauto.
  by rewrite H2.
Qed.
```

```
Lemma ext_sb_irr : irreflexive ext_sb.
Proof using.
  unfold ext_sb; red; ins.
  destruct x; ins; desf; splits; firstorder.
  lia.
Qed.
```

Figure 6: Theorems with dissimilar statements and similar proofs

If we measure the distance between theorems from Firgure 6 using the convential Jaccard distance, which is used by default in CoqPilot, we get 0.67:

$$\begin{aligned}\text{Jaccard\_distance}(t1, t2) &= 1 - \frac{|\{\texttt{transitive}, \texttt{ext\_sb}\} \cap \{\texttt{irreflexive}, \texttt{ext\_sb}\}|}{|\{\texttt{transitive}, \texttt{ext\_sb}\} \cup \{\texttt{irreflexive}, \texttt{ext\_sb}\}|} \\ &= 1 - \frac{1}{3} = 0.67\end{aligned}$$

Jaccard ranker focuses only on statement similarity, which in this case is relatively small, the only similar parts are highlighted with red. Jaccard would probably not select theorem `ext_sb_irr` as a premise for theorem `ext_sb_trans`; however, they have similar proofs and one could help the model to generate the proof for the other. Similar parts of the proofs are highlighted with yellow. If we measure the distance between these theorems using the `proof_similarity` metric we define, we get 0.32, and our trained model yields 0.28. When using our ranker, it is probable that one theorem would be selected as a premise for the other.

$$\begin{aligned}\text{proof\_sim}(t1, t2) &= 0.32 \\ \text{embedder\_pred}(t1, t2) &= 0.28\end{aligned}$$

## 5.1 Experiment resources

As all of the computations in the case of these particular experiments are running in the cloud of LLM providers, the experiments were conducted on a single MacBook Pro with an M1 chip. The only computationally expensive part of the experiments is launching multiple `Coq-LSP` servers at once (CoqPilot benchmark does that to optimize the time of the experiments and accelerate type-checking).

As we use a middleware service over LLM APIs, our financial estimations might not be accurate. However, we roughly estimate 12 generation attempts per theorem with seven contextual theorems at 12 cents per theorem for Claude 3.5 and 7 cents for GPT-4o. We run an experiment for 300 theorems and repeat it three times, resulting in 114 US Dollars.

# 6 Conclusion

We have presented BigRocq, which automates the augmentation of Rocq projects by transforming existing proofs into proof-state trees and emitting auxiliary lemmas for every intermediate state. Given a Rocq project, it utilizes `Coq-LSP` to extract goals before and after each tactic, builds a tree representation, and writes out new `.v` files containing the generated sublemmas. All proofs are batch-validated, and detailed statistics and interactive HTML visualizations of each proof tree are produced. We release BigRocq as an open-source CLI so the community can expand their Rocq corpora effortlessly.

We introduced a retrieval-augmented generation approach for Rocq, incorporating a neural premise selector based on a self-attentive embedding model. On a benchmark of 300 Rocq theorems evaluated with two different proof generators, our method achieved up to a 28% relative improvement over the Jaccard-based baseline. These findings indicate that using proof-aware embeddings for premise ranking can substantially boost proof synthesis performance, especially for the medium-difficulty theorems where purely token-based similarity falls short.

We open-source all code; therefore, RocqEta ranker is available to try from the CoqPilot plugin. Model is published at https://huggingface.co/kdizzled/rocqstar-ranker-theorem-embeddings. To explore the project one shall clone the mono-repo of the project from https://github.com/JetBrains-Research/big-rocq. To run the server of the ranker one needs to navigate to the `ranker-server` subdirectory and follow the guide in the `README`. Last step is to apply the git patch, which is located in the `CoqPilot+RocqStar` subfolder, to the CoqPilot project. This will add our ranker to the plugin and to the benchmarking system of CoqPilot.

## 6.1 Future Work

We plan to extend our work in several directions. First, we will explore the use of more advanced neural architectures for premise selection, such

as graph neural networks or transformer-based models. Second, we will investigate the integration of our retrieval-augmented generation approach with other proof assistants and theorem provers, such as Lean or Isabelle. Finally, we will conduct a more extensive evaluation of our method on larger and more diverse datasets to better understand its strengths and limitations.

# References


[1] Emilio Jesús Gallego Arias et al. *Visual Studio Code Extension and Language Server Protocol for Coq*. 2022. URL: https://github.com/ejgallego/coq-lsp.

[2] Jesse Alama et al. "Premise Selection for Mathematics by Corpus Analysis and Kernel Methods". In: *Journal of Automated Reasoning* 52.2 (Apr. 2013), pp. 191–213. ISSN: 1573-0670. DOI: 10.1007/s10817-013-9286-5. URL: http://dx.doi.org/10.1007/s10817-013-9286-5.

[3] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. DOI: 10.1007/978-3-662-07964-5.

[4] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. "The tactician: A seamless, interactive tactic learner and prover for coq". In: *International Conference on Intelligent Computer Mathematics*. Springer. 2020, pp. 271–277. DOI: https://doi.org/10.1007/978-3-030-53518-6_17.

[5] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for dependent type theory". In: *Journal of automated reasoning* 61 (2018), pp. 423–453. DOI: doi:10.1007/s10817-018-9458-4.

[6] Leonardo De Moura et al. "The Lean theorem prover (system description)". In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer. 2015, pp. 378–388. DOI: https://doi.org/10.1007/978-3-319-21401-6_26.

[7] Henrico Dolfing. *The $440 Million Software Error at Knight Capital*. 2019. URL: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html.

[8] et al. Enrico Tassi Romain Tetley. *Visual Studio Code extension for Coq*. 2022. URL: https://github.com/rocq-prover/vscoq.

[9] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[10] Emily First, Yuriy Brun, and Arjun Guha. "TacTok: Semantics-aware proof synthesis". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–31. DOI: https://doi.org/10.1145/3428299.

[11] Andreas Florath. *Enhancing Formal Theorem Proving: A Comprehensive Dataset for Training AI Models on Coq Code*. 2024. arXiv: 2403.12627 [cs.AI]. URL: https://arxiv.org/abs/2403.12627.

[12] David Freedman, Robert Pisani, and Roger Purves. "Statistics (international student edition)". In: *Pisani, R. Purves, 4th edn. WW Norton & Company, New York* (2007).

[13] Georges Gonthier. "The four colour theorem: Engineering of a formal proof". In: *Computer Mathematics: 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*. Springer. 2008, pp. 333–333. DOI: https://doi.org/10.1007/978-3-540-87827-8_28.

[14] Ronghui Gu et al. "CertiKOS: an extensible architecture for building certified concurrent OS kernels". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 653–669. ISBN: 9781931971331.

[15] Ivan Klimov. *Mechanized verification of pretty-printing library implemented in C*. Bachelor's thesis. 2023. URL: https://github.com/klimoza/verified-kisa/tree/master/thesis.

[16] Denis Kocetkov et al. *The Stack: 3 TB of permissively licensed source code*. 2022. arXiv: 2211.15533 [cs.CL]. URL: https://arxiv.org/abs/2211.15533.

[17] Wen Kokke, Jeremy G. Siek, and Philip Wadler. "Programming language foundations in Agda". In: *Science of Computer Programming* 194 (2020), p. 102440. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102440. URL: https://www.sciencedirect.com/science/article/pii/S0167642320300502.

[18] Andrei Kozyrev et al. “CoqPilot, a plugin for LLM-based generation of proofs”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 2382–2385. ISBN: 9798400712487. DOI: 10.1145/3691620.3695357. URL: https://doi.org/10.1145/3691620.3695357.

[19] Daniel Kühlwein et al. “Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics”. In: June 2012, pp. 378–392. ISBN: 978-3-642-31364-6. DOI: 10.1007/978-3-642-31365-3_30.

[20] Xavier Leroy et al. “CompCert-a formally verified optimizing compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[21] Zhouhan Lin et al. “A structured self-attentive sentence embedding”. In: *arXiv preprint arXiv:1703.03130* (2017).

[22] Jessica MacNeil. *Mariner 1 destroyed due to code error, July 22, 1962*. 2019. URL: https://www.edn.com/mariner-1-destroyed-due-to-code-error-july-22-1962/.

[23] Evgenii Moiseenko et al. “Relaxed Memory Concurrency Re-executed”. In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704908. URL: https://doi.org/10.1145/3704908.

[24] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. DOI: https://doi.org/10.1007/3-540-45949-9_5.

[25] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. “Representation learning with contrastive predictive coding”. In: *arXiv preprint arXiv:1807.03748* (2018).

[26] Semen Panenkov. *Mechanizing semantics of graph query languages in Coq*. Bachelor’s thesis. 2023. URL: https://github.com/cyphercert/opencypher-coq/blob/master/papers/panenkov-thesis.pdf.

[27] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. “Bridging the gap between programming languages and hardware weak memory models”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–31.

[28] Talia Ringer et al. "QED at large: A survey of engineering of formally verified software". In: *Foundations and Trends® in Programming Languages* 5.2-3 (2019), pp. 102–281. DOI: 10.1561/2500000045.

[29] Alex Sanchez-Stern et al. "Generating correctness proofs with neural networks". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2020, pp. 1–10. DOI: https://doi.org/10.1145/3394450.3397466.

[30] Kyle Thompson et al. "Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification". In: *arXiv preprint arXiv:2412.14063* (2024).

[31] Josef Urban et al. "MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance". In: *International Joint Conference on Automated Reasoning*. 2008. URL: https://api.semanticscholar.org/CorpusID:45162613.

[32] Chuhan Wu, Fangzhao Wu, and Yongfeng Huang. "Rethinking infonce: How many negative samples do you need?" In: *arXiv preprint arXiv:2105.13003* (2021).

[33] Kaiyu Yang and Jia Deng. "Learning to prove theorems via interacting with proof assistants". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6984–6994. DOI: https://doi.org/10.48550/arXiv.1905.09381.

[34] Kaiyu Yang et al. "Leandojo: Theorem proving with retrieval-augmented language models". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 21573–21612.