Synergy of Machine Learning and Automated Reasoning

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

dinsdag 5 december 2023
om 10:30 uur precies

door

Bartosz Paweł Piotrowski

geboren op 12 juni 1992
te Pruszków, Polen

Promotor:

- Prof. dr. Herman Geuvers

Copromotoren:

- Dr. Josef Urban (České vysoké učení technické v Praze, Tsjechië)
- Dr. Mikoláš Janota (České vysoké učení technické v Praze, Tsjechië)

Manuscriptcommissie:

- Prof. dr. Tom Heskes
- Prof. dr. Stephan Schulz (Duale Hochschule Baden-Württemberg Stuttgart, Duitsland)
- Dr. Konstantin Korovin (The University of Manchester, Verenigd Koninkrijk)
- Prof. dr. Jasmin Blanchette (Ludwig-Maximilians-Universität München, Duitsland)
- Prof. dr. Mateja Jamnik (University of Cambridge, Verenigd Koninkrijk)

Synergy of Machine Learning and Automated Reasoning

Dissertation to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.M. Sanders,
according to the decision of the Doctorate Board
to be defended in public on

Tuesday, December 5, 2023
at 10:30 am

by

Bartosz Paweł Piotrowski

born on June 12, 1992
in Pruszków, Poland

Supervisor:

- Prof. dr. Herman Geuvers

Co-Supervisors:

- Dr. Josef Urban (Czech Technical University in Prague, Czech Republic)
- Dr. Mikoláš Janota (Czech Technical University in Prague, Czech Republic)

Manuscript Committee:

- Prof. dr. Tom Heskes
- Prof. dr. Stephan Schulz (Baden-Württemberg Cooperative State University, Germany)
- Dr. Konstantin Korovin (The University of Manchester, United Kingdom)
- Prof. dr. Jasmin Blanchette (Ludwig Maximilian University of Munich, Germany)
- Prof. dr. Mateja Jamnik (University of Cambridge, United Kingdom)

For it is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be safely relegated to anyone else if the machine were used.

— Gottfried Leibniz

We may hope that machines will eventually compete with men in all purely intellectual fields.

— Alan Turing

Contents

1 Introduction **1**
1.1 The quest for mechanized reasoning 1
1.1.1 Automated theorem proving 4
1.1.2 Interactive theorem proving 6
1.2 The promise of learning from examples 9
1.3 Automated reasoning meets machine learning 11
1.3.1 Incorporating data-driven paradigm into formal tools . . . 12
1.3.2 Challenges of making machine learning enhancements . . 14
1.3.3 Can machine learning methods reason on their own? . . . 16
1.4 Thesis outline 18
1.4.1 Structure of the thesis 18
1.4.2 Main research questions 20

2 ATPboost: Learning premise selection in binary setting with ATP feedback **23**
2.1 Introduction: Machine learning for premise selection 24
2.1.1 Premise selection in binary setting with multiple proofs . 25
2.2 ATPboost: Setting, algorithms and components 26
2.2.1 Algorithms 26
2.2.2 Components 27
2.3 Evaluation 30
2.3.1 Parameter tuning 30
2.3.2 Incremental feedback loop with train/test split 32
2.3.3 Incremental feedback loop with no initial proofs 32
2.4 Conclusions and future work 34

3 Stateful premise selection by recurrent neural networks 37
3.1 Introduction: Premise selection over large libraries 37
3.2 Premise selection and neural machine translation 38
3.3 Data, their representation and augmentation 39
3.3.1 Initial data for training RNNs 40
3.3.2 Representation of the statements 40
3.3.3 Ordering of the premises 41
3.3.4 Augmentation with subproof data 43
3.3.5 Oversampling rare examples 44
3.4 Experimental evaluation 44
3.5 Results and discussion 46
3.5.1 Source and target combinations 46
3.5.2 Augmentation with subproof data and oversampling . . . 47
3.6 Subproofs as standalone data set 49
3.7 Examples of predictions from RNN 50
3.7.1 Theorem `t128_zfmisc_1` 50
3.7.2 Theorem `t30_tops_1` 53
3.8 Conclusions and future work 55

4 Guiding inferences in connection tableau by recurrent neural networks 57
4.1 Introduction 57
4.2 A data set for connection-style internal guidance 58
4.3 Neural modelling and evaluation metric 59
4.4 Results 60
4.5 Conjecturing new literals 62
4.6 Conclusions and future work 63

5 Towards learning quantifier instantiation in SMT 65
5.1 Introduction 66
5.2 Background 67
5.2.1 Enumerative instantiation 68
5.3 Learning ordering of terms 70
5.3.1 Featurization 71
5.4 Experimental evaluation 75
5.4.1 Experimental setting 77
5.4.2 Data for evaluation 79
5.4.3 Results and discussion 79
5.5 Related work 86
5.6 Conclusions and future work 86

6 Online machine learning techniques for Coq **89**
6.1 Introduction . . . 90
6.1.1 Contributions . . . 91
6.2 Tactic and proof state representation . . . 91
6.3 Prediction models . . . 93
6.3.1 Locality sensitive hashing forests for online k-NN . . . 93
6.3.2 Online random forest . . . 95
6.3.3 Boosted trees . . . 99
6.4 Experimental evaluation . . . 102
6.4.1 Split evaluation . . . 103
6.4.2 Chronological evaluation . . . 103
6.4.3 Evaluation in Tactician . . . 103
6.4.4 Feature evaluation . . . 105
6.5 Related work . . . 106
6.6 Conclusions and future work . . . 106

7 Machine-learned premise selection for Lean **109**
7.1 Introduction . . . 110
7.2 Dataset collection . . . 111
7.2.1 Features . . . 111
7.2.2 Relevant premises . . . 112
7.3 Machine learning models . . . 113
7.3.1 k-nearest neighbours . . . 113
7.3.2 Random forest . . . 114
7.4 Evaluation setup and results . . . 115
7.5 Interactive tool . . . 117
7.6 Future work . . . 118

8 Symbolic rewriting with neural networks **121**
8.1 Introduction . . . 121
8.2 Data . . . 122
8.2.1 The AIM data set . . . 122
8.2.2 The polynomial data set . . . 124
8.3 Experiments . . . 124
8.3.1 AIM data set . . . 125
8.3.2 Polynomial data set . . . 125
8.4 Integration data sets from Facebook Research . . . 127
8.5 Conclusions and future work . . . 130

Bibliography **133**

Research data management **155**

Summary **157**

Samenvatting **161**

Contributions **165**

Curriculum vitae **167**

Acknowledgments **169**

Chapter 1

Introduction

1.1 The quest for mechanized reasoning

Performing rigorous reasoning in an automated, mechanized way is an old dream. One could trace the origins of this idea, through Turing, Frege, and Leibniz, back to the medieval period and a Christian thinker Ramon Llull.[1]

Born in Mallorca in the 13th century, he spent his life trying to convert Muslims and Jews. This endeavour motivated him to create a new, rigorous method of reasoning. He realised that typical Christians' arguments lack solid grounding in commonly accepted truths, and are invariably bogged down in endless, undirected disputes.

Therefore, Llull attempted to construct a conceptual framework – or maybe even a *mechanism* – for reasoning abstracted from the beliefs of any specific religion, but based on their common grounds. The mechanism would *generate* truths from the assumed premises. It should be designed in such a way that once the input assumptions were agreed by the adversaries, they were forced to accept the produced conclusions by the objectivity of the procedure.

Llull's system nowadays appears to be quirky and difficult to comprehend, and we will not attempt to demonstrate it here. However, in Figure 1.1, we provide a glimpse into the Llull's work. It is one of the multiple graphical schemas

[1] For an interesting and thorough discussion of the seminal influence of Ramon Llull on the human thought leading to the modern logic and computer science, see [50] – a book vindicating the work of Llull, published on the occasion of the 22nd International Joint Conference on Artificial Intelligence (IJCAI) which took place in Barcelona in 2011. The initial part of this section is based on the first three chapters of this book.

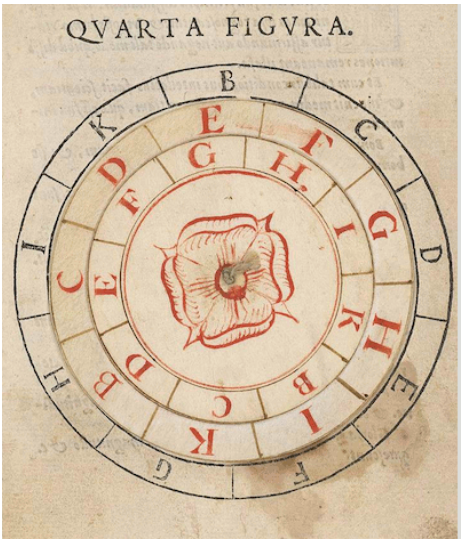


Figure 1.1: The *Fourth Figure* of the Lull's *Ars brevis*. It consists of three concentric circles capable of rotating around their common centre. Each of the circles is marked with the nine letters from B to K. The letters are constants signifying various abstract primitive concepts, and the rotating mechanism aims to facilitate combining the primitives into compound concepts.

included in his *opus magnum*. It contains three concentric, physically rotating circles described with nine letters – constants referring to certain primitive concepts. By rotating the circles, one could mechanically obtain multiple different *compound concepts*. This *Fourth Figure* of Llull illustrates three conceptual innovations he pioneered, that are recognized by Jeremy Avigad in [2], and which are now elementary ingredients of the modern formal reasoning methods:

- Concepts can be represented with symbols.
- Concepts can be combined into compound ones.
- Concepts can be manipulated in a mechanized way.

Despite its innovative qualities, the method of Ramon Llull did not get much traction in the medieval period. However, 400 years later, the idea of symbolic representations of concepts and rules for reasoning resonated with a great German polymath – Gottfried Leibniz. He wrote in a letter:

> When I was young, I found pleasure in the Llullian art, yet I thought also that I found some defects in it, and I said something about these in a little schoolboyish essay called *On the Art of Combinations*, published in 1666, and later reprinted without my permission.

Leibniz did not wish his *De Arte Combinatoria* to circulate widely, as he considered it incomplete. Still, this work – and its continuation – turned out to be highly influential. He develops there an idea of *characteristica universalis* – a universal language of thought – and a *calculus ratiocinator* – a calculus for reasoning. Leibniz envisions that if these ideas were perfected and widely adopted,

> [...] there would be no more need of disputation between two philosophers than between two accountants. For it would suffice to take their pencils in their hands, and say to each other: Let us calculate.

Leibniz made some rudimentary progress towards realizing his dream. Nowadays, we can recognize that along the way he developed roughly a propositional fragment of logic. A substantial continuation of this work came later – from Gottlob Frege at the turn of the nineteenth and twentieth centuries.

In his two-volume, seminal work – *Grundgesetze der Arithmetik* – he develops a logical calculus involving predicates, functions, and quantifiers. Unfortunately, the axioms chosen by Frege for his system turned out to be inconsistent, as was famously noticed by Bertrand Russell just before the publication of the second volume of *Grundgesetze*. Frege acknowledges it desperately in the appendix:

> There is nothing worse that can happen to a scientist than to have the foundation collapse just as the work is finished. I have been placed in this position by a letter from Mr. Bertrand Russell.

Despite the fatal flaw in the axioms, Frege's work has become a landmark in the history of logic and formal methods. His analysis of quantified statements and the strict notion of proof are accepted from the standpoint of modern logical standards.

An inconsistency found in Frege's system by Russell motivated the latter to develop his own new logical foundations for mathematics. His grand *Principia Mathematica* written together with Alfred Whitehead develops the basis of *type theory*, whose modern extensions have profound importance in the current formal methods.

Thanks to Frege, Russell, and many other pioneers in logic at the beginning of the twentieth century, the conceptual apparatus has become rich enough, a critical mass has been achieved, and logic became a flourishing scientific discipline. It developed a rich language and original methods. It secured firm foundations for modern mathematics. In addition to being a useful tool, it has become a topic of study in its own right. Its strengths, but also its inherent limitations (*viz.* the famous Gödel theorems) have been discovered.

In parallel to the developments in logic and foundations of mathematics, in the twentieth century, a digital computer was born. On the theoretical front, Alan Turing ultimately refined the notion of an algorithm in his work "On Computable Numbers, with an Application to the Entscheidungsproblem" published in 1936. There, the *Turing machine* was defined – a conceptual, idealized device for modelling all possible computations.

In addition to the theoretical advancements, physical implementations of general-purpose computing machines became feasible. In conjunction with the freshly created firm axiomatic foundations of mathematics, this meant that the great dream about mechanized reasoning finally started becoming truly real – at least within the realm of mathematics.

Since then, the discipline of automated reasoning has been advanced with sustained efforts, driven by visionary goals and practical applications, intertwined with developments of computer science, formal logic, and mathematics.

Jumping to the present day, the discipline may be seen as having two major subfields focusing on different goals: *automated theorem proving* and *interactive theorem proving*. We briefly characterize both of them below.

1.1.1 Automated theorem proving

The main goal of automated theorem proving is to establish the truth of formal conjectures without human intervention during the process. Somewhat surprisingly, historically, researchers focused on this problem first rather than on the seemingly easier problem of constructing formal proofs with human guidance – the domain of *interactive* theorem proving discussed in Subsection 1.1.2.

One of the very first programs designed to prove theorems automatically was *Logic Theorist* implemented already in 1956 by Allen Newell, Herbert Simon, and Cliff Shaw. Famously, it was able to prove 38 of the first 52 theorems from *Principia Mathematica* – and some of the automatic proofs were reportedly more elegant than their human counterparts.

When in late 1956 Simon wrote to Russell (85 at the time) to describe the work on Logic Theorist, Russell replied:

> I am delighted to know that *Principia Mathematica* can now be done by machinery. I wish Whitehead and I had known of this possibility before we wasted ten years doing it by hand. I am quite willing to believe that everything in deductive logic can be done by machinery.

Nowadays, we know that far from everything in deductive logic can be proved mechanically. Still, the technology of automated deduction has become a successful and often indispensable tool in many areas – especially for industry-originating problems. It developed several important subfields, in particular:

SAT that develops algorithms to establish the *sat*isfiability of propositional formulas. Despite the theoretically unwieldy complexity of the problem (which is NP-complete), in practice, SAT solvers are able to deal with very large formulas encoding various important practical problems. As expressed by Edmund Clarke – a 2007 Turing Award recipient – "Clearly, efficient SAT solving is a key technology for 21st-century computer science."

SMT (*satisfiability modulo theories*) that generalizes SAT by incorporating specialized decision procedures for dealing with a plethora of specific theories, like linear integer arithmetic, arrays, or bit vectors. SMT solvers typically consist of a *SAT solver* and a *theory solver*, and these two modules guide each other in a feedback loop towards finding a contradiction or declaring the satisfiability of the input problem. Although SMT solving was initially dedicated to ground problems only, some SMT solvers may now also work with quantified formulas – see Chapter 5. Examples of strong SMT solvers include cvc5 [8], Z3 [38], veriT [24], or Yices [44].

FOL automated theorem proving that is concerned with developing algorithms for proving theorems in full classical *first-order logic*. These provers are based on various reasoning strategies, including tableaux calculus (where a tree of literals is being built by applying clauses of the input problem until all the branches contain complementary literals; see Chapter 4) or saturation looping (where the input problem is clausified and all possible inferences are being generated – until a contradiction is found, or no new clause can be produced). The TPTP language is the standard input language for most modern first-order provers. CASC (CADE ATP System Competition) is an annual "world championship" addressed to (mainly but not only) first-order provers. Saturation-based theorem provers like E [144], iProver [91], or Vampire [92] regularly achieve state-of-the-art performance for first-order problems in this competition.

HOL automated theorem proving that is concerned with developing algorithms for proving theorems in *higher-order logic*, i.e., logic that adds expressiveness to the first-order logic by allowing quantification for variables referring

not only to individual elements of the universum, but also higher-order concepts (like sets, predicates, or functions). Some of the most prominent HOL automated theorem provers are Satallax [28] and LEO-III [148], and more recently also Zipperposition [165], and higher-order versions of E [166] and Vampire [16].

1.1.2 Interactive theorem proving

In interactive theorem proving, proofs of mathematical claims are constructed with the use of *proof assistants*. These are computational systems that allow a user to input the proof using a specialized language – much like a programming language. The proof assistant then checks if the proof is *correct* – specifically, if it follows the rules and axioms encoded in the system. Additionally, various proof assistants provide various degree of automation or interactiveness during the proof development. Many of them feature a variety of *tactics* being able to automatically make smaller or larger leaps in the proof construction, relieving the user from explicitly stating every atomic logical step of the reasoning.

The world of interactive theorem proving is diverse [168] and proof assistants are based on various logical foundations. For instance, Mizar [60] builds on *set theory* (with a soft-typing layer), HOL-family provers (like HOL Light [64] or HOL4 [147]) are based on *simple type theory*, whereas Coq [155] and Lean [37] use *dependent type theory* (more specifically, its variant known as *calculus of inductive constructions*). Some interactive theorem provers, like Metamath [105] or Isabelle [123], aim at being generic – agnostic with respect to any specific logical foundational systems – and provide meta-logical frameworks.

Different logical foundations have different merits, and choosing a good foundation for a proof assistant, balancing expressiveness, simplicity, and convenience for mathematicians is an important research question [13].

Proof assistants are useful for at least four important reasons:[2]

Providing correctness guarantees. This is perhaps the main motivation for formalizing proofs in proof assistants. As mathematics gets more complex and specialized, it is increasingly difficult to establish the correctness of proofs via the standard means of peer-reviewing. As a small but vivid example of human unreliability in this domain see Figure 1.2 showing a footnote from *The Axiom Of Choice* by Thomas Jech [77]. Sometimes it also happens that a proof contains large computational component, or is complex to such an extent that despite excessive effort reviewers are not able to achieve complete certainty regarding the proof correctness. A remarkable example of such a situation was

[2] One could find many other reasons justifying the growing relevance of interactive theorem proving nowadays – a far more complete account is presented by Jeremy Avigad in [3].

when Thomas Hales submitted his proof of the long-standing Kepler conjecture to the prestigious *Annals of Mathematics*. The referees accepted the proof for publication [61], yet they admitted they were not *completely certain*, but rather "99% certain" of its correctness. This verdict was unsatisfactory for Hales and he decided to formalize the proof. This project (code-named Flyspeck) required an extensive effort and a long time – but was finished successfully [62].

[1] The result of Problem 11 contradicts the results announced by Levy [1963b]. Unfortunately, the construction presented there cannot be completed.
[2] The transfer to ZF was also claimed by Marek [1966] but the outlined method appears to be unsatisfactory and has not been published.
[3] A contradicting result was announced and later withdrawn by Truss [1970].
[4] The example in Problem 22 is a counterexample to another condition of Mostowski, who conjectured its sufficiency and singled out this example as a test case.
[5] The independence result contradicts the claim of Felgner [1969] that the Cofinality Principle implies the Axiom of Choice. An error has been found by Morris (see Felgner's corrections to [1969]).

Figure 1.2: A footnote at the bottom of page 131 of *The Axiom Of Choice* by Thomas Jech [77].

Aiding proof invention. Typically, the user should have at least a general idea for a proof before attempting its formalization. Yet, as many proof assistants have tactics automatically taking care of simpler, more mundane details of reasoning, the attention of the mathematician can remain directed to a higher level of reasoning. We should note, however, that currently, the automation in modern proof assistants still tends to be weak, being able to target more obvious reasoning steps. But the situation certainly improves, and we may anticipate stronger tactics with the ability of making critical progress in a proof. Chapter 6 and Chapter 7 contribute towards achieving such stronger automation.

Apart from the automation, many proof assistants help in development by keeping track of – and conveniently displaying – the current *goal* and a set of (local) hypotheses that can be used to progress. This vastly helps to navigate complex and large proofs.

Aiding proof understanding. The ability of navigating a user through a large proof by exposing the exact proof state not only helps to construct proofs, but also aids understanding of them.

Peter Scholze – a Fields Medal awarded mathematician – challenged the Lean proof assistant community to formalize his research-level result in analytic geometry, which resulted in the Liquid Tensor Experiment [142] led by Johan Commelin. After the project was completed, Scholze said in an interview [143]:

> Really, one key problem I had when I was trying to find this proof was that I was essentially unable to keep all the objects in my "RAM," and I think the same problem occurs when trying to read the proof. Lean always gives you a clear formulation of the current goal, and Johan confirmed to me that when he formalized the proof of Theorem 9.4, he could — with the help of Lean — really only see one or two steps ahead, formalize those, and then proceed to the next step. So I think here we have witnessed an experiment where the proof assistant has actually assisted in understanding the proof.

Facilitating collaboration. Proof assistants facilitate to develop large mathematical proofs *collaboratively*, in a similar style as programmers work together on developing software. The formal language ensures that the notions are used consistently, and the proof assistant guarantees eliminating mistakes. This makes it possible for a large group of formalizers to build sizable proofs and comprehensive, uniform libraries of formal mathematics.[3] This is a big cultural change in mathematics and possibly a trend anticipating the future norm.

Apart from the four mentioned reasons motivating the use of proof assistants, there are two additional considerations important in the context of this thesis:

Interactive and automated theorem proving benefit from each other. Automated methods provide strong automation of the proof development, for instance via so-called *hammers* [22]. On the other hand, formal libraries are sources of hard, interesting problems motivating development of stronger methods in automated theorem proving [159].

Formal mathematics enables development of AI methods. Formal libraries are sources of high-quality data that can be used to develop AI / machine-learning / data-driven approaches enhancing automation in proof assistants, therefore making them easier to use. This topic is developed in Subsection 1.3.1.

[3] As an example of the scale of formal libraries today: Lean's `mathlib` [104], which is hosted on GitHub, as of 11th May 2023 has been developed by a community of 286 contributors. Since 21st July 2017, they authored 17 972 commits resulting in 863 103 lines of non-whitespace code.

1.2 The promise of learning from examples

Below, there is an equation expressing Kepler's first law of planetary motion:

$$r = \frac{p}{1 + \epsilon \cdot \cos(\phi)}.$$

r is the distance of the planet from the Sun, ϕ is the angle between the planet's current position and its closest approach, as seen from the Sun, and $p \geq 0$ and $0 \leq \epsilon \leq 1$ are fixed parameters specific for the planet.[4]

How did Kepler discover this formula? It was not via a metaphysical reflection nor from some *a priori* principles, but primarily by collecting rich data coming from his (and Tycho Brahe's) astronomical observations, and attempting multiple times to fit some mathematical model to the collected data points.

What Kepler has done manually, *machine learning* (ML) algorithms are intended to perform automatically. The defining aspect of these algorithms is that they derive a useful data-processing *model* from a data set of examples representing the data-processing task. In machine learning vernacular, the process of deriving such a model based on data is called *training* or *fitting*.

This data-driven paradigm can be opposed to an alternative approach where the computer would be programmed manually with an explicit, non-adaptive algorithm encompassing expert knowledge for performing a given task.

The tasks that machine learning algorithms are supposed to learn can be thought of as finding an appropriate mapping $f\colon X \to Y$. For instance, using Kepler's example, mapping the angle of the planet to its distance from the Sun, or mapping pictures of handwritten digits to the digits themselves. The examples used for training are then multiple different (*input, output*) pairs. The *output* is often called a *label*, whereas the *input* is often called *features*.

In machine learning – as opposed to our astronomical example – the learned mapping f is not required to be fully precise – only to be as precise as possible.

A machine learning algorithm is expected to automatically pick up patterns and logic behind the concept represented by training examples. The produced mapping should fit the training examples well, but crucially, it should also *generalize* – predict correct outputs for new, unseen inputs. (Producing a model that only performs well on training examples would be trivial – it would amount just to "memorizing" all the training examples in a database.) A model that fits training data well but does not generalize is said to *overfit*.

[4] In geometric terms, this equation means that the orbit of the planet is an ellipse with the Sun at one of the two foci.

Therefore, there are two measures of predictive performance of a trained machine learning model: the *training performance* that measures how well the model predicts correct outputs for the training examples, and the *testing performance* that measures the correctness of the model's predictions for new examples, unseen during training. Achieving high testing performance is essential; training performance is more of a diagnostic measure – if a machine learning algorithm cannot produce a model performing well on training examples, it is an indication that either the algorithm is flawed, or the problem represented by the examples is too complex to be modelled by the given method.

In practice, to measure the testing performance of a machine learning model, a set of available examples is typically split into two parts: a *training set* (used for training) and a *testing set* (used solely for evaluation).

There is a large variety of machine learning algorithms producing data-processing models of different forms and complexity. In general, the more complex the data and the task to learn, the more sophisticated machine-learning approach is needed to achieve good results. The traditional taxonomy of machine learning approaches splits them along several orthogonal criteria. Below we describe a split that is quite relevant in the context of this thesis (as we apply approaches from both categories): *eager* algorithms *vs lazy* algorithms.

- **Eager algorithms** have a proper training phase resulting in a standalone model. After this phase, the training data is no longer needed for the model to produce responses, as it internalized the patterns from the data.

  There is a wide spectrum in the complexity of possible models. On the one side of it there are *linear models* being able to only correctly represent linear relations. Fitting their parameters to data typically amounts to reversing and multiplying matrices, which can be done very efficiently.

  On the other side of the spectrum, we may find *deep neural networks*, that may be seen as compositions of multiple layers consisting of multidimensional linear functions interspersed with simple, non-linear ones. Their training is typically performed via the means of *stochastic gradient descent*. Deep neural networks may sometimes contain *billions* of trainable parameters, and their training may take *months* and often utilizes specialized hardware.

- **Lazy algorithms** do not have an actual training phase and do not produce a proper model that is separate from the training data. Instead, the training dataset itself is directly used by the algorithm while producing responses.

The algorithms from this class are often simpler, yet for many problems entirely sufficient. The advantage of the lack of training phase is counterbalanced by the disadvantage of more intensive computation during the prediction time.

A representative lazy machine learning algorithm is k-nearest neighbors (k-NN). Given an unlabeled example, k-NN produces a prediction by extracting the labels of the k most similar examples in the data set and returning an averaged (or most frequent) label.

Finishing this brief outline of machine learning, we need to mention impressive achievements in a newly emerged domain of so-called *generative artificial intelligence*. Models developed in this domain are generally based on large neural networks and instead of classifying objects or predicting simple, unstructured responses they are trained to *generate* human-like, original, complex outputs in the form of text [134, 153], image [135, 140], or audio [70]. These outputs often are conditioned on *prompts* – textual inputs influencing the desired output.

The techniques of generative AI clearly belong to the data-driven paradigm, yet they tend to escape the classical understanding of machine learning. The generative models are still trained on sets of examples, yet they, in a sense, transcend the information explicitly demonstrated in the examples and attain more advanced abilities. This is the case, e.g., for *large language models* that are trained using a simple objective: given a partial sentence predict a subsequent word. When a language model is trained in such a way on sufficiently rich and high-quality textual data (readily available in the age of the internet), one often sees the emergence of various capacities from the model, like translation between languages, text summarization [29, 134], or simple reasoning skills. The latter is interesting in the context of automated reasoning, and Chapter 8 is dedicated to an initial exploration of this phenomenon.

1.3 Automated reasoning meets machine learning

Automated reasoning and machine learning have contrasting natures: the former relies on precise calculation and produces exact results; the latter learns from borderless sets of examples to deliver only approximate solutions.

Despite these two paradigms being very different, it seems natural to incorporate both of them when designing methods for automating the development of mathematics. After all, both of them are present in human mathematical practice: On the one hand, in order to prove a theorem, one must ensure that every step is *logically correct*. On the other hand, a mathematician learns how

to search for a proof by *learning patterns* appearing in the proofs of similar problems.

This motivates the general theme of this Ph.D. thesis: joining the realms of automated reasoning and machine learning. More specifically, the main target of the thesis is applying various machine-learning-based methods to improve the success rate of automated theorem provers and to facilitate construction of formal proofs in proof assistants. Therefore, the goal is not to *substitute* the existing formal tools with completely new ones, based primarily on machine learning, but rather to *incorporate* into the state-of-the-art formal tools data-driven approaches that learn from past successes and failures.

In the subsection below, we provide a high-level characterization of how that can be achieved, and where machine learning approaches may be applied to improve the performance of the formal techniques. Later, we emphasise that applying machine learning in the context of automated reasoning needs to overcome substantially different challenges than those encountered in other, more common scenarios of using data-driven approaches. Finally, we discuss other possible modes of applying machine learning to challenges arising in automated reasoning where ML is not used to enhance existing formal tools, but rather constitutes standalone, new methods.

1.3.1 Incorporating data-driven paradigm into formal tools

Given an **automated theorem proving** algorithm, there are essentially two ways of enhancing it with data-driven guidance via machine learning:

- **External guidance:** this is a situation where machine-learned advice does not influence the prover / solver during its run, but only its *initial setting*. This may be, for instance, choosing the initial parameters of the prover that the ML-advisor finds good for the current input problem [26, 73, 96]. This may be also performing *premise selection* – making the input problem smaller by selecting only a subset of the axioms from it that seem the most relevant for the conjecture included in the problem. Chapters 2 and 3 discuss the premise selection problem in more detail and propose two novel machine learning methods for dealing with it.

- **Internal guidance:** this is a situation where a machine-learning advisor influences the prover / solver *during* its run. This often can be done by substituting various heuristics implemented in the provers by data-driven approaches. This may be, for instance, using a machine-learned mechanism for selecting clauses for resolution in a saturation-based theorem

prover, selecting clauses for extension steps in tableaux theorem prover (which is the topic of Chapter 4) or selecting terms for performing instantiation of quantified formulae in an SMT solver (the target of Chapter 5).

In the context of **interactive theorem proving**, machine learning methods may be targeted towards two important problems:

- **Searching in a formal library:** All major proof assistants are associated with large community-developed libraries containing thousands of formalized definitions and theorems. Therefore, one of the major challenges in constructing formal proofs of theorems depending on multiple other results is the prerequisite of having a good familiarity with the structure and the contents of the library. This may be mitigated by machine-learning-based tools for suggesting definitions and theorems already existing in the library that may be useful in a given proof state. Chapter 7 presents such a tool integrated with the Lean proof assistant.

  Note that this problem is similar to premise selection as described in the context of automated theorem proving. However, in the interactive scenario, premise selection methods need to be optimized for somewhat different goals:

  - The set of suggested premises does not necessarily need to be *complete* (containing all the premises needed for proving a theorem, as is necessary for ATPs) but it should be *precise* – containing as few irrelevant premises as possible. This is because the user can browse through a relatively small number of suggestions, so it is better when this set is smaller, but contains only relevant premises.
  - The tool should produce suggestions *fast*, as a user wants a quick feedback in real time when developing a proof. This means that slower machine learning methods may not be appropriate.
  - The machine learning model should have the possibility to be trained in the *online* fashion, i.e., by incorporating training examples one-by-one. This allows the model to be trained along with the development of the formal theory, so that it can suggest also premises recently formalized by the user.
  - Optimally, the tool should be lightweight, tightly integrated with the proof assistant, and easy to install, to provide a seamless user-friendly experience.

- **Suggesting subsequent proof steps:** Formal proofs contain many routine, repetitive steps, and machine learning methods may easily lower the cognitive effort of formalizers by suggesting proof steps that with high certainty progress the proof state in a good direction. Such suggestion methods may be used in conjunction with a proof search algorithm so that not only the proof state is advanced by one step, but a whole goal is closed. Chapter 6 addresses this problem and develops methods for tactic prediction and proof automation for the Coq proof assistant.

  This problem is akin to internal guidance in automated theorem provers, but again – the emphasis is put differently in the interactive context, and the last three points from the list above apply also here.

1.3.2 Challenges of making machine learning enhancements

Applying machine learning techniques to problems arising in automated reasoning has its own specific challenges that are not characteristic for other, more typical applications of machine learning. Below, we highlight predominant ones:

- **Measuring improvements:** When an automated theorem prover is augmented with machine-learned advice, the effectiveness of such an improvement needs to be evaluated experimentally. To design a good evaluation, one needs to face two important questions:
  - **How to split data into training and testing parts for evaluation?** Doing such a split is a standard procedure when evaluating machine learning methods (see Section 1.2), and often data is split simply *randomly* into the two parts. However, corpora related to automated reasoning are often very non-uniform in terms of complexity, and moreover, there are many dependencies hidden within them. For these reasons, a random split may often not be appropriate for evaluation. For instance, libraries of formalized mathematics contain theorems which depend on each other, and which were formalized in some particular chronological order. One should preserve the chronology when spiting the library into training and testing parts. However, parts developed later tend to be more complex than the initial fragments of the library, which may skew the evaluation of the method. See Sections 6.4 and 7.4 for more discussion related to this issue and proposed approaches.
  - **How to determine if the improvement in performance is robust?** Automated theorem provers often are complex systems that are addi-

tionally *fragile* – even small, seemingly insignificant changes in their implementation (such as shuffling the order of formulae of the input problem) may cause a failure to solve problems solvable by a prover (under a fixed time limit) before; or the other way around – some new problems may suddenly become solved [149].

This fragility becomes problematic when evaluating an enhancement (ML-based or not) introduced to the prover. The enhanced prover may solve more problems in a benchmark compared to the baseline prover. However, we cannot be sure whether these newly solved problems were solved *thanks* to the idea behind the enhancement. It could happen that we just introduced an accidental perturbation – not related to the original idea at all – in the complex system which a theorem prover is, and that helped it to solve additional problems.

This issue is especially serious when the achieved improvement in terms of the number of the benchmark problems solved is relatively small – which is typical.

To mitigate this issue, it is important to not rely on one-off comparison of the base and enhanced prover. Instead, it is recommended that some non-essential parts of the algorithm of the enhanced prover are randomized, and it is evaluated on a benchmark multiple times, and the results are averaged. Randomness could be introduced, as suggested in [149], to all these decisions of the algorithm which we deem non-essential for the success rate of the prover. In the case of machine-learning-based enhancements, we may not need to randomize other parts of the base prover, as most of the machine-learning techniques are inherently randomized. We embrace the methodology of evaluation by multiple randomized runs in the projects described in Chapters 2 and 5.

- **Featurizing mathematical formulae:** The classical machine learning approaches (the non-neural ones) typically work on *tabular data* meaning that the inputs to an ML-model are fixed-length vectors of *features*. The features are numerical (or categorical) qualities meant to provide characterization of the examples that is useful for the learned task. The right design of the features often significantly influences the ability of the ML algorithm to learn.

  This thesis often tackles problems where the inputs to a machine learning model are based on symbolic formulae which cannot be processed by the model directly, but first need to be *featurized*. There is no perfect way of

doing that – the feature encoding will either be overly complicated (making the input space too complex for the ML model) or will not represent the meaning of the formula sufficiently precisely. This means that featurizing formulae is inherently hard and needs to be engineered carefully. See Chapters 2, 5, 6, and 7 where the featurization problem is tackled.

- **The speed versus quality trade-off:** Evaluating a machine-learning predictor inside an automated theorem prover during its run introduces a slow-down, and the more complex the ML model, the higher the slow-down. When the evaluation involves time limits for the prover's runs (and it always does) this creates an issue: even if the ML model correctly guides the prover, its overall performance under a time limit may easily decrease. On the other hand, if the ML model is fast, it is also typically simple, and thus may provide lower-quality guidance. This trade-off always needs to be taken into account when designing ML-augmentation for automated theorem provers. Chapter 5, where machine-learning-based guidance is developed for instantiation in an SMT solver, deals with this problem.

1.3.3 Can machine learning methods reason on their own?

The main theme of the thesis is applying machine learning to enhance existing tools in automated reasoning. However, this is not the *only* theme of the thesis, and not the only way that ML can be helpful in formal domains.

Below, we discuss a few different important domains of problems where more advanced machine learning approaches can be applied, in a sense, directly, without an intermediary reasoning system being augmented with ML.

- **Autoformalization:** This is a task of automatically translating from natural language mathematics to a formal language. As manually formalizing mathematics is tedious and requires specialized expertise, an autoformalization system would have significant practical implications. For that reason, various approaches to autoformalization have been attempted since the 2000s [85, 86, 87, 173]. However, it was only after the advent of large neural networks and neural language models when strong and practical solutions started to emerge. Neural language models were applied to the autoformalization problem for the first time in [167], where (unexpectedly) good accuracy of translation between synthetic statements in LaTeX and their formal counterparts in Mizar was achieved. Later works using neural language models for autoformalization include [169] and [4]. [154]

is a recent position paper concerning the importance of autoformalization and possible ways of achieving it.

- **Conjecturing:** This is a task of forming mathematical conjectures, which lies at the core of mathematical activity. Generated conjectures can either be intermediate lemmas helping to prove hard problems, or unrestricted, new, interesting conjectures based on a theory (a task known as *theory exploration*). A pioneering work on the topic is [99]. More recent approaches based either on statistical methods or neural models started to be developed in [55] and [160]. The Section 4.5 of Chapter 4 describes more modest experiments on conjecturing literals on branches in tableaux proofs using recurrent neural networks.

- **Symbolic rewriting:** This is a problem when one symbolic expression is being rewritten into another, satisfying desired properties. Many common problems can be cast as symbolic rewriting problems; examples of such include: normalizing polynomials, integrating arithmetical expressions, computing conjunctive normal form of propositional formulae, applying logical inference rules, etc.

  Many such problems can be solved with deterministic algorithms – either simple ones (as for normalizing polynomials) or more complex (like the Risch algorithm for integrating [139]). However, it was discovered that neural network architectures designed for natural language tasks have also the potential to efficiently deal with symbolic rewriting problems, including the more complex ones. One of the very first explorations of this topic was conducted by the thesis author and is reported in Chapter 8. Works continuing this research avenue in more depth include [98] and [32].

  Neural networks by their very nature are only approximate and one cannot expect perfect correctness from a network trained for a rewriting task. Yet, this approach is still interesting for two reasons: Firstly, in some situations, having a solution that is only *probably* correct is sufficient to be useful. This is true, for instance, in situations when generating a solution is difficult, but verifying its correctness is easy – as is the case of integrating complicated arithmetical expressions. Secondly, the ability of the neural networks to perform such symbolic tasks seemingly requiring precise algorithmic computation is an interesting phenomenon on its own that deserves effort of explaining it by machine learning researchers.

1.4 Thesis outline

Having sketched the background of this thesis – the research fields of automated reasoning (Section 1.1), machine learning (Section 1.2), and their intersection (Section 1.3) – we now outline the structure of the thesis (Subsection 1.4.1), and we list major research questions this thesis targets (Subsection 1.4.2).

The main content of the thesis consisting of Chapters 2–8 is based on six conference publications [76, 126, 127, 129, 130, 172] and one workshop publication [131]. I was the main author of five out of all these seven works.

The detailed contributions are specified at the end of the manuscript, on page 165.

1.4.1 Structure of the thesis

The thesis may be split into four parts concerning four different problems on the intersection of automated reasoning and machine learning.

The first part of the thesis focuses on the premise selection task in automated theorem proving. This is a critical task when an automated theorem prover (ATP) is used over a large theory where typically only a small fraction of the available facts are relevant for proving a new conjecture. Giving too many redundant premises to the ATP significantly decreases the chances of proving the conjecture.

Chapter 2 introduces the ATPboost system addressing this problem. It solves sets of large-theory problems by interleaving ATP runs with machine learning of premise selection from the proofs. Unlike many approaches that use a multi-label setting, the learning is implemented as a binary classification that estimates the pairwise relevance of (*theorem, premise*) pairs. ATPboost uses for this the gradient boosting decision tree algorithm. Learning in the binary setting however requires negative examples, which is nontrivial due to many alternative proofs. We implement several solutions of this problem in the context of the ATP/ML feedback loop and show a significant improvement over the multi-label approach.

In Chapter 3, a novel method for premise selection is developed based on recurrent neural networks (RNNs). Unlike the previous method which chooses sets of facts independently of each other by their rank, the new method uses the notion of *state* that is updated each time a choice of a fact is made. The new method is combined with data augmentation techniques. The evaluation shows improvements in terms of the number of new problems solved in comparison to the previous approach.

The second part of the thesis focuses on internal guidance for ATPs. Certain parts of their algorithms require non-deterministic choices to be made. These choices are normally either randomized or governed by pre-designed heuristics. The goal is to provide machine-learned advice instead, and by this improve the performance.

In this spirit, in Chapter 4, experiments with applying RNNs for guiding clause selection in the connection tableau proof calculus are described. The RNN encodes a sequence of literals from the current branch of the partial proof tree to a hidden vector state; using it, the system selects a clause for extending the proof tree. Additionally, a conjecturing experiment is performed where the RNN does not select an existing clause but completely generates the next tableau goal.

In Chapter 5, we develop an approach of applying ML to solve quantified satisfiability modulo theories (SMT) problems more efficiently. We focus on the enumerative instantiation method of solving quantified formulas. The task is to select the right ground terms to be instantiated. In ML parlance, this means learning to rank ground terms. We devise a series of features of the considered terms and train on them using gradient boosted decision trees. The experiments demonstrate that the ML-guided solver enables us to solve more problems than the base solver and reduce the number of quantifier instantiations.

The third part of the thesis develops ML-based automation for proof assistants. Formalizing mathematics using proof assistants is a laborious task requiring expert knowledge. The formal proofs need to deal with low-level reasoning steps. Also, a mastery of the existing formal library is required in order to reuse formalized theorems. To make proof assistants more user-friendly various forms of automation need to be developed. Here, ML-based approaches learning from already completed proofs are developed.

Chapter 6 focuses on the Coq proof assistant. Its proofs consist of sequences of *tactics* that modify *proof states*. The goal is to learn to suggest the next tactic in a given proof state. We build on top of Tactician, a plugin for Coq that provides a framework for learning from proofs written by the user to synthesize new proofs. Learning happens in an online manner, meaning that the ML model is updated every time the user performs a step in an interactive proof. This provides the user with a seamless, interactive experience, and it takes advantage of the locality of proof similarity: proofs similar to the currently constructed proof are likely to be found close by. Two online methods are implemented: k-nearest neighbors based on locality sensitive hashing and custom online random forest. We compare the relative performance of these methods on Coq's standard library.

In Chapter 7, we introduce an ML-based tool for the Lean proof assistant that suggests *relevant premises* for a theorem being proved by a user. The tool is based on a modification of the custom random forest model used in the Coq project. It is implemented directly in Lean, which was possible thanks to the rich and efficient metaprogramming features of Lean 4. The random forest is trained on data extracted from mathlib – Lean's formal mathematics library. The advice from the trained model is accessible to the user via a command that can be called while constructing a proof interactively.

The last part of the thesis investigates the capabilities of neural language models in the context of mathematics. More specifically, in Chapter 8, we investigate if the current neural architectures are adequate for learning symbolic rewriting. Two kinds of data sets are proposed for this investigation – one derived from automated proofs and the other being a synthetic set of polynomial terms. The experiments with neural machine translation models are performed and their (surprisingly) good results are discussed.

1.4.2 Main research questions

Below we emphasise major research questions this thesis addresses and we identify specific preexisting research shortfalls it fills in.

- **How far can one go with classical machine learning algorithms applied to premise selection?**

  The works preceding this thesis that focused on applying data-driven methods to premise selection either used simpler, classical machine learning approaches (like k-NN [80], naive Bayes [159], or kernel methods [156]), or applied novel, but computationally involving deep learning methods [71]. A research gap remaining to be filled in was to investigate the possibility of handling premise selection with the state-of-the-art, strongest available *classical, fast, non-neural* machine learning method, i.e., *gradient boosted trees*. The possibility of improvement compared to the simpler ML methods was not obvious as gradient boosted trees are not directly applicable to premise selection and special preprocessing of the data is additionally required. Chapter 2 contributes positive results regarding this research question.

- **How can the relations between the predicted premises be modelled?**

  So far, all the previous publications dealing with premise selection (including [71, 80, 159]) cast this problem as a *retrieval task*, where premises

are treated as independent entities, and a *ranking* of them is returned conditioned on a theorem being proved. This approach, however, oversimplifies the situation in that it does not model the existing – but hidden – *relations between premises*: some groups of lemmas may work well together; on the other hand, it may be unlikely to see some pairs of them in one proof. Moreover, theorems often have multiple proofs using different sets of premises. This motivates the search for a suitable approach for premise selection that models the hidden relations in the output. Chapter 3 proposes a new approach that satisfies this requirement, and experimentally shows its advantage compared to the alternative methods treating premises as independent entities.

- **Can proving be cast as a next-word prediction task?** A proof can be seen as consisting of inter-connected *proof steps* that modify an implicit *proof state*. A question arises whether there is a machine learning approach that could naturally model the evolving proof state and predict (or *decode*) promising proof steps conditioned on the state. In Chapter 4, we identify *connection tableau* and *recurrent neural networks* as a proving calculus and a machine learning method, respectively, that can embody a solution to such a framing of the problem. Connection tableau's proofs have the form of trees of literals, and they are constructed by applying a limited number of available proof steps. RNNs, on the other hand, implement an evolving hidden vector state that can naturally model growing branches of the tableau trees.

- **How can online learning be applied to advising in proof assistants?**

  Providing machine-learned advice to the users of proof assistants is an attractive and useful research goal. However, it induces unique challenges from the perspective of machine learning algorithms. When a user develops a piece of mathematical theory, it is likely that lemmas and definitions that directly precede a theorem being proved (and likely appear in the same file) will be useful for completing the proof. To incorporate the recent lemmas and definitions in machine-learned advisor, an *online* learning algorithm is needed, i.e., an algorithm that can effectively learn on examples digesting them *one by one*, as opposed to learning from a large, *static dataset*. In Chapters 6 and 7, we develop a novel, custom version of an online random forest algorithm capable of efficient learning from sparse features, and tightly integrated with a host proof assistant (Coq and Lean, respectively).

- **Can machine learning techniques realistically improve the performance of a state-of-the-art SMT solver?**

  Machine learning has been applied as internal guidance in various first-order ATPs (like saturation-based [35, 72, 74] or connection [81, 83, 162] provers) and even in highly engineered SAT solvers [69]. However, its applicability to improve the performance of SMT solvers remained largely unexplored. In Chapter 5, we picked a challenging and important subdomain of SMT – solving *problems with quantifiers* – and successfully improved the performance for some families of problems using a carefully engineered ML-based approach applied in place of a solid heuristic.

- **Can neural networks be trained to perform symbolic rewriting?**

  Neural language models have emerged as the best available approach for dealing with a variety of tasks related to natural language. However, the applicability of these architectures to reasoning-related tasks using symbolic, formal languages was mostly unexplored. Intuitively, the results of applying neural language models in such a strict, formal setting should not give great results given the very different nature of symbolic languages, which do not admit the flexibility or indeterminateness of natural languages. Yet, in Chapter 8, we show that neural language models with relatively light training can achieve good accuracy for rewriting symbolic expressions. This was one of the first experiments on applying neural language models to symbolic tasks.

Chapter 2

ATPboost: Learning premise selection in binary setting with ATP feedback

Abstract

ATPboost is a system for solving sets of large-theory problems by interleaving ATP runs with state-of-the-art machine learning of premise selection from the proofs. Unlike many approaches that use multilabel setting, the learning is implemented as binary classification that estimates the pairwise-relevance of (*theorem, premise*) pairs. ATPboost uses for this the fast state-of-the-art XGBoost gradient boosting algorithm. Learning in the binary setting, however, requires negative examples, which is nontrivial due to many alternative proofs. We discuss and implement several solutions in the context of the ATP/ML feedback loop, and show substantial improvement over the multilabel approach.

2.1 Introduction: Machine learning for premise selection

Assume that c is a conjecture which is a logical consequence of a large set of premises[1] P. The chance of finding a proof of c by an automated theorem prover (ATP) often depends on choosing a small subset of P relevant for proving c. This is known as the *premise selection* task [1]. This task is crucial to make ATPs usable for proof automation over large formal corpora created with systems such as Mizar, Isabelle, HOL, and Coq [22]. Good methods for premise selection typically also transfer to related tasks, such as *internal proof guidance* of ATPs [74, 81, 101, 162] and *tactical guidance* of ITPs [56].

The most efficient premise selection methods use *data-driven* or *machine-learning* approaches. Such methods work as follows. Let T be a set of theorems with their proofs. Let C be a set of conjectures without proofs, each associated with a set of available premises that can be used to prove them. We want to learn a (statistical) model from T, which for each conjecture $c \in C$ will rank its available premises according to their relevance for producing an ATP proof of c. Two different machine learning settings can be used for this task:

1. *multilabel classification*: we treat premises used in the proofs as opaque labels and we create a model capable of labeling conjectures based on their features,

2. *binary classification*: here the aim of the learning model is to recognize pairwise-relevance of the (*conjecture, premise*) pairs, i.e. to decide what is the chance of a premise being relevant for proving the conjecture based on the features of both the conjecture and the premise.

Most of the machine learning methods for premise selection have so far used the first setting [21, 80, 82]. This includes fast and robust machine learning algorithms such as *naive Bayes* and *k-nearest neighbors* (k-NN) capable of multilabel classification with many examples and labels. This is needed for large formal libraries with many facts and proofs. There are, however, several reasons why the second approach may be better:

1. Generality: in binary classification it is possible to estimate the relevance of (*conjecture, premise*) pairs where the premise was so far unseen (i.e., not in the training data).

[1] By a *premise* we mean either a theorem, an axiom, or a definition. The term *premise* is used here interchangeably with *fact*.

2. State-of-the-art ML algorithms are often capable of learning subtle aspects of complicated problems based on the features. The multilabel approach trades the rich feature representation of the premise for its opaque label.

3. Many state-of-the-art ML algorithms are binary classifiers or they struggle when performing multilabel classification for a large number of labels.

Recently, substantial work [71] has been done in the binary setting. In particular, applying deep learning to premise selection has improved state of the art in the field. There are, however, modern and efficient learning algorithms such as XGBoost [33] that are much less computationally-intensive then deep learning methods. Also, obtaining negative examples for training the binary classifiers is a very interesting problem in the context of many alternative ATP proofs and a feedback loop between the ATP and the learning system.

2.1.1 Premise selection in binary setting with multiple proofs

The existence of multiple ATP proofs makes premise selection different from conventional machine learning applications. This is evident especially in the binary classification setting. The ML algorithms for recognizing pairwise relevance of (*conjecture, premise*) pairs require good data consisting of two (typically balanced) classes of positive and negative examples. But there is no conventional way how to construct such data in our domain. For every true conjecture (in typical proof systems) there are infinitely many formal proofs. The ATP proofs are often based on many different sets of premises. The notions of *useful* or *superfluous premise* are only approximations of their counterparts defined for sets of premises.

As an example, consider the following frequent situation: a conjecture c can be ATP-proved with two sets of axioms: $\{p_1, p_2\}$ and $\{p_3, p_4, p_5\}$. Learning only from one of the sets as positives and presenting the other as negative (*conjecture, premise*) pairs may considerably distort the learned notion of a *useful premise*. This differs from the multilabel setting, where negative data are typically not used by the fast ML algorithms such as naive Bayes and k-NN. They just aggregate different positive examples into the final ranking.

Therefore, to further improve the premise selection algorithms it seems useful to consider learning from multiple proofs and to develop methods producing good negative data. The most suitable way how to do that is to allow multiple interactions of the machine learner with the ATP system. In the following section we present the ATPboost system, which implements several such algorithms.

2.2 ATPboost: Setting, algorithms and components

ATPboost[2] is a system for solving sets of large-theory problems by interleaving ATP runs with learning of premise selection from the proofs using the state-of-the-art XGBoost algorithm. The system implements several algorithms and consists of several components described in the following sections. Its setting is a large theory $\mathcal{T}$, extracted from a large ITP library where facts appear in a chronological order. In more detail, we assume the following inputs and notation:

1. T – names of theorems (and problems) in a large theory $\mathcal{T}$.
2. P – names of all facts (premises) in $\mathcal{T}$. We require $P \supseteq T$.
3. Statements_P of all $p \in P$ in the TPTP format [150].
4. Features_P – characterizing each $p \in P$. Here we use the same features as in [82] and write $\boldsymbol{f}_p$ for the (sparse) vector of features of p.
5. Order_P ($<_P$) – total order on P; p may be used to prove t iff $p <_P t$. We write A_t for $\{p : p <_P t\}$, i.e. the set of premises allowed for t.
6. $\text{Proofs}_{T'}$ for a subset $T' \subseteq T$. Each $t \in T'$ may have many proofs $\mathcal{P}_t$. P_t denotes the premises needed for at least one proof in $\mathcal{P}_t$.

2.2.1 Algorithms

We first give a high-level overview and pseudocode of the algorithms implemented in ATPboost. Subsection 2.2.2 then describes the used components in detail.

Algorithm 1 is the simplest setting. Problems are split into the train/test sets, XGBoost learns from the training proofs, and its predictions are ATP-evaluated on the test set. This is used mainly for hyper-parameter optimization.

Algorithm 2 evaluates the trained XGBoost also on the training part, possibly finding new proofs that are used to update the training data for the next iteration. The test problems and proofs are never used for training. Negative mining may be used to find the worst misclassified premises and to correspondingly update the training data in the next iteration.

[2]The Python package is available at `https://github.com/BartoszPiotrowski/ATPboost`.

Algorithm 3 begins with no training set, starting with ATP runs on random rankings. XGBoost is trained on the ATP proofs from the previous iteration, producing new ranking for all problems for the next iteration. This is a MaLARea-style [161] feedback loop between the ATP and the learner.

2.2.2 Components

Below we describe the main components of the ATPboost algorithms and the main ideas behind them. As discussed in Section 2.1, they take into account the binary learning setting, and in particular implement the need to teach the system about multiple proofs by proper choice of examples, continuous interaction with the ATP and intelligent processing of its feedback. The components are available as procedures in our Python package.

Algorithm 1 Simple training/test split.

Require: Set of theorems T, set of premises $P \supseteq T$, $\textsc{Proofs}_T$, $\textsc{Features}_P$, $\textsc{Statements}_P$, $\textsc{Order}_P$, $\textsc{params}_{\text{set}}$, $\textsc{params}_{\text{model}}$.

1: $T_{\text{train}}, T_{\text{test}} \leftarrow \textsc{RandomlySplit}(T)$
2: $\mathcal{D} \leftarrow \textsc{TrainingSet}(\textsc{Proofs}_{T_{\text{train}}}, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{set}})$
3: $\mathcal{M} \leftarrow \textsc{TrainModel}(\mathcal{D}, \textsc{params}_{\text{model}})$
4: $\mathcal{R} \leftarrow \textsc{Rankings}(T_{\text{test}}, \mathcal{M}, \textsc{Features}_P, \textsc{Order}_P)$
5: $\mathcal{P} \leftarrow \textsc{ATPevaluation}(\mathcal{R}, \textsc{Statements}_P)$

$\textsc{TrainingSet}(\textsc{Proofs}_T, \textsc{Features}_P, \textsc{Order}_P, \textsc{params})$ This procedure constructs a $\textsc{TrainingSet}$ for a binary learning algorithm. This is a sparse matrix of positive/negative examples and a corresponding vector of binary labels. The examples (matrix rows) are created from $\textsc{Proofs}_T$ and $\textsc{Features}_P$, respecting $\textsc{Order}_P$. Each example is a concatenation of $\boldsymbol{f}_t$ and $\boldsymbol{f}_p$, i.e., the features of a theorem t and a premise p. Positive examples express that p is relevant for proving t, whereas the negatives mean the opposite.

The default method (simple) creates positives from all pairs (t, p) where $p \in P_t$. Another method (short) creates positives only from the *short* proofs of t. These are the proofs of t with at most $m+1$ premises, where m is the minimal number of premises used in a proof from $\mathcal{P}_t$. Negative examples for theorem t are chosen randomly from pairs (t, p) where $p \in A_t \setminus P_t$. The number of such randomly chosen pairs is $\textsc{ratio} \cdot N_{\text{pos}}$, where N_{pos} is the number of positives and $\textsc{ratio} \in \mathbb{N}$ is a hyper-parameter that needs to be optimized experimentally.

Algorithm 2 Incremental feedback loop with training/test split.

Require: Set of theorems T, set of premises $P \supseteq T$, $\textsc{Features}_P$, $\textsc{Statements}_P$, $\textsc{Proofs}_T$, $\textsc{Order}_P$, $\textsc{params}_{\text{set}}$, $\textsc{params}_{\text{model}}$, $\textsc{params}_{\text{negmin}}$ (optionally).

$T_{\text{train}}, T_{\text{test}} \leftarrow \textsc{RandomlySplit}(T)$
$\mathcal{D} \leftarrow \textsc{TrainingSet}(\textsc{Proofs}_{T_{\text{train}}}, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{set}})$
repeat
  $\mathcal{M} \leftarrow \textsc{TrainModel}(\mathcal{D}, \textsc{params}_{\text{model}})$
  $\mathcal{R}_{\text{train}} \leftarrow \textsc{Rankings}(T_{\text{train}}, \mathcal{M}, \textsc{Features}_P, \textsc{Order}_P)$
  $\mathcal{R}_{\text{test}} \leftarrow \textsc{Rankings}(T_{\text{test}}, \mathcal{M}, \textsc{Features}_P, \textsc{Order}_P)$
  $\mathcal{P}_{\text{train}} \leftarrow \textsc{ATPevaluation}(\mathcal{R}_{\text{train}}, \textsc{Statements}_P)$
  $\mathcal{P}_{\text{test}} \leftarrow \textsc{ATPevaluation}(\mathcal{R}_{\text{test}}, \textsc{Statements}_P)$
  $\textsc{Update}(\textsc{Proofs}_{\text{train}}, \mathcal{P}_{\text{train}})$
  $\textsc{Update}(\textsc{Proofs}_{\text{test}}, \mathcal{P}_{\text{test}})$
  if $\textsc{params}_{\text{negmin}}$ **then**
    $\mathcal{D} \leftarrow \textsc{NegMin}(\mathcal{R}, \textsc{Proofs}_{\text{train}}, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{negmin}})$
  else
    $\mathcal{D} \leftarrow \textsc{TrainingSet}(\textsc{Proofs}_{\text{train}}, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{set}})$
until Number of $\textsc{Proofs}_{\text{test}}$ not increased after $\textsc{Update}$.

Algorithm 3 Incremental feedback loop starting with no proofs.

Require: Set of theorems T, set of premises $P \supseteq T$, $\textsc{Features}_P$, $\textsc{Statements}_P$, $\textsc{Order}_P$, $\textsc{params}_{\text{set}}$, $\textsc{params}_{\text{model}}$, $\textsc{params}_{\text{negmin}}$ (optionally).

$\textsc{Proofs}_T \leftarrow \emptyset$
$\mathcal{R} \leftarrow \textsc{RandomRankings}(T)$
$\mathcal{P} \leftarrow \textsc{ATPevaluation}(\mathcal{R}, \textsc{Statements}_P)$
$\textsc{Update}(\textsc{Proofs}_T, \mathcal{P})$
$\mathcal{D} \leftarrow \textsc{TrainingSet}(\textsc{Proofs}_T, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{set}})$
repeat
  $\mathcal{M} \leftarrow \textsc{TrainModel}(\mathcal{D}, \textsc{params}_{\text{model}})$
  $\mathcal{R} \leftarrow \textsc{Rankings}(T, \mathcal{M}, \textsc{Features}_P, \textsc{Order}_P)$
  $\mathcal{P} \leftarrow \textsc{ATPevaluation}(\mathcal{R}, \textsc{Statements}_P)$
  $\textsc{Update}(\textsc{Proofs}_T, \mathcal{P})$
  if $\textsc{params}_{\text{negmin}}$ **then**
    $\mathcal{D} \leftarrow \textsc{NegMin}(\mathcal{R}, \textsc{Proofs}_T, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{negmin}})$
  else
    $\mathcal{D} \leftarrow \textsc{TrainingSet}(\textsc{Proofs}_T, \textsc{Features}_P, \textsc{Order}_P, \textsc{params}_{\text{set}})$
until Number of $\textsc{Proofs}_T$ not increased after $\textsc{Update}$.

Since $|A_t \setminus P_t|$ is usually much larger than $|P_t|$, it seems reasonable to have a large RATIO. This, however, increases class imbalance and the probability of presenting to the learning algorithm a *false negative*. This is a pair (t, p) where $p \notin P_t$, but there is an ATP proof of t using p that is not yet in our dataset.

TRAINMODEL(TRAININGSET, PARAMS) This procedure trains a binary learning classifier on the TRAININGSET, creating a MODEL. We use XGBoost [33] – a state-of-the-art tree-based gradient boosting algorithm performing very well in machine learning competitions. It is also much faster to train compared to deep learning methods, performs well with unbalanced training sets, and is optimized for working with sparse data. XGBoost has several important hyper-parameters, such as NUMBEROFTREES, MAXDEPTH (of trees) and ETA (learning rate). These hyper-parameters have significant influence on the performance and require tuning.

RANKINGS(C, MODEL, FEATURES$_P$, ORDER$_P$) This procedure uses a trained MODEL to construct RANKINGS$_C$ of premises from P for conjectures $c \in C \subseteq T$. Each conjecture c is paired with each premise $p <_P c$ and concatenations of $\boldsymbol{f}_c$ and $\boldsymbol{f}_p$ are passed to the MODEL. The MODEL outputs a real number in $[0, 1]$, which is interpreted as the relevance of p for proving c. The relevances are then used to sort the premises into RANKINGS$_C$.

ATPEVALUATION(RANKINGS, STATEMENTS) Any ATP can be used for evaluation. By default we use E [144].[3] As usual, we construct the ATP problems for several top slices (lengths $1, 2, 4, \ldots, 512$) of the RANKINGS. To remove redundant premises we *pseudo-minimize* the proofs: only the premises needed in the proofs are used as axioms and the ATP is rerun until a fixpoint is reached.

UPDATE(OLDPROOFS, NEWPROOFS) The UPDATE makes a union of the new and old proofs, followed by a subsumption reduction. I.e., if premises of two proofs of t are in a superset relation, the proof with the larger set is removed.

NEGMIN(PROOFS$_T$, RANKINGS$_T$, FEATURES$_P$, ORDER$_P$, PARAMS) NEGMIN stands for *negative mining*. This procedure is used as a more advanced alternative to TRAININGSET. It examines the last RANKINGS$_T$ for the most *misclassified positives*. I.e., for each $t \in T$ we create a set MP_t of those p that were

[3] The default time limit is 10 seconds and the memory limit is 2GB. The exact default command is: `./eprover --auto-schedule --free-numbers -s -R --cpu-limit=10 --memory-limit=2000 --print-statistics -p --tstp-format problem_file`

previously ranked high for t, but no ATP proof of t was using p. We define three variants:

1. NEGMIN_ALL: Let m_t be the maximum rank of a t-useful premise ($p \in P_t$) in $\text{RANKINGS}_T[t]$. Then $MP_t^1 = \{p : rank_t(p) < m_t \wedge p \notin P_t\}$.
2. NEGMIN_RAND: We randomly choose into MP_t^2 only a half of MP_t^1.
3. NEGMIN_1: $MP_t^3 = \{p : rank_t(p) < |P_t| \wedge p \notin P_t\}$.

The set MP_t^i is then added as negatives to the examples produced by the TRAININGSET procedure. The idea of such negative mining is that the learner takes into account the mistakes it made in the previous iteration.

2.3 Evaluation

We evaluate[4] the algorithms on a set of 1342 MPTP2078 [1] large (*chainy*) problems that are provable in 60 s using their small (*bushy*) versions.

2.3.1 Parameter tuning

First we run Algorithm 1 to optimize the hyper-parameters. The dataset was randomly split into a training set of 1000 problems and a test set of 342. For the training set, we use the proofs obtained by the 60 s run on the bushy versions. We tune the RATIO hyper-parameter of TRAININGSET, and the NUMBEROFTREES, MAXDEPTH and ETA hyper-parameters of TRAINMODEL. Due to resource constraints we *a priori* assume reasonable defaults: RATIO = 16, NUMBEROFTREES = 2000, MAXDEPTH = 10, ETA = 0.2. Then we observe how changing each parameter separately influences the results. Table 2.1 shows the ATP results for the RATIO hyper-parameter, and Figure 2.1 for the model hyper-parameters.

It is clear that a high number of negatives is important. Using RATIO = 16 proves 6% more test problems than the balanced setting (RATIO = 1). It is also clear that a higher number of trees – at least 500 – improves the results. However, too many trees (over 8000) slightly decrease the performance, likely due to over-fitting. The ETA hyper-parameter gives best results with values between 0.04 and 0.64, and the MAXDEPTH of trees should be around 10.

[4] All the scripts we used for the evaluation are available at `https://github.com/BartoszPiotrowski/ATPboost/tree/master/experiments`

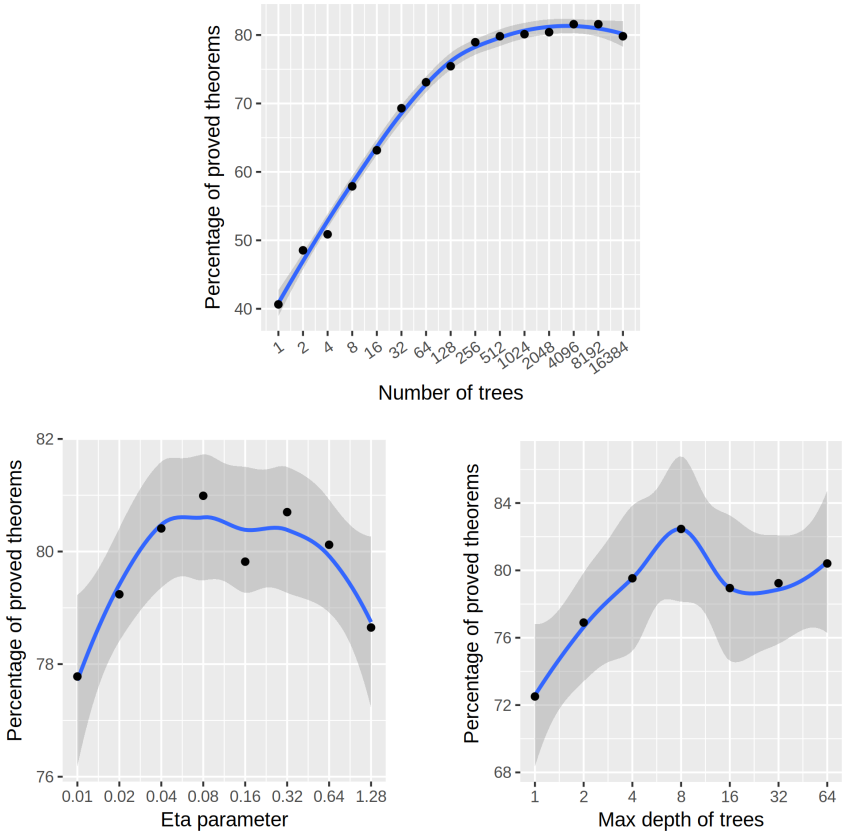


Figure 2.1: ATP performance for different values of hyper-parameters of the XGBoost model. *Number of trees* shows a clear trend: up to the value of 256 increasing this parameter substantially improves the performance; above this point, we observe diminishing returns, which is an expected behaviour of the XGBoost algorithm. For *eta parameter*, which is also known as *learning rate* and intuitively governs the speed of learning, the trend is less obvious, but the optimal values appear in the middle range of the tested values (between 0.04 and 0.64). *Max depth of trees* influences the complexity of the model; for lower values, we observe the phenomenon of *under-fitting* (the model is too simple to represent the learned concept), whereas the higher values result in *over-fitting* (the model picks up spurious correlations in the training data). The optimal depth of trees is 8; higher values (16, 32, and 64) result in a similar performance.

Table 2.1: Influence of the RATIO of randomly generated negatives to positives. It is visible that low ratio of negatives to positives is not beneficial.

RATIO	1	2	4	8	16	32	64
proved (%)	74.0	78.4	79.0	78.7	80.1	79.8	80.1

We evaluate Algorithm 1 also on a much bigger ATP-provable part of MML with 29 271 theorems in train part and 3253 in test. With hyper-parameters RATIO = 20, NUMBEROFTREES = 4000, MAXDEPTH = 10 and ETA = 0.2 we proved 58.78% theorems (1912). This is a 15.7% improvement over k-NN, which proved 50.81% (1653) theorems. For a comparison, the improvement over k-NN obtained (with much higher ATP timeouts) with deep learning in [71] was 4.3%.

2.3.2 Incremental feedback loop with train/test split

This experiment evaluates Algorithm 2, testing different methods of negative mining. The train/test split and the values of the hyper-parameters RATIO, NUMBEROFTREES, MAXDEPTH, ETA are taken from the previous experiment. We test six methods in parallel. Two XGB methods (SIMPLE and SHORT) are the variants of the TRAININGSET procedure, three XGB methods (NEGMIN_ALL, NEGMIN_RAND and NEGMIN_1) are the variants of NEGMIN, and the last one is a k-NN learner similar to the one from [82], used here for comparison. The experiment starts with the same proofs for training theorems as in the previous one, and we performed 30 rounds of the feedback loop. Figure 2.2 shows the results.

All the new methods largely outperform k-NN. XGB_SHORT is much better than XGB_SIMPLE, which means that using positives from too many proofs seem harmful, as in [95] where this was observed with k-NN. The differences between the XGB variants SHORT, NEGMIN_1, NEGMIN_ALL, and NEGMIN_RAND do not seem significant and all perform well. At the end of the loop (30th round) 315–319 theorems of the 342 (ca. 93%) are proved.

2.3.3 Incremental feedback loop with no initial proofs

The final experiment corresponds to Algorithm 3. There is no train/test split and no initial proofs. The first ATP evaluation is done on random rankings, proving 335 theorems out of the 1342. Then the loop starts running with the same options as in the previous experiment. Figure 2.3 shows the numbers of

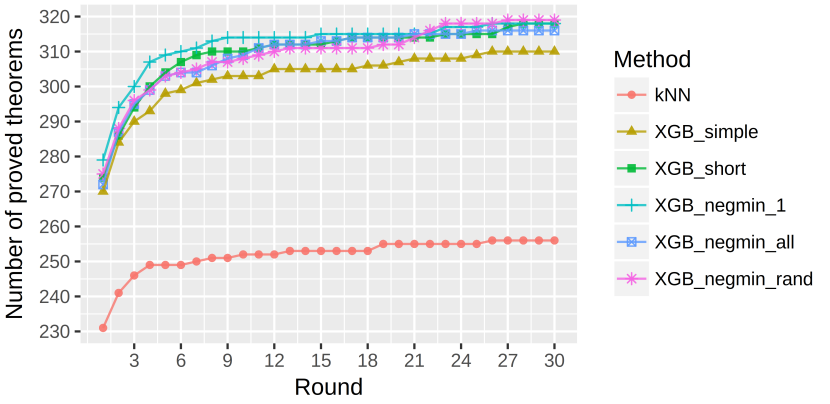


Figure 2.2: Number of proved theorems (cumulatively) in subsequent iterations of Algorithm 2. After fast increase in the first few iterations we observe diminishing returns. All the XGBoost-based methods give substantially better performance than the baseline k-NN approach. Using positive examples only from the shortest proofs (XGB_SHORT) is better than using all collected positives (XGB_SIMPLE). There are no significant differences between the variants of negative mining.

theorems that were proved in the subsequent rounds, as well as the growth of the total number of different proofs. This is important, because all these proofs are taken into account by the machine learning. Again, k-NN is the weakest and XGB_SIMPLE is worse than the rest of the methods, which are statistically indistinguishable. In the last round XGB_NEGMIN_RAND proves 1150 (86%) theorems. This is a 26.8% improvement over k-NN (907) and 7.7% more than XGB_SIMPLE (1068).

2.4 Conclusions and future work

In this work we distinguished between *multilabel* and *binary* machine-learning approaches to premise selection. The latter is more complex as it requires dealing with negative examples, which are not necessary in the former approach.

Several strategies were shown for creating the training negative examples based on collected proofs. This included the approach where highly misclassified positives were more likely to be used as negatives (*negative mining*).

The evaluation on the MPTP2078 showed that a multilabel method using the k-NN algorithm was inferior to a binary method using the XGBoost algorithm (across all tested strategies for creating negative and positive examples).

The results showed that using as positive training examples all premises used in at least one of the proofs of a given theorem is worse than focusing only on premises used in short proofs only. However, the results did not allow to draw a conclusion what is the best strategy for creating negative training examples. The answer to this may be a goal of the future work.

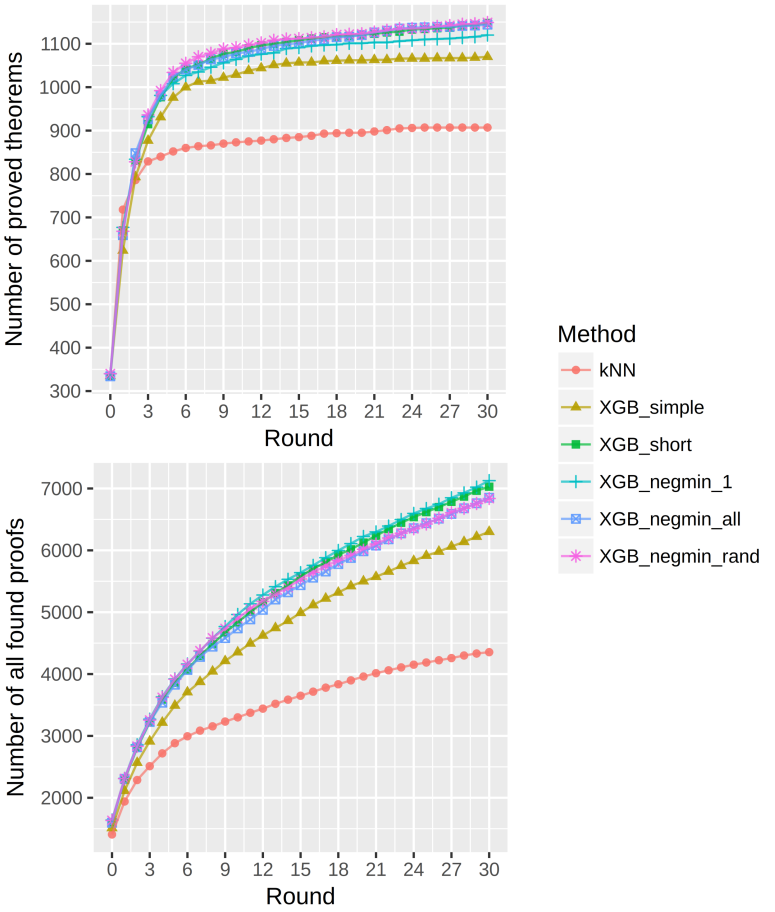


Figure 2.3: Proved theorems (top) and number of all found proofs (bottom) in subsequent rounds of the experiment corresponding to Algorithm 3. The various methods seem to perform similarly as in Figure 2.2 in relation to each other. Interestingly, the growth of the number of different *proofs* is substantially steeper than the growth of the number of *theorems* proved (as each theorem can have multiple proofs). This means that the training set is still quickly growing, which gives a reason to expect more new theorems proved if more iterations of the loop were run.

Chapter 3

Stateful premise selection by recurrent neural networks

Abstract

In this work we develop a new learning-based method for selecting facts (premises) when proving new goals over large formal libraries. Unlike previous methods that choose sets of facts independently of each other by their rank, the new method uses the notion of *state* that is updated each time a choice of a fact is made. Our stateful architecture is based on recurrent neural networks which have been recently very successful in stateful tasks such as language translation. The new method is combined with data augmentation techniques, evaluated in several ways on a standard large-theory benchmark and compared to state-of-the-art premise approach based on gradient boosted trees. It is shown to perform substantially better and to solve many new problems.

3.1 Introduction: Premise selection over large libraries

Premise selection [1] is a critical task in automated theorem proving (ATP) over large theories where typically only a small fraction of the available facts

are relevant for proving a new conjecture. One of the main applications is in ITP/ATP *hammers* [22] that assist ITP users to automatically discharge proof obligations in large formalizations. Several heuristic approaches to premise selection such as SiNE [68] and MePo [106], as well as learning-based methods using hand-designed features have been developed so far. The latter include naive Bayes [159], kernel methods [156], k-nearest neighbors [80,82] and gradient boosted trees [127]. This has been followed by neural architectures that learn the features on their own [71, 94].

The learning-based premise selection methods have been so far based on the same paradigm of *ranking* the available facts (premises) *independently* with respect to the conjecture that is being proved. The highest ranked facts are then used together as axioms and given to the ATP systems together with the conjecture. This approach, although useful and reasonably successful, does not take into account an important aspect of the premise selection problem: premises are *not* independent of each other. There are important logical relations among them. Some premises complement each other better when proving a particular conjecture, while some highly-ranked premises might be just minor variants of one another.

In this work, we first (Section 3.2) propose the recurrent neural network (RNN) encoder-decoder model [34] as a suitable stateful approach for premise selection and we describe the RNN architecture we have chosen for this task. In Section 3.3 we develop several data augmentation methods that help training the RNNs for the premise selection task. Section 3.4 describes the experimental evaluation and Section 3.5 discusses the results.

3.2 Premise selection and neural machine translation

In recent years, powerful methods for learning sequences of conditional stateful decisions have been developed in machine translation of the natural languages. In *neural machine translation* (NMT) [34] the source sentence is encoded as a hidden vector representation by the *encoder* and the translated target sentence is produced word-by-word by the *decoder*. Each translated word is conditioned not only on the source sentence but also on the *previously decoded words*. Words and phrases in natural languages are related in many ways and such relations have to be taken into account for the produced sequence of words to be a sensible, grammatically correct sentence. Successful NMT methods using recurrent encoder-decoder architectures [153] are explicitly based on a notion of a learned

hidden state that is updated with each produced word. This corresponds to our requirement of *stateful premise selection*: We want to have a (learned) hidden state after selecting a particular fact with particular mathematical content that should not be repeated in the next facts but rather suitably complemented by them to justify the conjecture.

Another aspect of neural machine translation that is relevant in premise selection is the *multiplicity of correct outputs*. In translation, there are often multiple correct translations of a given sentence that deliver its *meaning* (perhaps more or less clumsily). Similarly, in mathematics there is typically no single, golden proof of a conjecture. Often there are many different proofs that use various sets of premises and various sequences of inferences. NMT methods accommodate the multiplicity of possible outputs – typically by using *beam search* [52]. Such mechanisms seem directly usable also in premise selection and proof search. NMT systems have already been successfully applied in autoformalization and symbolic settings [98, 131, 167].

Our recurrent neural architecture There are various state-of-the-art neural sequence-to-sequence architectures that can be applied for modelling premise selection. Although ultimately a custom architecture could be designed to capture all aspects of this task, our initial choice was to experiment with an existing established implementation of neural machine translation system. We have chosen the OpenNMT toolkit [90]. It implements the LSTM [67] recurrent cells and several state-of-the-art techniques, including the attention mechanisms [103] and beam search [52]. It has proven to be very good in natural language translation and related tasks [90].

We have decided to use the default parameters for training OpenNMT on our tasks – in this work we mainly investigate the influence of various forms of training data (Section 3.3) on the predictive performance. The more important values of the OpenNMT hyper-parameters chosen by us are as follows: the number of training epochs: 100 000, the size of encoder's and decoder's LSTM cells: 2 layers of 500 units, word embedding size: 500. Additionally, we have used the attention mechanism by Luong [103].

3.3 Data, their representation and augmentation

A recurrent NMT system is trained on pairs of *source* and *target* sequences. In our case, the source is a statement of a theorem and the target is a list of names of its premises. There are multiple ways how to transform ATP proofs

to training examples of such form, and it is not clear which way is the best for training the RNN. In this section, we describe several methods of constructing the training examples[1]. This includes the following topics: (i) representation of the conjecture as a sequence (Subsection 3.3.2), (ii) ordering of the premises into a sequence (Subsection 3.3.3), (iii) using subproof data for augmenting the training data (Subsection 3.3.4), and (iv) oversampling of rare premises (Subsection 3.3.5).

3.3.1 Initial data for training RNNs

The experimental data originate from the Mizar Mathematical Library [60] (MML) translated [159] to the TPTP language [151]. We use the MPTP2078 benchmark [1] – a subset of 2078 Mizar theorems. Using the ATPboost [127] system we have initially proved as many of the MPTP2078 problems as possible, recording each distinct proof. ATPboost in turn relies on the E prover [144] and the XGBoost machine learning system [33] using gradient boosted trees for premise selection. 24 087 different proofs of 1469 theorems were found in total. The number of different proofs per theorem ranged between 1 and 265 (on average 16.4). The proofs used in total 2227 different premises. Each proof used between 1 and 50 premises (on average 11.5). Each proof determines a pair $(t, \{p_1, p_2, \ldots, p_n\})$, where the first element t is the proved theorem and the second element is the set of premises p_i used in the proof. These pairs constitute examples for training a machine learning model to propose useful premises for theorems. The 1469 theorems that have an ATP proof were randomly split in proportions 0.75 and 0.25 into training and testing parts. The 1100 training theorems with their proofs resulted in 18 361 training pairs. From the set of remaining 369 theorems we filtered out those which contained in all their proofs premises not appearing in the training set. This yields our testing set of 310 theorems.

3.3.2 Representation of the statements

The simplest way of constructing the source sequences of the examples is just using tokenized statements in standard TPTP syntax. We label this type of source as `standard`. The tokenized TPTP statements can also be transformed to other formats. A suitable one is the Polish prefix notation (labeled as `prefix`) as shown in Table 3.1. The motivation is that this format is more compact as

[1] All the data along with scripts allowing reproduction of the experiments are available at: `https://github.com/BartoszPiotrowski/stateful-premise-selection-with-RNNs`

Table 3.1: An example of a Mizar statement translated to TPTP, tokenized (`standard`), and additionally transformed to prefix notation (`prefix`).

Mizar:	`for A being set st A is empty holds A is finite`
TPTP:	`![A] : (v1_xboole_0(A) => v1_finset_1(A))`
`standard`:	`! [ A ] : ( v1_xboole_0 ( A ) => v1_finset_1 ( A ) )`
`prefix`:	`! A => v1_xboole_0 A v1_finset_1 A`

the formulas do not contain brackets and commas. In our case, the average length of a `standard` source training sequence is 81, whereas for `prefix` it is only 39. This may be useful for NMT architectures that suffer when using long input (and output) sequences [34]. Related work using NMT in symbolic setting reports improvements when using prefix notation [167].

3.3.3 Ordering of the premises

In the abstract premise selection task the order of the premises is not constrained in any way. In practice, ATP systems may be influenced by the order of the premises given in the input. More importantly, existing efficient learning methods that are capable of capturing dependencies among the elements of sequences (such as RNNs, used this work) are sensitive to the order of the elements (premises in our case). Preferably, we want to train on examples that illustrate dependencies between the premises. On the other hand, we do not want to rely too much on a particular order of premises in the target sequence. We propose several possible approaches.

Permutations As a baseline approach we permute the target premises randomly, thus not passing to the recurrent neural model any additional information about the order of the premises. We either produce only one permutation (`permuted`) or 100 of them (`permuted_100`).

Permutations preserving the proof tree Each proof produced by a refutational prover (such as E) is a tree (more precisely, a DAG), with `FALSE` in its root and the premises and the conjecture in its leaves. See Figure 3.1 for an example. We produce a compacted version of the proof by removing all intermediate nodes that have only one parent[2] (right side of Figure 3.1). The

[2]This terminology is used in such a way that children are derived from their parents.

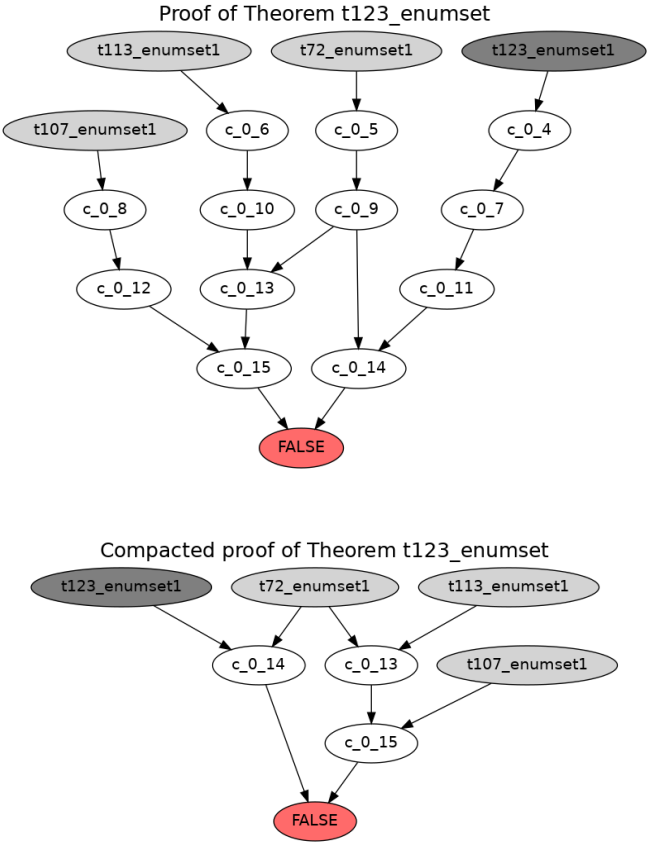


Figure 3.1: Trees representing a refutational proof of Mizar theorem `t123_enumset` (top) and its compacted version (bottom). Light-grey nodes are the input premises and a dark-grey node represents the (negated) conjecture. Nodes prefixed by `c_0` are intermediate lemmas.

premises t72_enumset1 and t113_enumset1 there interact directly resulting in an intermediate lemma c_0_13. This lemma subsequently interacts with t107_enumset1. Putting t72_enumset1 and t113_enumset1 closer together than t107_enumset1 and t72_enumset1 tells the learner how much the premises interact with each other. This idea is implemented in the following way. Each tree induces its *nested list representation*, which we define recursively in the following way: (i) list representation of a tree rooted in a non-leaf node n is a list of list representations of parents of n, (ii) list representation of a leaf node is its label. For instance, `[[t123, t72], [[t72, t113], t107]]` is a list representation of the tree from Figure 3.1. Each tree has many list representations depending on how the elements of the lists are ordered.

We say that a sequence s of labels of the leaves *respects the tree* if s resulted from the flattening of a list representation of the tree. (In the example shown in Figure 3.1 the sequence (t123, t72, t72, t113, t107) respects the tree but the sequence (t123, t107, t72, t113, t72) does not.) Such sequences have the property of keeping closer the premises that interacted closer. Note that the sequences may contain repetitions, as in the example above, and each tree has many sequences respecting it. For each proof tree we remove from its sequences the conjecture. We take either only one such sequence for each proof or (up to) 100 different sequences, which yields the permuted_tree and permuted_tree_100 sets of training examples.

ATP induced order We also experiment with using the proofs as linearized by the E prover. For a given E proof $\mathcal{P}$ we first order its internal lemmas: lemma_1 $<_L$ lemma_2 iff lemma_1 appears in $\mathcal{P}$ (linearized by E) before lemma_2. Then we define a non-strict ordering of the premises of $\mathcal{P}$: $p_1 \leq_P p_2$ iff the $<_L$-minimal child of p_1 is smaller than the $<_L$-minimal child of p_2, where both children are taken from the *compacted* tree of $\mathcal{P}$. For our example proof of t123_enumset we have: c_0_13 $<_L$ c_0_14 $<_L$ c_0_15. Hence t72 $\leq_P$ t107 (since $\min_L(\{$c_0_14, c_0_13$\})$ $<_L$ c_0_15), and t113 $=_P$ t107 (since $\min_L(\{$c_0_14, c_0_13$\})$ = c_0_13). We break the ties randomly. Different E proofs of one theorem may result in different premise orderings. This way of ordering premises in the target of the examples is labeled as order_from_proof.

3.3.4 Augmentation with subproof data

We can also augment the training data by extracting many subproofs from the original training proofs. For this, we use the intermediate lemmas from the compacted representations of proofs that are not derived from the negated

conjecture. The pairs $(l, \{p_1, p_2, \ldots, p_n\})$ where p_i are all premise ancestors of lemma l constitute additional examples that can be used for augmenting the main training data. From the subproofs of the training theorems we extracted 46 094 such different training pairs. The data set that includes these examples together with the main ones is marked as `augmented`. The experiments with sublemmas only are described in Section 3.6.

3.3.5 Oversampling rare examples

Some premises appear frequently in the training examples while some are rare. *Oversampling* is a general method that often improves the performance of neural architectures on imbalanced data [30]. We experiment with a simple oversampling scheme: training examples that contain rare premises are used multiple times. More precisely, for an example $e = (t, P) \in \mathcal{T}$, where $\mathcal{T}$ is the training set, we define the *occurrence rate* (OR) of e as:

$$\mathrm{OR}(e) = \frac{1}{|P|} \sum_{p \in P} \mathrm{OR}_{\mathrm{premise}}(p), \quad \text{where} \quad \mathrm{OR}_{\mathrm{premise}}(p) = \frac{|\{(t, P) \in \mathcal{T} \colon p \in P\}|}{\sum_{(t,P)\in\mathcal{T}} |P|}.$$

The idea is simple: OR of an example is the average $\mathrm{OR}_{\mathrm{premise}}$ of its target premises (P). $\mathrm{OR}_{\mathrm{premise}}$ measures how often a premise appears in all the targets of all the training examples.

The set of training examples $\mathcal{T}$ is split into 10 evenly sized chunks $\mathcal{T}_1, \ldots, \mathcal{T}_{10}$ according to their occurrence rate so that a higher index of $\mathcal{T}_i$ implies lower OR:

$$x \in \mathcal{T}_i \wedge y \in \mathcal{T}_j \wedge i < j \Rightarrow \mathrm{OR}(x) > \mathrm{OR}(y).$$

Each example $e \in \mathcal{T}_i$ is oversampled i times: the more rare premises an example e contains the more often e appears in the oversampled training set. This scheme was applied both to the main training set and to the augmented one (described in Subsection 3.3.4), resulting in data sets `oversampled` and `augmented_oversampled`.

3.4 Experimental evaluation

We train[3] and evaluate RNNs using the OpenNMT toolkit with its default hyper-parameters (Section 3.2) on the various premise-selection data described in Section 3.3. When evaluating on the testing sets, we use OpenNMT's beam

[3]We train on a single GeForce RTX 2080 Ti GPU. Training each model took 2–4 hours.

search with width 10 to get for each conjecture its 10 most probable sequences of premises. We want to compare the results also with state-of-the-art premise selection based on gradient boosted trees using the XGBoost toolkit. For that, we use the features and settings developed in our ATPboost system [127]. As explained in Section 3.1, the training data used for XGBoost are unordered. XGBoost produces a ranking of the premises and we use several segments of the top-ranked premises for the XGBoost evaluation. While OpenNMT needs only positive examples, XGBoost also needs negative examples. We produce them by sampling negatives randomly, which performed well in ATPboost.

To allow a meaningful comparison of the two approaches, we shorten the rankings produced by XGBoost according to the lengths of the sequences produced by OpenNMT for a given conjecture. In more detail: if $\widehat{R}$ is a ranking produced by XGBoost and $\widehat{P}_1, \widehat{P}_2, \ldots, \widehat{P}_{10}$ are sequences of premises produced by OpenNMT, we take $\widehat{R}_1, \widehat{R}_2, \ldots, \widehat{R}_{10}$ to be the top slices of the ranking $\widehat{R}$ of lengths $|\widehat{P}_1|, |\widehat{P}_2|, \ldots, |\widehat{P}_{10}|$, respectively. These 10 top slices are treated as predictions from the XGBoost system and compared with the OpenNMT predictions. We always do both the standard *ML evaluation* and the *ATP evaluation*.

ML evaluation The Jaccard index and Coverage metrics are used, defined as follows:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}, \qquad \text{Coverage}(A, B) = \frac{|A \cap B|}{|B|}.$$

Each theorem T from the test set is associated with n_T different sets of premises $P_1^T, P_2^T, \ldots, P_{n_T}^T$ that were used as axioms in its n_T known proofs and with 10 sets of premises $\widehat{P}_1^T, \widehat{P}_2^T, \ldots, \widehat{P}_{10}^T$ predicted by a given machine learning model. We estimate the quality of the predictions with $\text{Jaccard}(\bigcup_i \widehat{P}_i^T, \bigcup_i P_i^T)$ and $\text{Coverage}(\bigcup_i \widehat{P}_i^T, \bigcup_i P_i^T)$. The Jaccard metric emphasises the *precision* of the prediction: it measures how much the predicted premises intersect with the premises used in the known proofs. At the same time the score decreases when the predicted set is large. The Coverage does not penalize large predicted sets. It takes into account the fact that *true positives* are typically more important than *true negatives*. The prover may deal with some redundant axioms while the lack of relevant premises may make conjectures unprovable.

ATP evaluation The simple ML metrics may not directly translate to ATP performance. They compare unions of premises whereas an ATP is run for each premise selection separately. Additionally, during the ATP evaluation new

Table 3.2: Performance of the neural model on the test set, trained on examples with different formats of source sequences (statements) and differently ordered target sequences (premises), expressed with the similarity metrics (Jaccard index and Coverage) and with ATP success rate. `standard` format with `order_from_proof` gave the best results. However, neither of the two formats is significantly better than the other.

	source format					
	`standard`			`prefix`		
target ordering	Jaccard	Coverage	Proved	Jaccard	Coverage	Proved
`permuted`	0.18	0.39	0.25	0.20	0.36	0.23
`permuted_100`	0.09	0.14	0.10	0.18	0.26	0.19
`permuted_tree`	0.16	0.23	0.16	0.17	0.25	0.18
`permuted_tree_100`	0.04	0.05	0.03	0.11	0.15	0.09
`order_from_proof`	0.22	0.46	0.29	0.22	0.43	0.27

proofs are often discovered. Such new proofs are not taken into account by the similarity metrics (the Jaccard index and Coverage).

We perform an ATP evaluation using the E prover [144] run with a time limit of 10 s and a memory limit of 2 GB, keeping the rest of the settings in its default values. This limits the power of the prover, preventing e.g. its own axiom pruning methods such as SInE [68]. To establish a simple ATP baseline, the E prover was run for all the testing theorems with all available premises as axioms, proving 9% of the theorems. For a given machine learning method and for each testing theorem T we run 11 proof attempts: one for each of the 10 sets of predicted premises $\widehat{P}_1^T, \widehat{P}_2^T, \ldots, \widehat{P}_{10}^T$ and one with $\bigcup_i \widehat{P}_i^T$.

3.5 Results and discussion

3.5.1 Source and target combinations

First, we evaluate combinations of the statement format (`standard` or `prefix` – Subsection 3.3.2) and orderings of the target premises (Subsection 3.3.3). The results are shown in Table 3.2. The first, simplest way of ordering premises – `permuted` – performs well: in combination with the two formats of source statements it resulted in predictions with ATP success rates 25% and 23%. Many permutations of the same target sequence are bad for the NMT learner: `permuted_100` performed much worse than `permuted`. Using multiple permu-

tations of the target sequences was motivated by the fact that the order of premises should not matter in the abstract premise selection task. The recurrent neural network, however, likely sees them as *contradictory data* detrimental to its training.

The `permuted_tree` ordering was meant to reflect the distance of interaction between the premises. However, it performed much worse than `permuted`. This may be caused by many repetitions in the target sequences, which also increase their lengths, making the task for the neural decoder more difficult. These repetitions appear because of the repeated premises in the leaves of the proof trees. Similarly to `permuted_100`, adding multiple permutations (`permuted_tree_100`) is detrimental also in the case of `permuted_tree`.

The best performing way of ordering the premises for the NMT learner is to use the `order_from_proof`. This is true both for the similarity metrics and ATP performance and in combination with both formats of the conjecture. This likely means that extracting the premise ordering from the proofs brings useful and consistent information which the neural model is able to take advantage of during the training.

There is only a small difference between the ATP performance of the `standard` and `prefix` encoding of the conjecture. This is somewhat surprising in the context of related work [167], where prefix notation is useful for NMT architectures. Shorter sequences should be easier to process by the recurrent encoder. In our case, however, the different structure of the `prefix` statements seems to be reducing the benefit of conciseness.

3.5.2 Augmentation with subproof data and oversampling

Next, we use the best performing combination (from now on called `basic`) from Subsection 3.5.1 for evaluating the augmentation and oversampling methods (Subsections 3.3.4 and 3.3.5). The results are shown in Table 3.3 which also contains the XGBoost results.

Oversampling trained the learner using data with changed distribution – less frequent premises were appearing more often. This made the predictions more diverse and less precise compared to `basic`. This is reflected in the change of the similarity metrics: the Jaccard index decreased and Coverage increased. Importantly, oversampling translates to better ATP performance of the NMT predictions.

Augmentation with subproof data improved the ATP performance by a large margin of 11% points over `basic`. It means that the RNN is helped a lot by the additional subproof training data despite their slightly different origin and

Table 3.3: Performance of the NMT and XGBoost systems trained on examples augmented with subproof data and with oversampling applied. The non-modified data set is denoted as `basic`. The examples used `standard` source format and `order_from_proof` ordering of the target sequences. Both the augmentation and oversampling have substantial positive effect on the proving performance. NMT has better proving performance than XGBoost on all datasets, despite this improvement is not visible in the ML metrics. This shows that predicting smaller, more precise sets of premises is important for ATP.

	machine learning system					
	`NMT`			`XGBoost`		
training data	Jaccard	Coverage	Proved	Jaccard	Coverage	Proved
`basic`	0.22	0.46	0.29	0.26	0.56	0.25
`oversampled`	0.21	0.48	0.31	0.24	0.61	0.30
`augmented`	0.27	0.51	0.40	0.26	0.51	0.27
`augmented_oversampled`	0.26	0.47	0.39	0.25	0.51	0.31

shape (internal clauses instead of input formulas). The last row of Table 3.3 shows the result of applying oversampling on top of the augmented training set. This does not improve the NMT performance compared to `augmented`.

ATP performance of XGBoost was worse than that of NMT for all 4 data sets, and the best XGBoost ATP result (0.31) is substantially (29%) worse than the best NMT ATP result (0.40). However, XGBoost tends to show better values in similarity metrics. This can be explained by the following effects:

1. The initial data come from ATPboost – a meta-system using XGBoost. The XGBoost predictions in our experiments may be correlated with the initial testing set.

2. Even though XGBoost achieves a higher similarity between unions of the predicted premises and the premises used in the known proofs, the recurrent neural network wins with its diverse (but stateful and therefore complementary) predictions for a given conjecture. I.e., when making several ATP attempts, it seems better to use several plausible premise sets (proof ideas) that are orthogonal to one another, rather than making incremental additions to the initial set of premises. This is the effect that we wanted to achieve with RNN-based premise selection, and indeed, it makes a substantial difference. XGBoost instead just extends its single most plausible set of premises more and more according to the single

ranking of premises. It seems nontrivial to instruct XGBoost to produce multiple alternative rankings with good complementary properties in the same way as RNN does.

The NMT predictions are quite orthogonal to those from XGBoost. In all the experiments related to the results shown in Table 3.3 there were 167 theorems proved with predictions from NMT and 142 theorems proved with the use of XGBoost. The size of the intersection of these sets is 121, and there were 46 theorems that were proved with NMT but not with XGBoost.

3.6 Subproofs as standalone data set

From all the proofs in the initial data set (not only the training part) we extracted 60 299 different pairs of the form

$$\big(\texttt{lemma}, \{\texttt{premise\_1}, \texttt{premise\_2}, \ldots, \texttt{premise\_n}\}\big),$$

same as those used for augmenting the training set (Subsection 3.3.4). This means that:

- `lemma` is an intermediate sublemma appearing in the compacted representation of the proof (see Subsection 3.3.3 and Figure 3.1), which has no negated conjecture of the original proof among its ancestors,
- $\{\texttt{premise\_1}, \texttt{premise\_2}, \ldots, \texttt{premise\_n}\}$ is a set of all the premises being among ancestors of the `lemma`.

These pairs have 29 616 different lemmas (each lemma may have several different proofs). We split these lemmas into training and testing parts in proportions approximately 0.75 and 0.25, respectively (independently from the main training/testing split of the Mizar theorems). We also recorded information about the heights of the subproof trees from which the lemmas were extracted – to investigate how the height of the tree correlates with the difficulty of learning premise selection. The heights of the subtrees vary between 1 and 35, where lower subtrees are much more frequent than higher ones.

The NMT system was trained on the training examples, with the same settings as in the main experiments. Additionally, we trained the XGBoost system for comparison.

The results of the evaluation on the testing examples, in terms of the similarity metrics (Jaccard index and Coverage) as well as an ATP evaluation, are presented in Table 3.4. The table presents the performance of the machine

learning methods with respect to all the testing examples as well as on subsets of them selected according to the heights of subtrees of proof trees a given sublemma originated from.

Overall, in comparison to the main data, premise selection for subproofs turned out to be a significantly easier task for both machine learning methods. On the whole testing set the ATP performance was 83% for NMT and 61% for XGBoost. When running the automated prover without any premise selection advice, with all available premises as axioms, the ATP success rate was 13%.

As for the results depending on the heights of the subtrees, in the table we present them up to the height 9 – for larger values the number of lemmas becomes very small. There are two trends visible for both NMT and XGBoost: with increased height the Jaccard metric goes up and Coverage goes down. The likely explanation is that the lower trees contain fewer premises in their leaves and precise selection of them by the predictor is less likely, hence the low Jaccard metric. On the other hand, the higher trees have more premises in their leaves and covering them by the predictor is more difficult, which is reflected by the low Coverage. The dependence of ATP performance on the heights is unclear. Surprisingly, it is *not* the case that the smallest subtrees contained the easiest premise selection examples.

3.7 Examples of predictions from RNN

In this section, we show two examples of predictions from the recurrent neural NMT system and compare them with the respective XGBoost predictions. All the presented predictions come from the systems trained on the `basic` data set. Note that in both cases below, the NMT predictions are more diverse, expressing different proof approaches and allowing quite different proof attempts. On the other hand, as soon as XGBoost ranks high a bad set of lemmas that mislead the E prover, adding more premises does not help in these cases.

3.7.1 Theorem `t128_zfmisc_1`

The Mizar statement of the theorem

```
for x, y, z, Y being set holds ( [x,y] in [:{z},Y:]
iff ( x = z & y in Y ) )
```

The NMT predictions

1. `d1_enumset1 t71_enumset1 t69_enumset1 t70_enumset1 l54_zfmisc_1`

Table 3.4: Performance of the NMT and XGBoost models evaluated on the testing part of the subproofs data set. We use our similarity metrics (Jaccard index and Coverage) and ATP evaluation (columns named Proved). The first column contains information about the average height of the proof subtrees the sublemmas originated from. The second column is the number of lemmas in a given subset. The first numeric row refers to all the testing examples, independently of the height. The largest numbers in the columns are marked in bold.

		machine learning model					
		`NMT`			`XGBoost`		
height	# lemmas	Jaccard	Coverage	Proved	Jaccard	Coverage	Proved
$[1, \infty)$	7300	0.30	0.80	0.83	0.27	0.74	0.61
1	1610	0.19	0.84	0.83	0.18	**0.82**	0.66
$(1, 2]$	**1803**	0.29	**0.87**	0.84	0.25	0.79	0.60
$(2, 3]$	1431	0.34	0.82	0.81	0.30	0.75	0.57
$(3, 4]$	936	0.36	0.75	0.82	0.31	0.69	0.56
$(4, 5]$	580	0.34	0.70	0.83	0.31	0.64	0.60
$(5, 6]$	319	0.37	0.70	0.86	0.33	0.64	0.62
$(6, 7]$	223	0.34	0.64	0.83	0.32	0.60	0.64
$(7, 8]$	124	0.37	0.70	0.89	0.34	0.66	**0.79**
$(8, 9]$	90	**0.40**	0.69	**0.92**	**0.36**	0.63	0.71

2. d2_tarski t77_enumset1 t79_enumset1 t76_enumset1 t84_enumset1 l54_zfmisc_1
3. l38_zfmisc_1 t69_enumset1 t70_enumset1 t71_enumset1 t20_zfmisc_1 l54_zfmisc_1
4. d1_tarski t69_enumset1 l54_zfmisc_1 d2_tarski
5. d1_tarski t69_enumset1 l54_zfmisc_1
6. l33_zfmisc_1 t69_enumset1 t70_enumset1 t71_enumset1 t20_zfmisc_1 l54_zfmisc_1
7. t76_enumset1 d1_enumset1 l54_zfmisc_1
8. t20_zfmisc_1 t69_enumset1 t70_enumset1 t71_enumset1 t65_zfmisc_1 l54_zfmisc_1
9. d2_tarski t70_enumset1 t71_enumset1 t69_enumset1 l54_zfmisc_1
10. l24_zfmisc_1 t69_enumset1 t70_enumset1 t71_enumset1 d1_enumset1 l54_zfmisc_1

The XGBoost predictions (ranking)

1. d1_tarski
2. t69_enumset1
3. t70_enumset1
4. t71_enumset1
5. l54_zfmisc_1
6. t106_zfmisc_1
7. t77_enumset1
8. d3_tarski
9. d2_tarski
10. t82_enumset1

⋮

Comparison The E prover without `auto` mode was able to prove `t128_zfmisc_1` with the 5th prediction proposed by NMT:

```
d1_tarski t69_enumset1 l54_zfmisc_1
```

but no proof could be found with the top slices of the ranking proposed by XGBoost. The reason for this is that premises appearing in the top part of the ranking:

```
t69_enumset1 t70_enumset1 t71_enumset1
```

are very similar, and the E prover is stuck with them, even with higher CPU time limits. Below we show the Mizar statements of the discussed premises:

```
d1_tarski:    for x being set holds ( x in it iff x = y )
l54_zfmisc_1: [x,y] in [:X,Y:] iff x in X & y in Y
t69_enumset1: for x1 being set holds {x1,x1} = {x1}
t70_enumset1: for x1, x2 being set holds {x1,x1,x2} = {x1,x2}
t71_enumset1: for x1, x2, x3 being set holds {x1,x1,x2,x3} = {x1,x2,x3}
```

3.7.2 Theorem t30_tops_1

The Mizar statement of the theorem

```
for GX being TopStruct for R being Subset of GX holds
( R is open iff R ‘ is closed )
```

The NMT predictions

1. dt_k3_subset_1 d1_tops_1 t52_pre_topc t29_tops_1 d8_tops_1 d7_tops_1
2. t100_xboole_1 t12_setfam_1 t28_xboole_1 t48_xboole_1 commutativity_k2_tarski d5_subset_1 t3_subset redefinition_k9_subset_1 d3_struct_0 dt_l1_pre_topc t32_subset_1 dt_k3_subset_1 dt_k2_subset_1 d4_subset_1 d6_pre_topc
3. involutiveness_k3_subset_1 t29_tops_1 t101_tops_1 t52_pre_topc dt_k3_subset_1 d8_tops_1 d7_tops_1
4. t100_xboole_1 t12_setfam_1 t36_xboole_1 t48_xboole_1 t7_ordinal1 t2_xboole_1 d5_xboole_0 t45_xboole_1 t47_xboole_1 dt_k6_subset_1 redefinition_k6_subset_1 t1_boole redefinition_k7_subset_1 d3_struct_0 dt_l1_pre_topc t22_pre_topc d6_pre_topc commutativity_k2_tarski t52_pre_topc
5. t100_xboole_1 t12_setfam_1 d5_subset_1 t2_boole t91_tops_1 involutiveness_k3_subset_1 dt_k3_subset_1 t5_boole t4_subset_1 cc3_tops_1 dt_k2_subset_1 d4_subset_1 d3_tops_1 t52_pre_topc fc1_xboole_0 cc1_tops_1
6. t28_xboole_1 t12_setfam_1 commutativity_k3_xboole_0 t100_xboole_1 t22_xboole_1 t36_xboole_1 redefinition_k7_subset_1 redefinition_k4_subset_1 t65_tops_1 dt_k2_tops_1 fc1_tops_1 t48_xboole_1 t74_tops_1 t69_tops_1
7. t12_setfam_1 t70_enumset1 t100_xboole_1 d5_subset_1 t71_enumset1 t72_enumset1 t73_enumset1 t74_enumset1 t75_enumset1 t2_boole d3_tops_1 d4_tops_1 dt_k3_subset_1 t5_boole t4_subset_1 d1_tops_1 d3_struct_0 dt_l1_pre_topc involutiveness_k3_subset_1 dt_k2_pre_topc

8. d10_xboole_0 t2_xboole_1 t43_xboole_1 t12_xboole_1 commutativity_k2_xboole_0 t41_xboole_1 t36_xboole_1 t44_xboole_1 d5_subset_1 t3_subset involutiveness_k3_subset_1 dt_k3_subset_1 t12_xboole_1 redefinition_k7_subset_1 l78_tops_1 dt_k2_pre_topc t52_pre_topc fc11_tops_1 dt_k2_tops_1 t62_tops_1 l80_tops_1
9. d10_xboole_0 t2_xboole_1 t43_xboole_1 t12_xboole_1 commutativity_k2_xboole_0 t41_xboole_1 t36_xboole_1 t44_xboole_1 d5_subset_1 dt_k3_subset_1 involutiveness_k3_subset_1 t3_subset t7_xboole_1 t44_tops_1 redefinition_k7_subset_1 l78_tops_1 dt_k2_pre_topc t84_tops_1 t48_pre_topc
10. d10_xboole_0 t2_xboole_1 t43_xboole_1 t12_xboole_1 commutativity_k2_xboole_0 t41_xboole_1 t36_xboole_1 t44_xboole_1 d5_subset_1 dt_k3_subset_1 involutiveness_k3_subset_1 t3_subset dt_k2_subset_1 d4_subset_1 t35_subset_1 involutiveness_k3_subset_1 dt_k3_subset_1 redefinition_k7_subset_1 l78_tops_1 dt_k2_pre_topc l80_tops_1 t52_pre_topc fc11_tops_1

The XGBoost predictions (ranking)

1. d10_xboole_0
2. t3_subset
3. d4_subset_1
4. d3_struct_0
5. d5_subset_1
6. involutiveness_k3_subset_1
7. dt_k3_subset_1
8. dt_k2_pre_topc
9. dt_l1_pre_topc
10. d3_tarski

⋮

Comparison The E prover without auto mode was able to prove t30_tops_1 with the 3rd prediction proposed by NMT:

```
involutiveness_k3_subset_1 t29_tops_1 t101_tops_1 t52_pre_topc
dt_k3_subset_1 d8_tops_1 d7_tops_1
```

actually using these 3 premises:

```
t29_tops_1 involutiveness_k3_subset_1 dt_k3_subset_1
```

The E prover was not able to prove the theorem with any top slice of the XGBoost ranking.

3.8 Conclusions and future work

We have shown that state-of-the-art recurrent neural architectures – originally designed for natural language tasks such as machine translation – are very useful for premise selection. In particular, they can be used to implement (i) stateful / conditional premise selection and (ii) beam search with multiple output sequences that may differ a lot while being meaningful as a whole. Our experiments show substantial improvement over the state of the art obtained by such methods. We have also developed several data representation and augmentation methods that result in additional improved performance of both the old and new premise selection methods.

NMT architectures are also more natural in some aspects. There is no need for hand-designed features of formulas and no need to construct negative training examples. This is important, because in theorem proving it is often difficult (or impossible) to say that a particular selection of premises *cannot* lead to a proof. Once the recurrent neural network is trained, it directly outputs the most probable sequences of candidate premises – not just their rankings. We have used 10 most probable sequences for the experiments described here, but larger numbers can be used and given to ATPs, depending on available resources.

Future work includes integration into AI/ATP meta-systems interleaving premise selection with learning such as ATPboost and MaLARea [161]. Another direction is tighter integration between the ATPs and the neural network so that the prover can take advantage of the order in which the premises are presented – a similar idea was implemented in [23], where the order of selected premises influenced the run of the SPASS theorem prover. Conditional selection might also be implemented in learning-guided ATP systems such as rlCoP [83], plCoP [174], TacticToe [57] and ENIGMA [35, 74]. Neural network research is advancing quickly and experiments with other stateful neural architectures may bring further improvement. Finally, we could provide the neural networks with more information about the premises. Currently, the premises are just names (words) and NMT can only learn their *latent semantics* [39]. Adding more information about their logical representation and meaning [121] may be useful.

Chapter 4

Guiding inferences in connection tableau by recurrent neural networks

Abstract

We present a dataset and experiments on applying recurrent neural networks (RNNs) for guiding clause selection in the connection tableau proof calculus. The RNN encodes a sequence of literals from the current branch of the partial proof tree to a hidden vector state; using it, the system selects a clause for extending the proof tree. The training data and learning setup are described, and the results are discussed and compared with the state of the art using gradient boosted trees. Additionally, we perform a conjecturing experiment in which the RNN does not just select an existing clause, but completely constructs the next tableau goal.

4.1 Introduction

There is a class of machine learning sequence-to-sequence architectures based on recurrent neural networks (RNNs) which are successfully used in the domain of natural language processing, in particular for translation between lan-

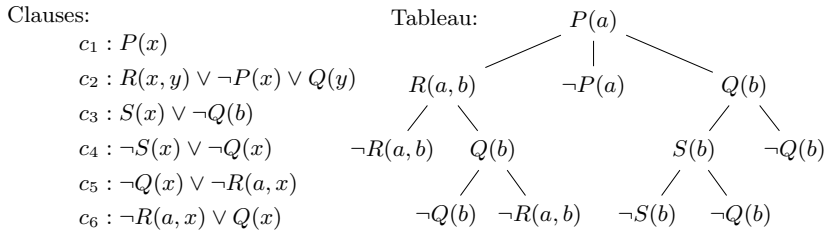


Figure 4.1: Closed connection tableau for a set of clauses.

guages [34]. Recently, such architectures proved useful also in various tasks in the domain of symbolic computation [53, 98, 131, 167]. The models *encode* the source sequence to a *hidden vector state* and *decode* from it the target sequence.

In this work, we employ such neural methods to choose among the non-deterministic steps in connection-style theorem proving. More specifically, we want to learn the *hidden proving states* that correspond to the evolving proof trees and condition the next prover steps based on them. To this end, from a set of connection tableau proofs we create a dataset (Section 4.2) of *source-target* training examples of the form (*partial_proof_state, decision*) that we then use to train the neural models (Section 4.3). The results are reported in Section 4.4. Section 4.5 shows an additional experiment with predicting (conjecturing) tableau goals.

The connection tableau seems suitable for such methods. The connection proofs grow as branches of a tree rooted in a starting clause. The number of options (clauses) to choose from is relatively small compared to saturation-style provers, where the number of clauses grows quickly to millions during the search. The tableau branches representing the proof states can be the sequential input to the RNNs, which can then decode one or more decisions, i.e., choices of clauses.

4.2 A data set for connection-style internal guidance

The experimental data used in this work originate from the Mizar Mathematical Library (MML) [60] translated [159] to the TPTP language. We have used the leanCoP connection prover [122] to produce 13 822 connection proofs from the

Mizar problems.

The connection tableau calculus searches for *refutational proofs*, i.e., proofs showing that a set of first-order clauses is *unsatisfiable*. Figure 4.1 (adapted from Letz et al. [100]) shows a set of clauses and a *closed connection tableau* constructed from them, proving their unsatisfiability. A closed tableau is a tree with nodes labeled by literals where each branch is *closed*. A *closed* branch contains a pair of *complementary literals* (identical but with opposite polarities). An *open* branch can be extended with descendant nodes by applying one of the input clauses. This *extension* step can often be performed with several different clauses – this is the main non-determinism point. Choosing the correct clause advances the proof, whereas choosing wrongly leads to redundant exploration followed by backtracking.

The training data for choosing good clauses were extracted from the proofs as follows. First, formulas in the proofs were made more uniform by substituting for each universal variable the token `VAR` and for each Skolem function the token `SKLM`. For each non-root and non-leaf node n in each proof tree, we exported two types of paths, which form two kinds of input data for the neural architecture:

(1) $P_{\text{lits}}(r \to n)$ – the literals leading from the root r to the node n,

(2) $P_{\text{cls}}(r \to n)$ – the clauses that were chosen on the way from r to n.

The output data are created as follows. For each node n we record the decision (i.e., the clause) that led to the proof. Let clause(n) be the clause selected at node n. For instance, if n is the node labeled by $R(a, b)$ in Figure 4.1, clause(n) = c_6.

The pairs $\big(P_{\text{lits}}(r \to n), \text{clause}(n)\big)$ and $\big(P_{\text{cls}}(r \to n), \text{clause}(n)\big)$ constitute two different sets of training examples for learning clause selection. Each of these sets contains 567 273 pairs. Additionally, we have constructed similar data in which the output contains not only the choice of the next clause, but a sequence of two or three such consecutive choices. All these data sets[1] were split into training, validation and testing sets – the split was induced by an initial split of the proofs in proportions 0.6, 0.1 and 0.3, respectively.

4.3 Neural modelling and evaluation metric

As a suitable sequence-to-sequence recurrent neural model we used an implementation of a neural machine translation (NMT) architecture by Luong

[1] The tableau proofs and the sequential training data extracted from it are available at `https://github.com/BartoszPiotrowski/guiding-connection-tableau-by-RNNs`

Table 4.1: Predictive accuracy of the NMT system trained on two types of source paths (literals or clauses), decoding 1–3 consecutive clauses. 1 or 10 best outputs were decoded and assessed. Predictions based on paths of literals are substantially better than these based on path of clauses.

	paths of literals		paths of clauses	
# clauses to decode	1 best output	10 best outputs	1 best output	10 best outputs
1	0.64	0.72	0.17	0.36
2	0.11	0.19	0.03	0.07
3	0.05	0.07	0.01	0.02

et al. [102], which was already successfully used for symbolic tasks in [167] and [131]. All the hyper-parameters used for training were inherited from [167].

Let $\mathit{subsequent\_clauses}_i(P_{\text{lits/cls}}(r \to n))$ be a set of i-long sequences of clauses found in the provided proofs, following a given path of literals/clauses from the root to a node n.[2] Let $\mathit{clauses\_from\_model}^i_k(P_{\text{lits/cls}}(r \to n))$ be a set of k i-long sequences of clauses decoded from the NMT model (we decoded for $k = 1$ or $k = 10$ most probable sequences using the *beam search* technique [52]). We consider the prediction from the model for a given path of literals/clauses as successful if the sets $\mathit{subsequent\_clauses}_i(P_{\text{lits/cls}}(r \to n))$ and $\mathit{clauses\_from\_model}^i_k(P_{\text{lits/cls}}(r \to n))$ intersect. The metric of predictive accuracy of the model is the proportion of successful predictions on the test set.

4.4 Results

The average results for the above metric are shown in Table 4.1. We can see that predicting the next clause is much more precise than predicting multiple clauses. The accuracy of predicting the next clause(s) from a sequence of clauses is lower than predicting the next clause(s) from a sequence of literals, which means the literals give more precise information for making the correct decision.

We have also investigated how the performance of NMT depends on the length of the input sequences. The results for the neural model trained on

[2] E.g., for the proof from Fig. 4.1, we have $(c_4) \in \mathit{subsequent\_clauses}_1(P_{\text{lits}}(P(a) \to S(b))$, $(c_6, c_5) \in \mathit{subsequent\_clauses}_2(P_{\text{lits}}(P(a) \to R(a,b))$, $(c_6) \in \mathit{subsequent\_clauses}_1(P_{\text{cls}}(c_1 \to c_2)$, or $(c_3, c_4) \in \mathit{subsequent\_clauses}_1(P_{\text{cls}}(c_1 \to c_2)$.

Table 4.2: Predictive accuracy of the NMT and XGBoost systems for different lengths of input sequences consisting of literals. The NMT model, as opposed to the XGBoost one, can take advantage of longer input sequences (i.e., more complex context).

length	1	2	3	4	5	6	7	8
NMT	0.19	0.48	0.64	0.70	0.68	0.72	0.79	0.85
XGB	0.43	0.35	0.42	0.39	0.47	0.41	0.51	0.46

the paths of literals as the input are shown in the second row of Table 4.2. As expected, the longer the input sequence, the better is the prediction. The neural model was capable of taking advantage of a more complex context. This differs significantly with the path-characterization methods using manual features (as in [83]) that just average (possibly with some decay factor) over the features of all literals on the path.

To compare with such methods, we trained a classifier based on gradient boosted trees for this task using the XGBoost system [33], which was used for learning feature-based guidance in [83]. To make the task comparable to the neural methods, we trained XGBoost in a multilabel setting, i.e., for each partial proof state (a path of literals) it learns to score all the available clauses, treated as labels. Due to limited resources, we restrict this comparison to the MPTP2078 subset of MML which has 1383 different labels (the clause names).

The average performance of XGBoost on predicting the next clause from the (featurized) path of literals was 0.43. This is lower than the performance of the neural model, also using literals on the path as the input (0.64). The XGBoost performance conditioned on the length of the input path is shown in the third row of Table 4.2. XGBoost is outperforming NMT on shorter input sequences of literals, but on longer paths, XGBoost gets substantially worse. The performance of the recurrent neural model grows with the length of the input sequence, reaching 0.85 for input length 8. This means that providing more context significantly helps the recurrent neural methods, where the hidden state much more precisely represents (encodes) the whole path. The feature-based representation used by XGBoost cannot reach such precision, which is likely the main reason for its performance flattening early and reaching at most 0.51.

Table 4.3: Predictive accuracy of conjecturing literals by the NMT system for input sequences of different lengths. The longer the input sequence, the higher accuracy of the predicted literals.

length	1	2	3	4	5	6	7	**all**
NMT	0.04	0.05	0.08	0.11	0.14	0.16	0.34	0.08

Table 4.4: Literals conjectured by NMT *vs.* the correct ones. (1) is an example of a correctly predicted output; in (2) NMT was wrong but proposed a literal which is similar to the proper one; (3) shows a syntactically incorrect literal produced by NMT. Shorter literals with fewer symbols were easier to predict by the NMT system.

	NMT prediction	correct output
(1)	`m1_subset_1(np__1,k4_ordinal1)`	`m1_subset_1(np__1,k4_ordinal1)`
(2)	`m1_subset_1(SKLM,k1_zfmisc_1(SKLM))`	`m1_subset_1(SKLM,SKLM)`
(3)	`k2_tarski(SKLM,SKLM)=k2_tarski(SKLM`	`k2_tarski(SKLM,SKLM)=k2_tarski(SKLM,SKLM)`

4.5 Conjecturing new literals

As an additional experiment demonstrating the power of the recurrent neural methods we constructed a data set for *conjecturing* new literals on the paths in the tableau proofs. The goal here is not to select a proper literal, but to *construct* it from the available symbols (the number of them for the MML-based data set is 6442). This task is impossible to achieve with the previous methods that can only *rank* or *classify* the available options. Recurrent neural networks are, on the other hand, well-suited for such tasks – e.g., in machine translation, they can learn how to compose grammatically correct and meaningful sentences.

It turns out that this more difficult task is to an extent feasible with NMT. Table 4.3 shows that NMT could propose the right next literal on the path in a significant number of cases. Again, there is a positive dependence between the length of the input sequence and the predictive performance. Most of the times the correct predictions involve short literals, whereas predicting longer literals is harder. The proposed longer literals often not only do not match the right ones but have an improper structure (see Table 4.4 for examples of the NMT outputs).

4.6 Conclusions and future work

In this work, we proposed RNN-based encoding and decoding as a suitable representation and approach for learning clause selection in connection tableau. This differs from previous approaches – both neural and non-neural – by emphasizing the importance of the evolving proof state and its accurate encoding.

The approach and the constructed datasets also allow us to meaningfully try completely new tasks, such as automatically conjecturing the next literal on the path. The experimental evaluation is encouraging. In particular, it shows that the longer the context, the more precise the recurrent methods are in choosing the next steps, unlike the previous methods.

The evaluation and data sets have focused (as similar research studies [46, 79]) on the *machine learning performance*, which is known to underlie the theorem proving performance. Future work includes integrating our neural method into a connection prover and performing ATP evaluation similar to [35, 83] that would compare with a base connection prover in terms of *proving performance*.

Chapter 5

Towards learning quantifier instantiation in SMT*

Abstract

This chapter describes a work where machine learning (ML) is applied to solve quantified satisfiability modulo theories (SMT) problems more efficiently. The motivating idea is that the solver should learn from already solved formulas to solve new ones. This is especially relevant in classes of similar formulas.

We focus on the enumerative instantiation – a well-established approach to solving quantified formulas anchored in the Herbrand's theorem. The task is to select the right ground terms to be instantiated. In ML parlance, this means learning to rank ground terms. We devise a series of features of the considered terms and train on them using boosted decision trees. In particular, we integrate the LightGBM library into the SMT solver cvc5. The experimental results show that the ML-guided solver enables us to solve more formulas than the base solver and reduce the number of quantifier instantiations. We also do an ablation study on the features used.

*This chapter is based on a joint work with Mikoláš Janota and Jelle Piepenbrock. I was involved in designing features used by machine learning, and I was responsible for designing, implementing and running looping-style experiments. Mikoláš Janota was responsible for implementing featurizer within cvc5, and implementing an interface between cvc5 and LightGBM. I wrote Section 5.4 and helped to write other sections as well.

5.1 Introduction

Solving formulas containing quantifiers in the context of Satisfiability Modulo Theories (SMT) is famously difficult. This difficulty is inherent since quantifiers lead to undecidability or high computational complexity [51,110]. Nevertheless, quantifiers are indispensable in practical problems. Notably, in software verification, they are used to express properties of memory, e.g., that an array is sorted. This chapter tackles the question: *Can machine learning (ML) make SMT solvers more efficient in the context of quantifiers?*

The potential of ML is to enable the solver to learn from problem instances that it has already solved. In contrast, current SMT solvers only take into account one formula during solving. However, integrating ML into this context is not straightforward. We are facing two main challenges:

1. ML operates in an approximate setting, while SMT is anchored in a rigorous background where inference steps need to follow the logic in question. This means the solver inference steps must be followed and the ML integrated into the solver framework.

2. SMT solvers make millions of decisions for a single problem. How can ML be integrated without dramatically slowing down the solver?

This chapter proposes a design that enables ML to steer the state-of-the-art SMT solver cvc5 [8][2] in quantifier instantiation (see Figure 5.1).

A general technique to handle quantifiers in SMT is to gradually instantiate the quantified sub-formulas with ground terms until obtaining contradiction. For instance the formula $(\forall x\, f(x) > x) \wedge (\forall y\, f(y) < 0)$ is readily refuted by instantiating both x and y with 0.

The terms to be used in instantiations may be chosen either by making use of syntactic properties, e.g. by *e-matching* [40], or by utilizing semantic properties, e.g. *model-based quantifier instantiation* [58]. Interestingly, the complexity of these techniques may not always pay off: simple *enumerative instantiation* of terms can often give better results [75,136]. For all of these techniques, the most important challenge is a large number of possible terms that can be chosen for instantiation, especially in later stages of solving.

In recent years, ML has been applied in countless settings, from computer vision [93] to natural language processing [41]. There is also work to learn decision-making for first-order theorem proving, cf. [72], but the use of ML for SMT guidance still has large untapped potential.

[2] cvc5 is a successor to CVC4 [8, 10].

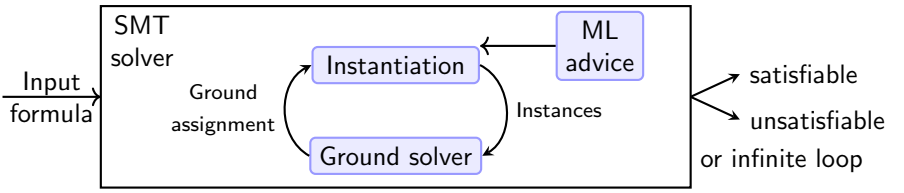


Figure 5.1: Schematic of the SMT solver with machine learning guidance for quantifier instantiation.

In this work, we use machine learning to improve the performance of cvc5 in real-time, by learning a scoring function for terms that guides the quantifier instantiation process. This addresses our first challenge, i.e., how to use an inherently approximate method within SMT. The second challenge of avoiding dramatic slowdown by ML within the solver is achieved by the choice of features, ML model, and its tight integration into the solver.
This work has the following primary contributions.

1. We design an integration of ML guidance for quantifier instantiation in the context of SMT. In particular, the enumerative instantiation is guided by ML during the run of the solver, while learning from existing solutions to already solved problems.

2. We implement the proposed method in cvc5 and the implementation shows a substantial increase in the number of solved instances and lowers the number of instantiations needed for many proofs.

5.2 Background

Throughout the chapter we assume familiarity with first-order logic, in particular, with satisfiability modulo theories (SMT) [12]. Formulas with no quantifiers, called *ground formulas*, are solved using the DPLL(T) paradigm [119]. DPLL(T) abstracts first-order logic atoms as propositional variables enabling the use of a SAT solver to reason about the Boolean structure of the formula and *theory solvers* to reason about theories.

SMT solvers reason about quantifiers by instantiating with ground terms to strengthen the ground part of the formula. Effectively, a quantified subformula or quantified expression $(\forall x_1 \dots x_n \, \phi)$ is a source of lemmas of the form $(\forall x_1 \dots x_n \, \phi) \Rightarrow \phi\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, where ϕ is quantifier-free

and t_i are ground terms. For instance, $\forall x\, f(x) > 0$ may be instantiated as $(\forall x\, f(x) > 0) \Rightarrow f(0) > 0$. For simplicity, assume that quantifiers are removed on preprocessing by Skolemization.

This approach results in a loop that moves back and forth between a *ground solver* and the *instantiation module*; see Figure 5.1. The ground solver only sees quantifiers as propositions that either should hold or not. Once the ground solver finds a satisfying assignment for the ground part of the formula, control is handed over to the instantiation module, which generates new instances of the quantified sub-formulas that currently should hold. This strengthens the ground part of the formula and the process repeats. *Our contribution is to provide ML advice for the instantiation module with the aim of suggesting instantiations that lead to unsatisfiability in the ground solver.*

There is a bevy of methods for choosing instantiations. For decidable fragments, dedicated approaches exist, e.g., for bit-vectors or linear arithmetic [17, 48, 116, 137]. General quantifiers are most notably tackled by *e-matching*, based on syntactic properties of the terms [40] and *model-based* [58] or *conflict-based* [138] instantiation, relying on the semantics of the formula. Niemetz et al. apply syntax-guided instantiation term generation [117].

The instantiation method we focus on here is *enumerative instantiation*. While the method is probably the most straightforward one, it has good performance on many SMT problem categories and adds to the robustness of the solver [136].

5.2.1 Enumerative instantiation

Herbrand's theorem [66] guarantees that for an unsatisfiable first-order logic formula, finitely many instantiations are sufficient to obtain an unsatisfiable ground part, and, these instantiations only need to use the Herbrand universe. Completeness is not guaranteed in theories (e.g., $\forall x : \mathbb{R}\ x^2 \neq 2$). However, the application of the theorem in SMT is justified as it provides a viable way to deal with the complex problem of quantifier instantiation.

Reynolds et al. invoke a stronger variant of Herbrand's theorem that enables a more practical method for quantifier instantiation [136]. It is sufficient to consider only the terms already within the ground part of the formula generated so far. This insight leads to the *enumerative instantiation* strategy, the technique we augment with machine learning guidance in this work. For a formula $G \wedge \forall x_1 \ldots x_n\, \phi$, with G ground, collect all ground terms $\mathcal{T}$ in G and strengthen G by an instantiation of ϕ by an n-tuple $t_1, \ldots, t_n$ with $t_i \in \mathcal{T}$; repeat the process until G becomes unsatisfiable or until resources are exhausted. The tuples

are enumerated systematically to guarantee fairness.

As a motivational (toy) example consider the following conjunctive set of formulas within the logic of uninterpreted functions and linear integer arithmetic (UFLIA).

$$\{f(d) > f(d+2),\ c \leq 0 \vee \underbrace{\forall x\, f(x) < f(x+1)}_{q}\}$$

For the ground solver, the quantifier is abstracted as a Boolean constant q and the instantiation module is responsible for adding lemmas of the form $q \Rightarrow f(t) < f(t+1)$ for some ground term t. Consider a context where the solver decides that $\neg(c \leq 0)$, which forces q to be true. Ideally, in this situation the solver instantiates x first with d and then with $d+1$, resulting in the following steps:

ground formula	**additional ground terms**
$\{c \leq 0 \vee q, f(d) > f(d+2)\}$	$\{c, 0, d, d+2, f(d), f(d+2)\}$
$\{q \Rightarrow f(d) < f(d+1)\}$	$\{d+1, f(d+1)\}$
$\{q \Rightarrow f(d+1) < f(d+2)\}$	$\{d+2, f(d+2)\}$

From transitivity of $>$, the ground part gives a contradiction for the current context, forcing q to false and $c \leq 0$ to true. Already this small example shows the difficulties we are facing. For instance, instantiating with the term $f(d+2)$ results in $q \Rightarrow f(f(d+2)) < f(f(d+2)+1)$, which not only is unhelpful but also produces a harder ground instance.

Individual instantiations lead to the addition of new ground terms into a sequence. We refer to the position of a term in the sequence as its *age*; in the above example, the term d has age 0. Terms added by the same instantiation are in the same *phase*; in the above example, the terms $d+1, f(d+1)$ are added in phase 2.

As an additional filter, cvc5 uses a technique called *relevant domain*, introduced by Ge and de Moura [58]. Intuitively, a term becomes relevant for a certain quantified variable if it appears in the same position as the variable, e.g., if $f(x)$ appears in the quantified formula and there is a ground term $f(t)$, the sub-term t is relevant for x; this is further closed by equality. By default, cvc5 first considers only instantiations by terms from the relevant domain and only after that moves on to the rest. Formulas with multiple quantified sub-formulas are solved by instantiating the sub-formulas independently but in a fair manner.

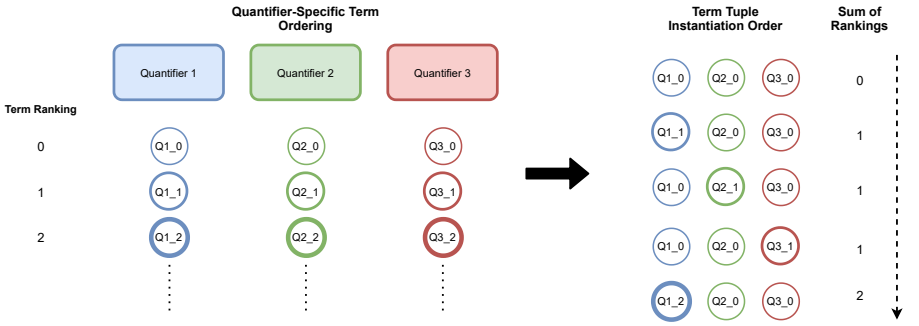


Figure 5.2: Schematic example of the term enumeration process with term tuple ordering based on the sum of term rankings. The task of the ML model is to take features based on the quantified formula, the variable, the terms, and the context to score the terms, thus changing the term ranking to deliver better instantiations. In cases where multiple quantifiers have to be instantiated at the same time, we instantiate with term tuples, which are ordered by the sum of the rankings of the individual terms.

5.3 Learning ordering of terms

When using the enumerative instantiation strategy, the SMT solver uses an ordering of the available terms (by default, this is primarily based on the *age* of the terms) and enumerates terms in this order, trying instantiations. As shown in Figure 5.2, when there are multiple quantifiers that need to be instantiated with a tuple of terms, the solver uses the sum of the rankings of the individual terms to determine the ranking of the term tuple [75].

cvc5 considers one quantified expression at a time. In this work, we also only consider the reranking of terms for a single variable at a time.[3] Thus, at each decision point of the quantifier module, we consider the current quantified expression Q, the variable V, the term T, and the wider context *Context*, containing all other quantified expressions and the ground part of the problem. Featurization is then viewed as a function $F : (Q, V, T, \mathit{Context}) \to \mathbb{R}^n$.

The machine learning heuristic's task is to reorder the term candidate lists for each quantifier so that more useful instantiations are tried earlier (see Figure 5.2). For this, we use a *scoring function* and rank the candidates according

[3]The enumeration still creates tuples of terms to instantiate quantified expressions with multiple variables.

to this score. This scoring function $S\colon \mathcal{F} \to [0,1]$ takes as its input the *features* obtained by applying the featurization F to a tuple $(Q, V, T, \mathit{Context})$.

The returned score is intended to reflect how likely it is that term T was used to instantiate variable V in quantified expression Q with the context *Context* in the final proof. The training data for the ML model is based on previously found proofs, so that we can extract a label for each tuple $(Q, V, T, \mathit{Context})$, based on what decision was made in a known proof of the problem. This label is 1 if the instantiation that the tuple represents was used in a successful proof (which we will call a *positive* example) or 0 otherwise. During solving, the scoring function S is applied to each candidate term. The terms are then sorted according to their score by *stable* sort, i.e., equally scored terms remain in their original order (see Figure 5.2).

The scoring function S is implemented as an ML model. Specifically, we experimented with a logistic regression model and with gradient boosted decision trees (GBDT). In contrast to popular neural network methods, these methods are sufficiently fast to run at solving time within the solver loop [146]. In the initial experiments, the boosted trees performed substantially better than logistic regression, thus we keep it for the rest of our experiments.

GBDT uses an ensemble of decision trees, where decisions of all the individual trees are aggregated into a more reliable decision. Gradient boosted trees are widely used in machine learning applications. In particular, this algorithm is one of the most successful approaches in machine learning competitions [120]. They have also been used for machine learning guidance in first-order logic [72, 127]. After training, we use this ensemble of decision trees to predict the label of each candidate term. Ideally, the trees predict 1 if the candidate leads to a proof and 0 if it does not. In practice, the prediction of the model is a float number between 0 and 1, which can be interpreted as a probability. We used the library LightGBM [88], a fast and efficient implementation of the GBDT algorithm.

5.3.1 Featurization

The GBDT algorithm makes decisions based on a representation of the state of the solver, the relevant quantified expression and the term that is being considered. This representation of the data is called the *featurization* of the data. There are several categories of features with different properties. First, we list them here. Afterwards, there are subsections with the details of the features in that category. Note that we use the single-letter abbreviations following the categories to denote them in figures in later sections. Table 5.1 contains a list of categories of the features we use and abbreviations designating them.

Table 5.1: Short description of feature categories. The abbreviations are later used to concisely refer to these feature categories.

category name	description	abbreviation
procedural features	Data from the solver, such as the age of terms, number of times a term was tried.	p
bag-of-words features	Amounts of nodes of certain type in quantified expression or term.	b
context features	The parent symbols of variables and the parents of the head symbols in candidate terms.	c
numeral features	The minimum and maximum number used in quantified expression or term.	n
parent features (parent label propagation)	Terms that are necessary to create the final proof term are also labeled as positive examples.	P

The designed features provide a simple characterization of the training examples. However, they are extracted using existing, efficiently implemented mechanisms of cvc5, which makes the process fast enough. All these design decisions enable us to perform ML prediction online, during the proof search. In Subsection 5.4.3, we show an ablation study, where impacts of each feature category for the performance of the solver can be observed.

Procedural features

The first category of features is the set of procedural features, properties of the solving process that can be quickly calculated within cvc5. Several of these features were explained in Subsection 5.2.1. In this category are the *age* and *phase* of the term. In addition to these, the features *varFrequency*, *tried* and *depth* are used. *VarFrequency* indicates how many times the variable (V) under consideration appears in the quantified expression (Q). The *tried* feature indicates how many times this term (T) was already tried within this quantified expression. The *depth* is simply the syntactic depth of the term.

Bag-of-words (BOW) features

The bag-of-words features represent the number of occurrences of symbols in the given quantified expression and term. In our design, uninterpreted symbols are treated anonymously and interpreted symbols non-anonymously. This means that any interpreted symbol, such as $+$, will get its own feature whose value is the number of occurrences of the symbol in the formula. Uninterpreted symbols, i.e., those that were introduced by the user are collapsed into representative classes. For instance, all function symbols are represented by one class. The numerals (interpreted constants) are a special case. While these are interpreted, there are infinitely many of them and they therefore cannot have separate features, and therefore are also collapsed into a single node type in this feature category. However, in the *numeral features* (Subsection 5.3.1), we deal with numerals in a more semantic way.

For the calculation of BOW we use the abstract syntax tree (AST) of cvc5. For every symbol appearing in terms and formulas cvc5 determines its *kind*. These kinds include, e.g., *variable*, *skolem*, *not*, *and*, *plus*, *forall*, and many others. We use these syntactic kinds to define a *bag-of-words*-type featurizer $\mathrm{BOW}(x)$, where x is a term or a quantified formula, and the information returned by BOW consists of counts of kinds of symbols appearing in x. For example, $\mathrm{BOW}(\forall x\,(2+x = \mathrm{skl}_1+3)) = \{\mathit{forall}:1, \mathit{variable:}\ 1, \mathit{const}:2, \mathit{skolem}:1, \mathit{plus}:2\}$. Non-occurring kinds are set to 0 in this representation. We remark that cvc5 represents formulas as directed acyclic graphs, rather than trees, which is also reflected here, i.e., any repeated sub-formula is counted only once.

Context features

The third category of features is the set of context features (see Figure 5.3). There are two types of context features used. The first is the *variable context*, which aggregates information about which symbols are used as parents of the variable in the current quantified expression. The second type is the *term head symbol context*, which contains information about which symbols are parents of the head symbol of the current candidate term in the ground term registry of the solver: this gives information about the whole problem, even across quantified expressions. The feature vectors are sparse, in the sense that most node types will not show up in every context: most of the possible features will be 0. In the figure we have simplified mostly to the symbols that do appear, but for the ML predictor many features are 0.

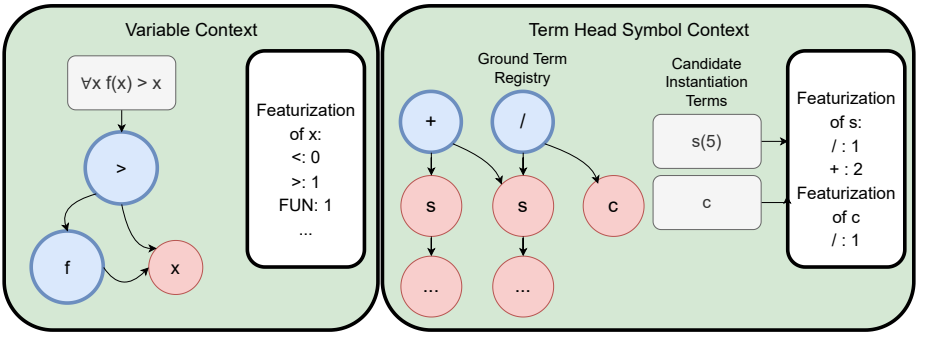


Figure 5.3: Schematic representation of the context features. The variable context aggregates which symbols occur as the parent of the variable in the current quantified expression. The term head symbol context aggregates which symbols occur in the ground term registry as parents of the head symbol of the candidate term under consideration.

Numeral features

While in the BOW features the numerals in the problems are all mapped to one single node type, this is not optimal. Especially because the benchmark problems we test on are integer arithmetic problems, giving the machine learning component some information about which numbers are in the formula should be useful. To allow the machine learning system to do some elementary comparison operations on the terms it needs to decide the score of, we add 4 additional features, which are the *minimum* and *maximum* number in the current quantified expression (Q) and the current candidate term (T). When there are no nodes of numeral type in Q or T, we fill in the features with a fixed combination of numbers where the minimum is higher than the maximum, so that it can be distinguished from the rest of the cases.

Parent label propagation

The last setting in our algorithm concerns the *parents* of the proof terms. Note that this is a different kind of setting than the features before. Here we are concerned about which candidate terms are counted as positive examples (that is to say, for which the score function should predict 1) and which are negative examples (for which the prediction should be 0).

When the base solver is run and produces a proof, a set of instantiations

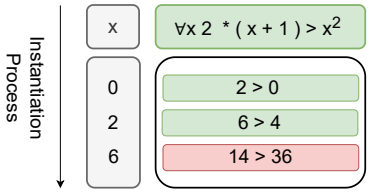


Figure 5.4: Example instantiation process that shows the difference between a term that constitutes a proof and the process of getting to the point where it is possible to instantiate that term within the cvc5 enumeration setup.

is obtained that leads to a contradiction. However, these terms are created by iterations of the enumeration loop, creating more and more terms from the Herbrand universe. It may be the case that these final instantiations cannot be created in one iteration. We may need more than 1 instantiation to create the final instantiated term that solves the problem. For example in Figure 5.4, we see that although instantiating with $x \mapsto 6$ will expose the contradiction, the ground term 6 is not available at the beginning of solving. Instantiating with 0 at the beginning makes the term 2 available. Instantiating with 2 makes the term 6 available, which leads to a contradiction. To reflect this, we propagate the positive label of these instantiations (i.e., 6) to the parent instantiations, i.e., the instantiations that were done to create the final instantiation candidate terms, 2 and 0 in the example from Figure 5.4.

5.4 Experimental evaluation

cvc5 implements multiple techniques for quantifier instantiation (see Section 5.2). Due to the inherent difficulty of the overall problem, it is not the case that one technique is uniformly better than all the others. On the contrary, the techniques exhibit a high degree of orthogonality[4] in terms of the number of solved instances [75, 136]. Therefore, it is meaningful to focus on improving the techniques independently of one another. In our scenario, we let the solver use the enumeration technique combined with the relevant domain heuristic (Section 5.2.1); all the other techniques are explicitly turned off, which also lets us more clearly isolate the effect of machine learning guidance.

[4]By saying that two techniques are *orthogonal* we mean that the symmetric difference of the sets of problems they solve is large; precisely this is demonstrated for various instantiation techniques in [75] and [136].

We refer to cvc5 with this baseline strategy as the *base solver*. We remark that this setup alone is already a very strong solver, often outperforming more complex techniques, as shown in the literature [75]. The objective of the evaluation is to compare the base solver with the base solver augmented with ML guidance.

A crucial prerequisite to training a strong ML model is to have a sizeable, high-quality set of training examples. They can be collected by running the solver on available problems and by recording which instantiations were *positive* (appeared in a proof) and which were *negative* (redundant).

Note that the notion of a *positive or negative example* is not strict here since a single problem may have multiple alternative proofs resulting in different sets of *positives* and *negatives*. Moreover, the solver may arrive at a given proof in multiple ways, performing the same set of useful instantiations but different sets of redundant ones, which results in a different collections of negative examples.

Based on these observations, it is clear that in order to collect a rich and illustrative set of training examples, it is beneficial to run the solver on a given set of problems multiple times in varied ways, which will result in multiple alternative proofs and proof searches. Thus, we establish the following methodology for collecting the training data. First, an unguided solver is run on a given set of problems. The data recorded from these proof attempts give an initial set of training examples, which is used to train an initial ML model. Then, the solver – this time guided by the ML model – is run again on the problems. This gives new training examples augmenting the database. Solving and training may be interleaved an arbitrary number of times. It constitutes a positive feedback loop – in each iteration, the solver is guided by a new, different, and hopefully stronger ML model which results in a growing and varied training set. A similar looping-style approach was already used in the context of automated theorem proving [127, 161].

In the evaluation, we focus on two separate, but equally important, goals:

1. the *cumulative goal*: solve automatically as many of the problems as possible over time. This is done by running the ML-guided solver multiple times over them and improving it by training the ML model on data collected across the runs. In this setting we gradually solve more problems that the base solver could not prove.

2. the *single-instance goal*: evaluate the ability of the learning model to improve the solver on unseen problems.

The importance of the single-instance goal is clear – this is how traditionally improvement in SMT is measured, i.e., how many more problem instances

are solved. Here we emphasize that the cumulative goal is just as important. Indeed, in many cases users wish to solve a group of formulas and are happy to leave the solver work on them for an extended period of time. This is particularly true for groups of *similar* formulas. This might be the case for example for verification conditions coming from a certain piece of software that is being verified. In such scenarios, the user is not interested if the SMT solver solves many instances in a competition but is interested in how many instances are solved from this particular set.

Note that in a traditional setting (without ML) it is unclear how to improve on the cumulative goal. In contrast, an ML-guided SMT solver naturally has the opportunity to generalize from previously solved (easier) problems to the harder ones.

5.4.1 Experimental setting

To assess the performance of our method for both these goals, we use the looping-style approach described above, additionally splitting the initial set of problems into *target set* and *holdout set*. The problems from the target set are used to gradually collect training examples for training the ML model to be used in the next iteration. The holdout set is only used for evaluation and not for training. The performance for the cumulative goal is measured by how many more instances are gradually solved from the target set. The performance for the single-instance goal is measured by the base solver with the solver guided by the ML model all obtained in the last iteration.

Algorithm 4 Incremental solving-training feedback loop ended with solving holdout problems.

Require: target problems: P_{target}, holdout problems: P_{holdout}, number of iterations: N, grid of hyper-parameters: H_{grid}

1: $M \leftarrow \{\}$ ▷ empty initial machine-learning model
2: $D \leftarrow \{\}$ ▷ empty initial set of training examples
3: **for** $i \leftarrow 0$ *to* $N - 1$ **do**
4: $L \leftarrow \textsc{Solve}(P_{\text{target}}, M)$ ▷ solve target problems, save proofs and stats
5: $D \leftarrow D \cup \textsc{ExtractTrainingExamples}(L)$ ▷ update training data
6: $H \leftarrow \textsc{GridSearch}(D, H_{\text{grid}})$ ▷ find good training hyper-parameters
7: $M \leftarrow \textsc{TrainModel}(D, H)$ ▷ train new model on all training examples
8: $\textsc{Solve}(P_{\text{holdout}}, M)$ ▷ solve holdout problems

The looping procedure is shown in Algorithm 4. The function $\textsc{Solve}(P, M)$

runs the solver over problems in P, using the ML model M for guidance. When $M = \{\}$, in the initial round 0, SOLVE(P, M) runs the solver with the standard, age-based ordering of the terms in place of the ML guidance. In the experiments, the number of iterations N is set to 20.

The used ML model (LightGBM) has multiple hyper-parameters governing its training, which potentially substantially influence the predictive performance of the model and therefore should be tuned [170]. We fix a set of several important parameters and candidate values for them (H_{grid}). These are the following: `learning_rate`: 0.01, 0.05, 0.1, `num_leaves`: 16, 64, 256, `max_bin`: 16, 64, 256.

After each update of the training set in the loop, a grid search is performed (function GRIDSEARCH) to establish the best hyper-parameters (H) for the next training (according to the AUC metric [49] on a random subset of validation examples). The number of trees in LightGBM model is an important parameter. However, we do not include this parameter in the grid search and just fix its value to 100. Increasing the number of trees typically improves the "offline" ML performance metrics. However, it also slows down producing the predictions, which in turn may decrease the number of solutions found within a time limit.

A large majority of the instantiations tried during the proof attempts are redundant, which results in a significant disproportion between numbers of the positive and the negative examples being collected. To expose the ML model more to the positives, we under-sample the negatives so that its number is kept below 10 $\times$ number of positives.

In experiments, the ML-guided solver is compared to the *base solver*. However, when considering the cumulative number of problems solved across multiple iterations, one should investigate whether the extra problems solved are really due to the learned strategy and not only due to the randomness injected into the process. Thus, to perform an ablation study, we additionally compare the ML-guided solver with a *randomized solver*. It is the same as the base solver with the following exception: it uses the predefined, age-based ordering additionally swapping each term randomly with a term next to it in the ranking with the probability 0.1. This parameter is selected heuristically: we want to have a solver which is similar to the well-performing, base solver, and at the same time is non-deterministic to some degree. Our initial experiments also indicate that deviating too much from the age-based ordering is detrimental to the solver: choosing a totally random order leads to a substantial decrease in the number of solved instances (around 30%).

In experiments, we fix a timeout of 60 s per one proof attempt for all the solvers. Note that the ML-guided solver spends a non-negligible amount of

time just on producing predictions from the ML model, which means that it will be able to perform fewer steps in its proof searches than the unguided solver. However, to have a realistic evaluation scenario, we give the same timeout for each solver, with the outlook that the ML-guided solver will compensate for the slowdown with its learned strategy.

5.4.2 Data for evaluation

For evaluation, we use six benchmarks from SMT-LIB [11]: (1) UFLIA boogie, (2) UFLIA grasshopper, (3) UFLIA tokeneer, (4) UFNIA sledgehammer, (5) UFNIA Preiner, (6) UFNIA vcc havoc.

UFNIA and UFLIA refer here to two different SMT logics: non-linear and linear integer arithmetic, respectively, with uninterpreted function symbols. (1) originates from various problems from formal verification formulated in an intermediate verification language Boogie [9]. (2) is a benchmark derived from a software verification project concerning heap-manipulating programs [132]. (3) was derived from a security verification project for biometric identification software [111]. (4) originates from Sledgehammer, a component of the Isabelle/HOL interactive theorem prover that enables applying SMT to discharge goals arising in interactive proofs [20]. The Sledgehammer problems come from various areas of mathematics and computer science. (5) was a project on verifying rewriting rules for bit-vectors irrespective of bit-width [118]. (6) are benchmarks taken from the VCC C program verifier [36] and HAVOC [163], a heap-aware verifier for C programs.

Some of the problems from these benchmarks may be solved without performing any instantiations. They are filtered out as not relevant for our evaluation. Then, the sizes of the benchmarks are: UFLIA boogie: 1005, UFLIA grasshopper: 382, UFLIA tokeneer: 257, UFNIA sledgehammer: 1329, UFNIA Preiner: 3897, UFNIA vcc havoc: 760. Each of the benchmarks is randomly split into target and holdout parts (P_{target}, P_{holdout}) of sizes 75% and 25%, respectively.

5.4.3 Results and discussion

This section presents the results of the evaluation of the ML-guided solver with one initial solving iteration performed by the unguided, base solver, and 19 training-solving iterations. Because of the non-deterministic nature of the training procedure, each loop with the ML-guided and the randomized solvers is run 3 times and the presented results are averaged.

Table 5.2: Target problems solved cumulatively across all 20 iterations of the training-solving loop by the randomized and the ML-guided solvers. ML-guided solver always performed better than the randomized one; however, the level of improvement differs: for two families it is not significant, whereas for UFLIA boogie it is surprisingly high. This shows that different families of problems are amenable to learning approaches to varying degrees.

family	randomized	ML-guided	improvement (%)	number of problems
UFLIA boogie	41.0	157.6	284.4	754
UFLIA grasshopper	159.0	183.3	15.3	287
UFLIA tokeneer	83.0	90.0	8.4	193
UFNIA sledgehammer	243.5	258.0	6.0	997
UFNIA Preiner	1228.3	1245.7	1.4	4776
UFNIA vcc havoc	513.0	515.7	0.5	570

The presentation of the results is divided into three subsections. The first two are concerned with the cumulative and single-instance goals (see introduction to Section 5.4). The last subsection presents an ablation study evaluating the importance of the different groups of features (see Subsection 5.3.1).

Cumulative goal

Figure 5.5 shows the number of solved instances for the considered families across the iterations of the loop (see Algorithm 4). All, except for the last two, families demonstrate a clear advantage of using ML guidance. When focusing on the cumulative goal (solid lines), both the ML-guided and the randomized solver exhibit diminishing returns – eventually they plateau. However, in the case of a randomized solver, a plateau occurs typically far earlier than in the ML-guided case. This is likely explained by the ability of the ML guidance to keep inventing new approaches inspired by newly solved problems. In contrast, randomization very quickly hits the wall since the original heuristic used by the base solver is already good. This is further supported by the observation that the learned ML model solves an increasing number of the overall instances, whereas the randomized one solves roughly the same number of problems in every iteration. More detailed numbers can be found in Table 5.2.

The last two families (UFNIA-Preiner, UFNIA-vcc-havoc) do not show any substantial improvements with ML guidance. Possibly, this might be that we are

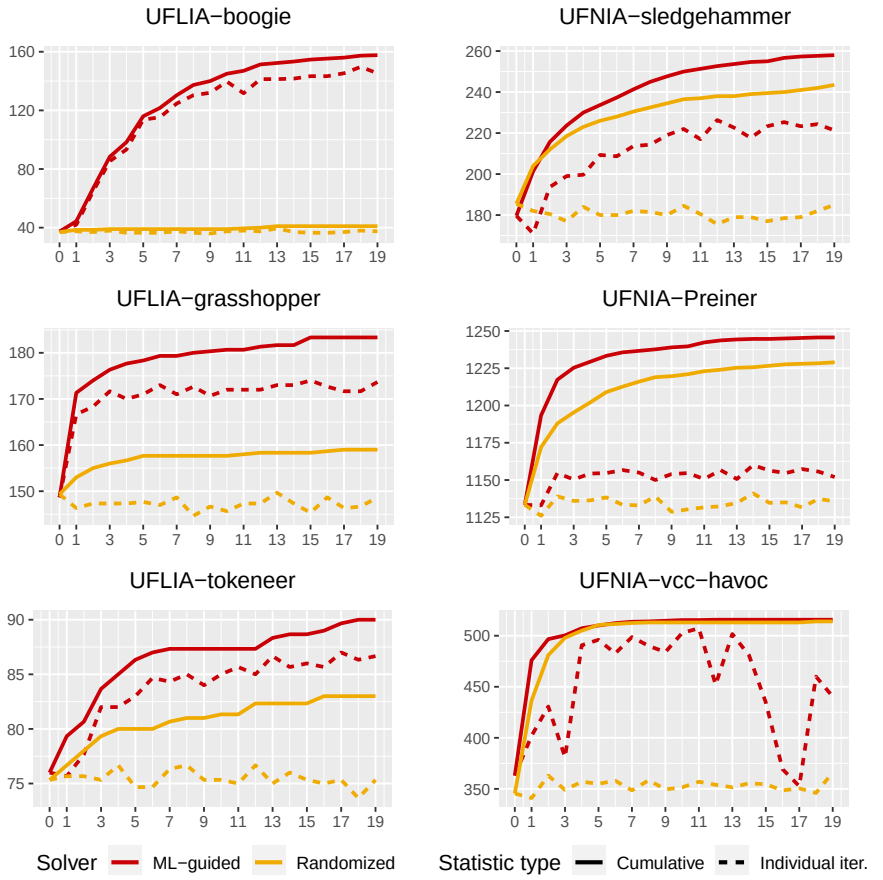


Figure 5.5: Numbers of problems solved (y-axis) in the looping evaluation across twenty iterations (x-axis) for six families. Dashed lines refer to numbers of problems solved in a given iteration; solid lines refer to cumulative number of problems solved in a given iteration and all past iterations. For all the families except UFNIA-vcc-havoc, the ML-guided solver performed better than the randomized, base solver, and for UFLIA-boogie the effect of learning is by far the most pronounced. For UFNIA-vcc-havoc, the ML model has no advantage for the cumulative numbers; moreover, there is an unexpected, large dip in performance for the individual iterations statistic, which shows that the learning dynamics on the self-collected, growing set of training examples may be complex and chaotic.

Table 5.3: Holdout problems solved by the base and ML-guided solvers. Similarly as in Table 5.2, for UFLIA boogie the ML-guided solver achieves very substantial improvement. For two families the results are actually worse compared to the base solver. This may be coused by the slow-down induced by the ML-advisor which was not compensated by the quality of the guidance.

family	base	ML-guided	improvement (%)	number of problems
UFLIA boogie	11	43.0	290.9	251
UFLIA grasshopper	59	66.6	12.9	95
UFLIA tokeneer	25	30.0	20.0	64
UFNIA sledgehammer	63	59.3	−5.8	332
UFNIA Preiner	383	394.3	3.0	1592
UFNIA vcc havoc	175	145.6	−16.8	190

simply too close to the limits of what the solver can do in this configuration and other quantifier instantiation methods need to be also considered (see discussion on the future work in Section 5.6). On the contrary, UFLIA-boogie shows an exceptional improvement with ML guidance and shows no improvement by randomization.

Single-instantiation goal

Here we compare the base solver with the ML-guided solver using the ML model obtained in the last iteration of the evaluation loop. These results are calculated on the *holdout set* – meaning, on a set of problems that were *not* used for training of the ML model. Two types of metrics are considered. First, we consider the number of instantiations that the solver needed to do to solve the given problem – effectively, this is the *abstract time*, measuring the quality of the guidance. Second, we consider the actual CPU time needed to solve the problem. Figure 5.6 shows the results for these two metrics in two separate scatter plots. The results in terms of the numbers of problems solved for different families can be found in Table 5.3.

The results for the number of instantiations clearly speak for ML guidance as the vast majority of the points are below the diagonal. Further, the histograms for the ML-guided solver show a more even distribution of values, whereas the base solver is mainly stacked on timeouts. This indicates that the ML guidance improves the performance across the whole range of the difficulty

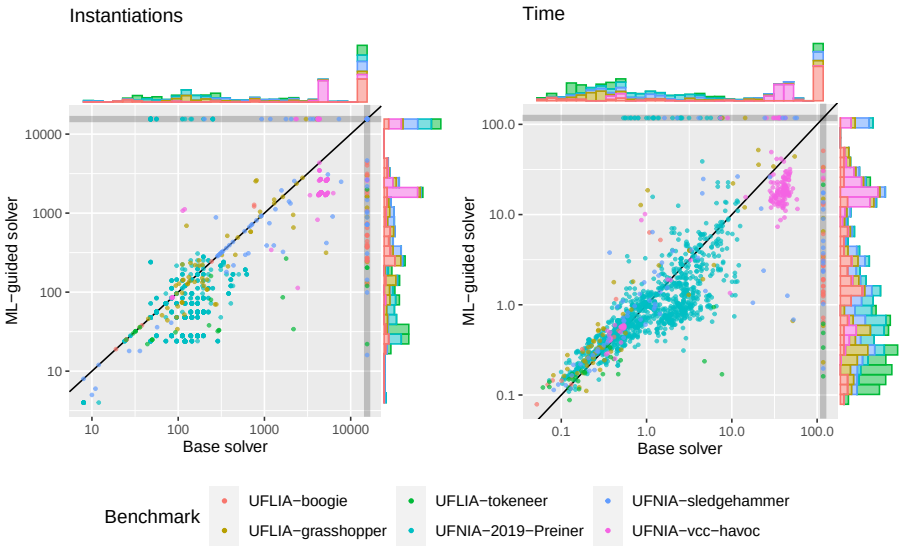


Figure 5.6: Comparison of numbers of instantiations (left) and solving times (right) in log scale, where each point is a testing problem. Points in dark gray stripes were solved by only one of the solvers. For both scatter plots the majority of the points are below the diagonal which clearly shows the advantage of the ML-guided solver.

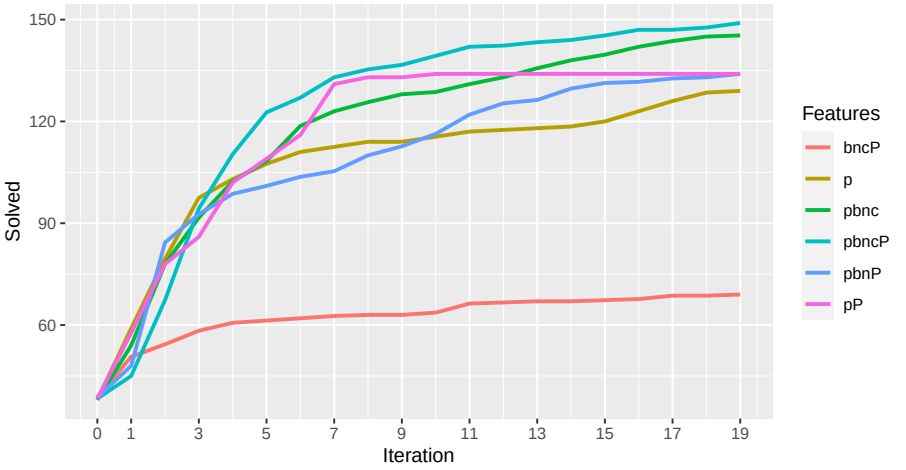


Figure 5.7: Ablation analysis for different sets of features. The dataset used here is UFLIA boogie, and the solid lines refer to the cumulative numbers of solved problems. The letters in the names of runs refer to abbreviations in Table 5.1. Using all the features (pbncP) is the most beneficial. Using the procedural features (p) is crucial: subtracting it from the full set of features reduces the performance dramatically.

of the problems as measured by instantiations / time. In the case of time-evaluation, we still see a large portion of the instances under the diagonal and the histogram is also more tilted towards lower values. However, we also do see some worsening in terms of time. This is not entirely surprising because ML guidance comes at a price because the ML model needs to be evaluated for each considered term in every instantiation step. This is also a likely explanation for the worsening in the havoc family. Nevertheless, given the highly positive results in terms of the number of instantiations (the left part of Figure 5.6), better engineering of the ML guidance has the potential to further improve these results.

Table 5.4: UFLIA boogie problems solved with the ML-guided solver using different sets of features. Using all the available features is the best.

features	solved cumulatively, in target	solved, in holdout
bncP	69.0	13.3
p	126.6	21.0
pP	132.6	27.6
pbnP	134.0	32.0
pbnc	145.3	33.3
pbncP	149.0	36.0

Ablation study

In this subsection we look at the importance of the different types of features that were used to train the ML model (see Subsection 5.3.1). For this we consider the boogie family, where ML guidance had the most effect and therefore enables us to clearly observe the effect of the different features. Since it would be impractical to try all combinations of the features, we use the standard ablation approach, i.e., removing one feature type at a time and observe the effect of this removal. The results are shown in Figure 5.7 and Table 5.4. The ablation study relates two very clear messages. Firstly, the full set of features outperforms all the other configurations. Importantly, the full set of features is also the best one on the unseen problems (holdout set). Secondly, the performance is much worse without the procedural features. This is definitely an interesting observation because the procedural features depend on the previous decisions of the solver, e.g., the number of times a term has been used so far. This observation indicates that is important for the ML guidance to "understand" its old decisions and therefore serves as a guideline for future research.

Training and predicting time

The time of training a single LightGBM model is negligible compared to the solving time and it is in the order of minutes (it grows as more training examples are collected, but it was below 10 minutes for all benchmark sets and iterations).

The total time required to run a loop of 20 training-solving iterations (including the hyper-parameter tuning) on one family depends on the number and complexity of problems in it. We ran all the loops parallelizing across 20 cores.

The total time for running one loop was between 3 and 30 hours. To optimize this, one could apply early stopping when observing diminishing returns.

5.5 Related work

In recent years, machine learning has been widely applied in automated theorem proving. Both gradient boosted trees and graph neural networks have been applied for premise selection and guidance of the automated theorem prover E [72, 130] as well as in a reinforcement learning setting for connection-style provers [83]. ML guidance was also used in the context of SAT solving [115,146]. In the context of SMT, ML has been mostly used *outside* of the solver. ML advice was used to predict the best SMT solver out of a given portfolio and problem [124, 145]. Similarly, FastSMT uses ML to design strategies for the SMT solver Z3 [5], where the BOW representation shows to be most successful, strengthening our choice of this representation.

An unpublished technical report by El Ouraoui et al. [45] describes an attempt to apply ML for quantifier instantiation in the SMT solver veriT [24]. The report demonstrates that ML-guided version of veriT on average decreased the number of generated instances needed to find a proof. It also increased the number of problems solved compared to the unguided, base solver – however, not by much: less than 1% across several families of problems.[5] The approach presented in [45] filters out instantiation terms deemed redundant by the ML model. Further, a different set of features – which are more expensive to compute – is considered. In contrast, we apply stable ordering on the existing terms, which enables us to piggyback on the existing good performance of the solver. Indeed, in our approach, if ML scores some term candidates equally, they are kept in the same order as in the base solver and candidates are never removed from the pool. Our choice of features lets us calculate quickly the ML predictions online without being detrimental to the solving time.

5.6 Conclusions and future work

In this work we design ML guidance of quantifier instantiation in the context of SMT for problems with quantifiers. Quantifiers are a particularly interesting target for ML because they typically cause undecidability and therefore represent an inherent challenge for automated solvers. In the presented approach,

[5] See Tables 10-13 in the report.

the ML advice influences the solver by ordering the candidate terms to be considered for quantifier instantiations. The right choice of instantiations is crucial for solving with quantifiers. Indeed, one particular formula is often solved by a handful of instantiations in one ordering and it times out after hundreds of thousands of instantiations in another. The challenge we are facing here is both conceptual and technological. At the conceptual level, the right set of features needs to be designed. At the technological level, we need an integration of ML predication into the solver that does not hinder the performance of the solver (ML prediction is run on each candidate term).

The experimental evaluation shows that our approach rises to the challenge. When run on a set of formulas, cumulatively ML guidance enables solving substantially more problems than randomizing the solver. Improvements are also seen on a holdout set (a set on which the solver was not trained). ML advice enables us to solve more problems and reduce the number of instantiations needed. The effect is most pronounced in the considered boogie benchmark, where the final ML model enables to solve nearly 3 times more testing problems, and during training it accumulates more than 3 times more solved instances compared to a randomized solver. We also achieve improved performance on the grasshopper, sledgehammer and tokeneer benchmarks. In some families, we have seen worsening in the holdout set, which could partially be explained by the CPU time overhead of running ML prediction. This indicates that it would pay off to better engineer the predictor so that this overhead is reduced. In an ablation study on the boogie benchmark, we show that each of our feature categories contributes to the final results.

This chapter shows that ML has the potential of boosting SMT solving and it opens a number of opportunities for future work. Further ML models may be proposed for specific logics (our method is generic). Direct interaction between quantifiers could be taken into account. Rather than predicting an order of terms on a single variable, the ML model could predict good combinations for tuples of variables. Last but not least, ML advice could be applied to the other quantifier instantiation methods that are in use in the SMT field.

Chapter 6

Online machine learning techniques for Coq*

Abstract

We present a comparison of several online machine learning techniques for tactical learning and proving in the Coq proof assistant. This work builds on top of Tactician, a plugin for Coq that learns from proofs written by the user to synthesize new proofs. Learning happens in an online manner, meaning that Tactician's machine learning model is updated immediately every time the user performs a step in an interactive proof. This has important advantages compared to the more studied offline learning systems: (1) it provides the user with a seamless, interactive experience with Tactician and, (2) it takes advantage of locality of proof similarity, which means that proofs similar to the current proof are likely to be found close by. We implement two online methods, namely approximate k-nearest neighbors based on locality sensitive hashing forests and random decision forests. Additionally, we conduct experiments with gradient

*This chapter is based on a joint work with Liao Zhang, Lasse Blaauwbroek, Prokop Cerný, Cezary Kaliszyk, and Josef Urban. The Tactician framework for machine learning experiments with Coq was implemented by Lasse Blaauwbroek. I was responsible for implementing and evaluating the online random forest algorithm, and I advised Liao Zhang on how to perform experiments with gradient-boosted trees. I wrote Subsection 6.3.2 describing the random forest algorithm, whereas the other sections were written mostly by Lasse Blaauwbroek and Liao Zhang. Josef Urban and Cezary Kaliszyk advised with the work.

boosted trees in an offline setting using XGBoost. We compare the relative performance of Tactician using these three learning methods on Coq's standard library.

6.1 Introduction

The users of interactive theorem proving systems are in dire need of a digital sidekick, which helps them reduce the time spent proving the mundane parts of their theories, cutting down on the man-hours needed to turn an informal theory into a formal one. The obvious way of creating such a digital assistant is using machine learning. However, creating a practically usable assistant comes with some requirements that are not necessarily conducive to the most trendy machine learning techniques, such as deep learning.

The environment provided by ITPs is highly dynamic, as it maintains an ever-changing global context of definitions, lemmas, and custom tactics. Hence, proving lemmas within such environments requires intimate knowledge of all the defined objects within the global context. This is contrasted by – for example – the game of chess; even though the search space is enormous, the pieces always move according to the same rules, and no new kinds of pieces can be added. Additionally, the interactive nature of ITPs demands that machine learning techniques do not need absurd amounts of time and resources to train (unless a pre-trained model is highly generic and widely applicable across domains, something that has not been achieved yet). In this chapter, we are interested in online learning techniques that quickly learn from user input and immediately utilize this information. We do this in the context of the Coq proof assistant [155] and specifically Tactician [19] – a plugin for Coq that is designed to learn from the proofs written by a user and apply that knowledge to prove new lemmas.

Tactician performs a number of functions, such as proof recording, tactic prediction, proof search, and proof reconstruction. In this chapter, we focus on tactic prediction. For this, we need a machine learning technique that accepts as input a database of proofs, represented as pairs containing a proof state and the tactic that was used to advance the proof. From this database, a machine learning model is built. The machine learning task is to predict an appropriate tactic when given a proof state. Because the model needs to operate in an interactive environment, we pose four requirements the learning technique needs to satisfy:

1. The model (datastructure) needs to support dynamic updates. That is, the addition of a new pair of a proof state and tactic to the current model needs to be done in (near) constant time.

2. The model should limit its memory usage to fit in a consumer laptop. We have used the arbitrary limit of 4 GB.

3. The model should support querying in (near) constant time.

4. The model should be persistent (in the functional programming sense [43]). This enables the model to be synchronized with the interactive Coq document, in which the user can navigate back and forth.

6.1.1 Contributions

In this work, we have implemented two online learning models. An improved version of the locality sensitive hashing scheme for k-nearest neighbors is described in detail in Subsection 6.3.1. An implementation of random forest is described in Subsection 6.3.2. In Section 6.4, we evaluate both models, comparing the number of lemmas of Coq's standard library they can prove in a chronological setting (i.e., emulating the growing library).

In addition to the online models, as a proof of concept, we also experiment in an offline fashion with boosted trees, specifically XGBoost [33] in Subsection 6.3.3. Even though the model learned by XGBoost cannot be used directly in the online setting described above, boosted trees are today among the strongest learning methods. Online algorithms for boosted trees do exist [171], and we intend to implement them in the future.

The techniques described here require representing proof states as feature vectors. Tactician already supported proof state representation using simple hand-rolled features [18]. In addition, Section 6.2 describes our addition of more advanced features of the proof states, which are shown to improve Tactician's performance in Section 6.4.

6.2 Tactic and proof state representation

To build a learning model, we need to characterize proof states and the tactics applied to them. To represent tactics, we first perform basic decompositions and simplifications and denote the resulting atomic tactics by their hashes [18].

Tactician's original proof state features [18] consist merely of identifiers and adjacent identifier pairs in the abstract syntax tree (AST). Various other, more

advanced features have been considered for automated reasoning systems built over large formal mathematical knowledge bases [35, 56, 84]. To enhance the performance of Tactician, we modify the old feature set and define new features as follows.

Top-down oriented AST walks We add top-down oriented walks in the AST of length up to 3 with syntax placeholders. For instance, the unit clause $f(g(x))$ will contain the features:

```
f:AppFun, g:AppFun, x:AppArg, f:AppFun(g:AppFun),
g:AppFun(x:AppArg), f:AppFun(g:AppFun(x:AppArg))
```

The feature `g:AppFun` indicates that g is able to act as a function in the term tree, and `x:AppArg` means that x is only an argument of a function.

Vertical abstracted walks We add vertical walks in the term tree from the root to atoms in which nonatomic nodes are substituted by their syntax roles. For the term $f_1(f_2(f_3(a)))$, we can convert each function symbol to `AppFun` whereas the atom a is transformed to `a:AppArg` as above. Subsequently, we can export this as the feature `AppFun(AppFun(AppFun(a:AppArg)))`. Such abstracted features are designed to better capture the overall abstract structure of the AST.

Top-level structures We add top-level patterns by replacing the atomic nodes and substructures deeper than level 2 with a single symbol `X`. Additionally, to separate the function body and arguments, we append the arity of the function to the corresponding converted symbol. As an example, consider the term $f(g(b, c), a)$ consisting of atoms a, b, c, f, g. We first replace a, b, c with `X` because they are of arity 0. We further transform f and g to `X2` according to the number of their arguments. However, b and c break the depth constraint and should be merged to a single `X`. Finally, the concrete term is converted to an abstract structure `X2(X2(X),X)`. Abstracting a term to its top-level structure is useful for determining whether a "logical" tactic should be applied. As an illustration, the presence of $X \wedge X$ in the goal often indicates that we should perform case analysis by the `split` tactic. Since we typically do not need all the nodes of a term to decide such structural information, and we want to balance the generalization with specificity, we use the maximum depth 2.

Premise and goal separation Because local hypotheses typically play a very different role than the conclusion of a proof state, we separate their feature

spaces. This can be done by serially numbering the features and adding a sufficiently large constant to the goal features.

Adding occurrence counts In the first version of Tactician, we have used only a simple boolean version of the features. We try to improve on this by adding the number of occurrences of each feature in the proof state.

6.3 Prediction models

6.3.1 Locality sensitive hashing forests for online k-NN

One of the simplest methods to find correlations between proof states is to define a metric or similarity function $d(x, y)$ on the proof states. One can then extract an ordered list of length k from a database of proof states that are as similar as possible to the reference proof state according to d. Assuming that d does a good job at identifying similar proof states, one can then use tactics known to be useful in a known proof state for an unseen proof state. In this chapter, we refer to this technique as the k-nearest neighbor (k-NN) method (even though this terminology is somewhat overloaded in the literature).

Our distance function is based on the features described in Section 6.2. We compare these features using the Jaccard index $\mathrm{J}(f_1, f_2)$. Optionally, features can be weighted using the tf-idf statistic [78], in which case the generalized index $\mathrm{J}_w(f_1, f_2)$ is used.

$$\mathrm{J}(f_1, f_2) = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|} \quad \text{tf-idf}(x) = \log \frac{N}{|x|_N} \quad \mathrm{J}_w(f_1, f_2) = \frac{\sum_{x \in f_1 \cap f_2} \text{tf-idf}(x)}{\sum_{x \in f_1 \cup f_2} \text{tf-idf}(x)}$$

Here N is the database size, and $|x|_N$ is the number of times feature x occurs in the database. In previous work [18], there is a more detailed comparison of similarity functions.

A naive implementation of the k-NN method is not very useful in the online setting because the time complexity for a query grows linearly with the size of the database. Indexing methods, such as k-d trees, exist to speed up queries [15]. However, these methods do not scale well when the dimensionality of the data increases [63]. In this work, we instead implement an approximate version of the k-NN method based on Locality Sensitive Hashing (LSH) [59]. This is an upgrade of our previous LSH implementation that was not persistent and was slower. We also describe our functional implementation of the method in detail for the first time here.

The essential idea of this technique is to hash feature vectors into buckets using a family of hash functions that guarantee that similar vectors hash to the same bucket with high probability (according to the given similarity function). To find a k-NN approximation, one can simply return the contents of the bucket corresponding to the current proof state. For the Jaccard index, the appropriate family of hash functions are the MinHash functions [27].

The downside of the naive LSH method is that its parameters are difficult to tune. The probability that the vectors that hash to the same bucket are similar can be increased by associating more than one hash function to the bucket. All values of the hash functions then need to pair-wise agree for the items in the bucket. However, this will naturally decrease the size of the bucket, lowering the number of examples k (of k-NN) that can be retrieved. The parameter k can be increased again by simply maintaining multiple independent bucketing datastructures. Tuning these parameters is critically dependent on the size of the database, the length of the feature vectors, and the desired value of k. To overcome this, we implement a highly efficient, persistent, functional variant of Locality Sensitive Hashing Forest [14] (LSHF), which is able to tune these parameters automatically, leaving (almost) no parameters to be tuned manually. Below we give a high-level overview of the algorithm as it is modified for a functional setting. For a more in-depth discussion on the correctness of the algorithm, we refer to the previous reference.

LSHFs consist of a forest (collection) of trees $\mathcal{T}_1 \dots \mathcal{T}_n$. Every tree has an associated hash function h_i that is a member of a (near) universal hashing family mapping a feature down to a single bit (a hash function mapping to an integer can be used by taking the result modulus two). To add a new example to this model, it is inserted into each tree according to a path (sequence) of bits. Every bit of this path can be shown to be locally sensitive for the Jaccard index [14]. The path of an example is calculated using the set of features that represents the proof state in the example.

$$\text{path}_i(f) = \text{sort}(\{h_i(x) \mid x \in f\})$$

For a given tree $\mathcal{T}$, the subtree starting at a given path $b_1 \dots b_m$ can be seen as the bucket to which examples that agree on the hashes $b_1 \dots b_m$ are assigned. Longer paths point to smaller buckets containing less similar examples, while shorter paths point to larger buckets containing increasingly similar examples. Hence, to retrieve the neighbors of a proof state with features f, one should start by finding examples that share the entire path of f. To retrieve more examples, one starts collecting the subtrees starting at smaller and smaller prefixes of $\text{path}_i(f)$. To increase the accuracy and number of examples retrieved, this

procedure can be performed on multiple trees simultaneously, as outlined in Algorithm 5.

Algorithm 5 Querying the Locality Sensitive Hashing Forest

```
1: function QueryLSHF(F, f)                          ▷ F a forest, f a feature set
2:     P ← ⟨path_i(f) : i ∈ [1..|F|]⟩
3:     neighbors ← FilterDuplicates(SimultaneousDescend(F, P))
4:     Optionally re-sort neighbors according to real Jaccard index
5: function SimultaneousDescend(F, P)
6:     F_rel ← ⟨if head(P) then left(T) else right(T) : T ∈ F when ¬ leaf(T) ⟩
7:     F_irrel ← ⟨if leaf(T) then T elseif head(P) then right(T)
8:                else left(T) : T ∈ F ⟩
9:     if F_rel is empty then
10:        neighbors ← empty list
11:    else
12:        P' ← ⟨tail(P_i) : i ∈ [1..n]⟩
13:        neighbors ← SimultaneousDescend(F_rel, P')
14:    if |neighbors| ≥ k then
15:        return neighbors
16:    else
17:        return Append(neighbors, Concat(⟨Collect(T : T ∈ F_irrel⟩)))
```

Tuning the LSHF model consists mainly of choosing the appropriate number of trees that maximizes the speed versus accuracy trade-off. Experiments show that 11 trees is the optimal value. Additionally, for efficiency reasons, it is a good idea to set a limit on the depth of the trees to prevent highly similar examples from creating a deep tree. For our dataset, a maximum depth of 20 is sufficient.

6.3.2 Online random forest

Random forest is a popular machine learning method combining many randomized decision trees into one ensemble, which produces predictions via voting [25]. Even though the decision trees are not strong learners on their own, because they are intentionally decorrelated, the voting procedure greatly improves on top of their individual predictive performance. The decision trees consist of internal nodes labeled by decision rules and leaves labeled by examples. In our case, these are tactics to be applied in the proofs.

Random forest iis a versatile method that requires little tuning of its hyper-parameters. Their architecture is also relatively simple, which makes it easy to provide a custom OCaml implementation easily integrable with Tactician, adhering to its requirement of avoiding mutable data structures. Direct usage of existing random forest implementations is impossible due to challenges in Tactician's learning setting. These challenges are: (1) numerous sparse features, (2) the necessity of online learning, as detailed in the next two paragraphs.

The decision rules in nodes of the decision trees are based on the features of the training examples. These rules are chosen to maximize the *information gain*, i.e., to minimize the *impurity* of the set of labels in the node.[2] There are more than 37 000 binary and sparse features in Tactician. Since the learner integrated with Tactician needs to be fast, one needs to be careful when optimizing the splits in the tree nodes.

Random forests are typically trained in an offline manner where the whole training data is available at the beginning of the training. In Tactician, this would be quite suboptimal. To take advantage of the locality of proof similarity and to be able to use data coming from new proofs written by a user, we want to immediately update the machine learning model after each proof.

There are approaches to turn random forests into online learners [42, 141] which inspired our implementation. The authors of [42] propose a methodology where new training examples are passed to the leaves of the decision trees, and under certain statistical conditions, the leaf is split and converted to a new decision node followed by two new leaves. We take a similar approach, but deciding a split in our implementation is simpler and computationally cheaper.

The pseudocode describing our implementation is presented below. Algorithm 6 shows how the training examples are added to the decision trees. A new training example is passed down the tree to one of its leaves. The trajectory of this pass is governed by binary decision rules in the nodes of the tree. Each rule checks whether a given feature is present in the example. Once the example reaches a leaf, it is saved there, and a decision is made whether to extend the tree (using function SplitCondition). This happens only when the Gini impurity measure [109] on the set of examples in the leaves is greater than a given impurity threshold i (a hyper-parameter of the model). When the split is done, the leaf becomes an internal node with a new split rule, and the collected examples from the leaf are passed down to the two new leaves. The new rule (an output from GenerateSplitRule) is produced in the following way. N features are selected from the features of the examples, where N is the square

[2] If we have labels $\{a, a, b, b, b\}$, ideally, we would like to produce a split which passes all the examples with label a to one side and the examples with b to the other side.

Algorithm 6 Adding training a example e to a decision tree $\mathcal{T}$

```
1: function AddExampleToTree(𝒯, e)
2:     match 𝒯 with
3:         Node(ℛ, 𝒯_l, 𝒯_r):  ▷ ℛ – binary rule, 𝒯_l, 𝒯_r – left and right subtrees
4:             match ℛ(e) with
5:                 Left: return Node(ℛ, AddExampleToTree(𝒯_l, e), 𝒯_r)
6:                 Right: return Node(ℛ, 𝒯_l, AddExampleToTree(𝒯_r, e))
7:         Leaf(l, ℰ):                          ▷ l – label/tactic, ℰ – examples
8:             ℰ ← Append(ℰ, e)
9:             if SplitCondition(ℰ) then
10:                ℛ ← GenerateSplitRule(ℰ)
11:                ℰ_l, ℰ_r ← Split(ℛ, ℰ)
12:                l_l ← label of random example from ℰ_l
13:                l_r ← label of random example from ℰ_r
14:                return Node(ℛ, Leaf(l_l, ℰ_l), Leaf(l_r, ℰ_r))
15:            else
16:                return Leaf(l, ℰ)
```

root of the number of examples. The selection of the features is randomized and made in such a way that features that are distinguishing between the examples have higher probability: First, we randomly select two examples from the leaf, and then we randomly select a feature from the difference of sets of features of the two examples. Among such selected features, the one maximizing the *information gain* [109] of the split rule based on it is selected. The two new leaves get labels randomly selected from the examples belonging to the given leaf.

When adding an example to a random forest (Algorithm 7), first, a decision is made whether a new tree (in the form of a single leaf) should be added to the forest. It happens with probability $\frac{1}{n}$, where n is the number of trees in the forest under the condition that n is lower than a given threshold.

Predicting a tactic for a given example with a random forest (Algorithm 8) is done in two steps. First, the example is passed to the leaves of all the trees and the labels (tactics) in the leaves are saved. Then the ranking of the tactics is made based on their frequencies.

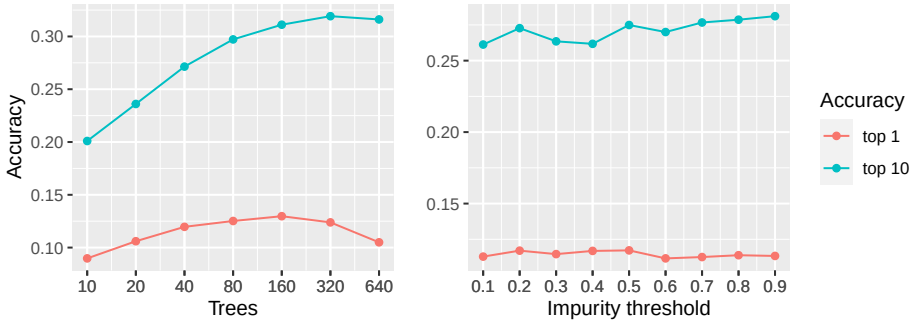
Algorithm 7 Adding a training example e to a random forest $\mathcal{F}$

```
1: function AddExampleToForest(F, e, n_max)  ▷ n_max – max n. of trees
2:     n ← number of trees in F
3:     m ← random number from {1, ..., n}
4:     F_updated ← empty list
5:     if n < n_max and m = 1 then
6:         T ← leaf labeled by tactic used in e
7:         F_updated ← Append(F_updated, T)
8:     for all T ∈ F do
9:         T ← AddExampleToTree(T, e)
10:        F_updated ← Append(F_updated, T)
11:    return F_updated
```

Algorithm 8 Predicting labels for unlabeled e in the random forest $\mathcal{F}$

```
1: function PredictForest(F, e)
2:     P ← empty list                          ▷ P – predictions
3:     for all T ∈ F do
4:         t ← PredictTree(e)
5:         append t to P
6:     R ← Vote(P)                             ▷ R – ranking of tactics
7:     return R
8: function PredictTree(T, e)
9:     match T with
10:        Node(R, T_l, T_r):
11:            match R(e) with
12:                Left: return PredictTree(T_l, e)
13:                Right: return PredictTree(T_r, e)
14:        Leaf(l, E): return l
```

Figure 6.1: Results of hyper-parameter tuning for random forests. Two accuracy metrics are shown: top-10 accuracy (how often the correct tactic was present in the first 10 predictions) and top-1 accuracy (how often the correct tactic coincided with the first prediction).



Tuning hyper-parameters

There are two hyper-parameters in our implementation of random forests: (1) the maximal number of trees in the forest and (2) the impurity threshold for deciding performing the node splits. the influence of these parameters on the predictive power, we perform a grid search. For this, we randomly split the data that is not held out for testing (see Subsection 6.4.1) into a training and validation part. The results of the grid search are shown in Figure 6.1. The best numbers of trees are 160 for top-1 accuracy and 320 for top-10 accuracy, where top-n refers to the frequency of the correct tactic being present in the first n predictions from a machine learning model. We used these two values for the rest of the experiments. For the impurity threshold, it is difficult to see a visible trend in performance; thus we selected 0.5 as our default.

6.3.3 Boosted trees

Gradient boosted decision trees is a state-of-the-art machine learning algorithm that transforms weak base learners, decision trees, into a method with stronger predictive power by appropriate combinations of the base models. One efficient and powerful implementation is the XGBoost library [33]. Here, we perform some initial experiments in an offline setting for tactic prediction. Although XGBoost can at the moment not be directly integrated with Tactician, this gives us a useful baseline based on existing state-of-the-art technology. Below,

we illustrate a procedure of developing our XGBoost model based on binary logistic regression.

The input to XGBoost is a sparse matrix containing rows with the format of (ϕ_P, ϕ_T) where ϕ_P includes the features of a proof state, and ϕ_T characterizes a tactic related to the proof state. We transform each proof state to a sparse feature vector ϕ_P containing the features' occurrence counts. Since there may be a large number of features in a given Coq development environment, which may hinder the efficiency of training and prediction, it is reasonable to decrease the dimension of the vectors. We hash the features to 20 000 buckets by using the modulo of the feature's index. As above, we also remap the tactic hashes to a 20 000-dimensional space separated from the state features.

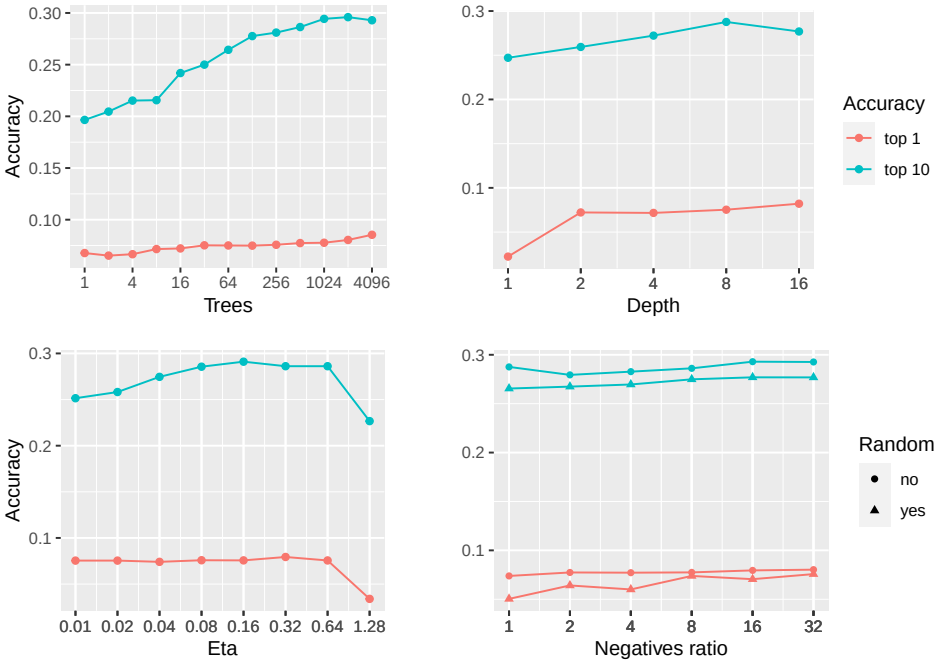
The training examples get labels 1 or 0 based on the tactics being useful or not for the proof state. A tactic for a certain proof state is labeled as positive if it is exactly the one applied to this state in the library. In contrast, negative tactics are elements in the tactic space that differ from the positive instance. We obtain negative data by two approaches: *strong* negatives and *random* negatives. Strong negative instances are obtained by arbitrarily selecting a subset from the best-100 k-NN predictions for this state. In the other approach, negative instances are arbitrarily chosen from the entire tactic space.

With a trained gradient boosted trees model, we can predict the scores of the tactics for an unseen proof state P. First, the top-100 k-NN predictions are preselected. Then, for each tactic, we input (ϕ_P, ϕ_T) to the model to obtain the score of T. The tactics are then sorted according to their scores.

Tuning hyper-parameters

Similarly as for the random forest model (Subsection 6.3.2), we optimize the most important hyper-parameters of the XGBoost training algorithm on the data coming from the non-sink nodes in the dependency graph of Coq's standard library (see Subsection 6.4.1). One essential parameter is the *ratio* of negative examples. Ratio n indicates that we generate n negative instances for each recorded proof state. Other influential parameters that we tune are: *eta* (learning-rate), *number of trees*, and *max depth*. Due to the limitations of computing resources, we assume a set of default parameters: *ratio* $= 8$, *eta* $= 0.2$, *number of trees* $= 500$, *max depth* $= 10$, and then separately modify each of these parameters to observe the influence caused by the change, which is depicted in Figure 6.2. Both strong and random negatives are evaluated. Strong negatives perform better than random negatives, and increasing the negative ratios will likely lead to higher success rates. The figure also shows that a higher

Figure 6.2: Results of hyper-parameter tuning for gradient boosted trees. As in Figure 6.1, two metrics are shown: top-10 and top-1 accuracy. In the lower right pane we also show the results for random negative examples sampling.



number of trees results in better performance. Learning rates that are between 0.08 and 0.64 give good results. It is also apparent that deeper trees (at least 8) increase the accuracy.

Experimental setup

The XGBoost model is evaluated on the task of tactic prediction both in the split setting and the chronological setting (both described in Section 6.4). We use the strong negative examples and determine the final parameters – *ratio* = 16, *eta* = 0.2, *number of trees* = 1024, *max depth* = 10 – for generating a model from non-sink nodes and use that to predict for sink nodes.

Since the entire dataset contains approximately 250 000 proof states, and it is time-consuming to generate a unique XGBoost model for each test case, we propose several ways to speed up the chronological evaluation. Instead of training on the data from all preceding states, we merely provide 1000 instances occurring previously as the training data. According to the results of parameter tuning depicted in Figure 6.2, we decide on the following hyper-parameters to balance the accuracy and efficiency: *ratio* = 4, *eta* = 0.2, *number of trees* = 256, *max depth* = 10.

6.4 Experimental evaluation

To compare the performance of the described machine learning models, we perform three kinds of experiments: *split* evaluation, *chronological* evaluation, and evaluation in Tactician. Achieving good performance in the last type of evaluation is the main goal. All three machine learning models are evaluated in the first two kinds of experiments, while in Tactician we only evaluate k-NN and online random forest. This is because the XGBoost system, while being potentially the strongest machine learner among tested, cannot be easily turned into an online learner and integrated into Tactician. We adopt the original features – terms and term pairs – for evaluation outside Tactician, whereas both the original features and the new are tested on Tactician's benchmark. To determine the relative importance of the feature classes described in Section 6.2, we benchmark the addition of each class separately in Tactician. All evaluations are performed on data extracted from the standard library of Coq 8.11.

Table 6.1: Performance of the three tested machine learning models in two types of evaluation: using the training/testing split of the dataset and the chronological evaluation. Random forest performs substantially better than k-NN and XGBoost in both types of evaluation.

	machine learning system					
	k-NN		random forest		XGBoost	
evaluation type	top-1	top-10	top-1	top-10	top-1	top-10
split	18.8%	34.2%	32.1%	41.2%	18.2%	38.2%
chronological	17.3%	43.7%	29.9%	58.9%	18.2%	43.4%

6.4.1 Split evaluation

In the directed acyclic graph of dependencies of the Coq modules, there are 545 nodes. 104 of them are *sink nodes*, i.e., these are the modules that do not appear among dependencies of any other module. We used these modules as final testing data for evaluation outside Tactician. The rest of the data was randomly split into training and validation parts and was used for parameter tuning of random forest and gradient boosted trees. The models with tuned hyper-parameters were evaluated on the testing data. The results of the evaluation of the three tested models are shown in the first row of Table 6.1.

6.4.2 Chronological evaluation

Although the split evaluation from the previous experiment is interesting, it does not correspond entirely to the Tactician's internal mode of operation. To simulate the real-world scenario in an offline setting, we create an individual model for each proof state by learning from all the previous states – data from dependent files and preceding lines in the local file. The second row of Table 6.1 presents the results of the evaluation in chronological order.

6.4.3 Evaluation in Tactician

Table 6.2 shows the results of the evaluation of two online learners – the k-NN and the random forest – within Tactician. The hyper-parameters of the random forest model were chosen based on the grid search in Subsection 6.3.2. We run the proof search for every lemma in the library with a 40-second time limit on both the original and the improved features.

Table 6.2: Proving performance of two online learners integrated with Tactician, k-NN and random forest, in the Coq standard library. The percentages in the table correspond to the fraction of lemmas proved in a given Coq module. The columns *union* show what fraction of the lemmas was proved by at least one of the learners. RF is an abbreviation of random forest. Random forest performs better than k-NN, but the improvement is not uniform across the Coq modules.

Coq module	**#lemmas**	**features type**					
		original			new		
		k-NN	RF	*union*	k-NN	RF	*union*
all	11 370	33.7%	35.3%	39.6%	34.7%	36.2%	40.4%
Arith	293	52%	59%	65%	56%	59%	66%
Bool	130	93%	87%	93%	92%	88%	92%
Classes	191	80%	76%	81%	79%	79%	83%
FSets	1137	32%	34%	37%	32%	35%	39%
Floats	5	20%	20%	20%	40%	19%	40%
Init	164	73%	51%	73%	73%	56%	73%
Lists	388	38%	43%	47%	38%	44%	49%
Logic	341	31%	27%	34%	32%	31%	35%
MSets	830	38%	40%	43%	36%	40%	43%
NArith	288	37%	43%	44%	35%	42%	47%
Numbers	2198	23%	22%	27%	24%	23%	27%
PArith	280	31%	40%	44%	35%	39%	45%
Program	28	75%	64%	75%	78%	66%	78%
QArith	295	33%	40%	43%	31%	39%	45%
Reals	1756	19%	23%	25%	21%	24%	26%
Relations	37	29%	24%	40%	27%	26%	29%
Setoids	4	100%	100%	100%	100%	97%	100%
Sets	222	43%	42%	49%	49%	47%	53%
Sorting	136	26%	29%	33%	25%	30%	33%
Strings	74	22%	22%	27%	17%	14%	20%
Structures	390	45%	49%	54%	51%	51%	56%
Vectors	37	37%	29%	40%	21%	23%	27%
Wellfounded	36	19%	05%	19%	16%	13%	16%
ZArith	953	41%	46%	49%	40%	43%	46%
btauto	44	11%	20%	20%	20%	17%	22%
funind	4	75%	50%	75%	50%	73%	75%
micromega	339	21%	27%	29%	27%	25%	30%
nsatz	27	33%	33%	37%	40%	26%	40%
omega	37	40%	67%	67%	48%	63%	64%
rtauto	33	30%	39%	48%	33%	44%	51%
setoid_ring	362	21%	23%	26%	27%	27%	30%
ssr	311	68%	55%	69%	70%	57%	71%

Table 6.3: Proving performance of each feature modification. $\mathcal{O}, \mathcal{W}, \mathcal{V}, \mathcal{T}, \mathcal{S}, \mathcal{C}$ denote original features, top-down oriented AST walks, vertical abstract walks, top-level structures, premise and goal separation, and adding feature occurrence, respectively. The symbol $\oplus$ denotes that we combine the original features and a new modification during the experiments.

features	$\mathcal{O}$	$\mathcal{O} \oplus \mathcal{W}$	$\mathcal{O} \oplus \mathcal{V}$	$\mathcal{O} \oplus \mathcal{T}$	$\mathcal{O} \oplus \mathcal{S}$	$\mathcal{O} \oplus \mathcal{C}$
success rate (%)	32.75	32.82	34.16	33.65	34.42	34.97

The random forest performed marginally better than k-NN on both kinds of features. With old features the k-NN proved 3831 lemmas (being 33.7% out of all 11 370), whereas the random forest proved 4011 lemmas (35.3% of all). With the new features, both models performed better, and again, the random forest proved more lemmas (4117, 36.2% of all) than k-NN (3945, 34.7% of all).

It is somewhat surprising that the random forest, which performed much better than k-NN on the split in the offline evaluation, is only better by a small margin in Tactician. This may be related to the time and memory consumption of random forest, which may be higher than for k-NN on certain kinds of data.[3]

It is worth noting that k-NN and random forest resulted in quite different sets of proofs. The columns marked as *union* show that the size of the union of proofs constructed by the two models is substantially larger than the number of proofs found by each model separately. In total, both models resulted in 4503 (39.6%) proofs using old features and 4597 (40.4%) proofs using the new features.

6.4.4 Feature evaluation

Table 6.3 depicts the influence of adding the new classes of features described in Section 6.2 to the previous baseline.[4] All of the newly produced features improve the success rates. However, the top-down oriented AST walks contribute little, probably due to Tactician having already included term tree walks up to length 2. Every other modification obtains a reasonable improvement, which confirms the intuitions described in Section 6.2.

[3]Splitting the leaves has quadratic time complexity with respect to the number of examples stored in the leaf; sometimes it happens, that leaves of the trees store large number of examples.

[4]The results here are not directly comparable to those in Table 6.2 mainly due to the usage of a non-indexed version of k-NN in contrast to Algorithm 5.

6.5 Related work

Random forests were first used in the context of theorem proving by Färber [47], where multi-path querying of a random forest would improve on k-NN results for premise selection. This work, however, did not integrate the machine-learned advisor into a proof assistant as we do, but instead performed an external evaluation using an automated theorem prover. Nagashima and He [113] proposed a proof method recommendation system for Isabelle/HOL based on decision trees on top of precisely engineered features. A small number of trees and features allowed for explainable recommendations. However, their implementation do not allow for incremental updating the trees with new examples. Frameworks based on random boosted trees (XGBoost, LightGBM) have also been used in automated reasoning, in the context of guiding tableaux connection proof search [83] and the superposition calculus proof search [35], as well as for handling negative examples in premise selection [127].

Machine learning to predict tactics was first considered by Gauthier et al. [56] in the context of the HOL4 theorem prover. His later improvements [56] added Monte-Carlo tree search, tactic orthogonalization, and integration of both Metis and a hammer [54]. In this line of work, for tactic prediction a k-NN algorithm was used. A similar system for HOL Light, where deep learning is used, was developed by Bansal et al. [7]. Nagashima and Kumar developed the proof search component [114] of such a system for Isabelle/HOL. This work builds upon Tactician [18,19], adapting and improving these works for dependent type theory and the Coq proof assistant.

6.6 Conclusions and future work

We have implemented several new methods for learning tactical guidance of Coq proofs in the Tactician system. This includes better proof state features and an improved version of approximate k-nearest neighbor based on locality sensitive hashing forests. A completely new addition is our online implementation of random forest in Coq, which can now be used instead of or together with the k-nearest neighbor. We have also started to experiment with strong state-of-the-art learners based on gradient boosted trees, so far in an offline setting using binary learning with negative examples.

Our random forest improves substantially on the k-nearest neighbor in an offline accuracy-based evaluation. In an online theorem-proving evaluation, the improvement is not as big, possibly due to the speed of the two methods and the importance of backtracking during the proof search. The methods are, however,

quite complementary and running both of them in parallel increases the overall performance of Tactician from 33.7% (k-NN with the old features) to 40.4% in 40 s. Our best new method (RF with the new features) now solves 36.2% of the problems in 40 s.

The offline experiments with gradient boosted trees are so far inconclusive. They outperform k-nearest neighbor in top-10 accuracy, but the difference is small, and the random forest performs much better in this metric. Since the random forest learns only from positive examples, this likely shows that learning in the binary setting with negative examples is challenging on our Tactician data. In particular, we likely need good semantic feature characterizations of the tactics, obtained, e.g., by computing the difference between the features of the proof states before and after the tactic application. The experiments, however, already confirm the importance of choosing good negative data to learn from in the binary setting.

Chapter 7

Machine-learned premise selection for Lean*

Abstract

We introduce a machine-learning-based tool for the Lean proof assistant that suggests relevant premises for theorems being proved by a user. The design principles for the tool are (1) tight integration with the proof assistant, (2) ease of use and installation, (3) a lightweight and fast approach. For this purpose, we designed a custom version of the random forest model, trained in an online fashion. It is implemented directly in Lean, which was possible thanks to the rich and efficient metaprogramming features of Lean 4. The random forest is trained on data extracted from mathlib – Lean's mathematics library. We experiment with various options for producing training features and labels. The advice from a trained model is accessible to the user via the `suggest_premises` tactic which can be called in an editor while constructing a proof interactively.

*This chapter is based on a join work with Ramon Fernández Mir and Edward Ayers. I led the project. I implemented and evaluated custom versions of the random forest and k-nearest neighbours algorithms. Ramon Fernández Mir implemented the data extraction tool. Edward Ayers implemented the interface allowing to use the tool in Visual Studio Code interactively. The text, except Section 7.2, was written by me, and it was complemented and improved together with the other authors.

7.1 Introduction

Formalizing mathematics in proof assistants is an ambitious and hard undertaking. One of the major challenges in constructing formal proofs of theorems depending on multiple other results is the prerequisite of having a good familiarity with the structure and contents of the library. Tools for helping users search through formal libraries are currently limited.

In the case of Lean proof assistant [37], users may look for relevant lemmas in its formal library, mathlib [104], either by (1) using general textual search tools and keywords, (2) browsing the related source files manually, (3) using mathlib's `suggest` or `library_search` tactics.

Approaches (1) and (2) are often slow and tedious. The limitation of approach (3) is the fact that `suggest` or `library_search` propose premises that strictly match the goal at the current proof state. This is often very useful, but it also means that these tactics often fail to direct the user to relevant lemmas not exactly matching the current goal. They may also suggest too many trivial lemmas if the goal is simple. Sometimes they are unable to help, like in proofs by contradiction where the only unchanging goal is simply `False`.

The aim of this project is to make progress towards improving the situation of a Lean user looking for relevant lemmas while building a proof. We develop a new tool that efficiently computes a ranking of potentially useful lemmas selected by a machine learning (ML) model trained on data extracted from mathlib. This ranking can be accessed and used interactively via the `suggest_premises` tactic.

The project described here belongs to the already quite broad body of work dealing with the problem of fact selection for theorem proving [1,71,80,97,108, 127,130]. This problem, commonly referred to as the *premise selection* problem, is crucial when performing automated reasoning in large formal libraries – both in the context of *automated* (ATP) and *interactive* (ITP) theorem proving, and regardless of the underlying logical calculus. Most of the existing work on premise selection focuses on the ATP context. Our main contribution is the development of a premise selection tool that is practically usable in a proof assistant (Lean in that case), tightly integrated with it, lightweight, extendable, and equipped with a convenient interface. The tool is available in a public GitHub repository: `https://github.com/BartoszPiotrowski/lean-premise-selection`.

7.2 Dataset collection

A crucial requirement of a useful ML model is a high-quality dataset of training examples. It should represent the learning task well and be suitable for the ML architecture being applied.

In this work, we use simple ML architectures that cannot process raw theorem statements and require *featurization* as a preprocessing step. The features need to be meaningful yet simple so that the model can use them appropriately. Our approach is described in Subsection 7.2.1. The notion of *relevant premise* may be understood differently depending on the context. In Subsection 7.2.2, we describe three specifications of this notion that we used in our experiments.

This project is implemented in Lean 4 but, at the time of writing, Lean 4's library is being ported from Lean 3, so we use mathlib3port[2] as our main data source.

7.2.1 Features

The features, similar to those used in [74, 127], consist of the symbols used in the theorem statement with different degrees of structure. In particular, three types of features are used: `names`, `bigrams` and `trigrams`. As an illustration, consider this theorem from `Algebra/GroupWithZero/Units/Basic.lean`:

```
theorem div_ne_zero (ha : a ≠ 0) (hb : b ≠ 0) : a / b ≠ 0 := ...
```

The most basic form of featurization is the bag-of-words model, where we simply collect all the `names` (and numerical constants) involved in the theorem. The ones starting with `T` appear in the conclusion, while the ones starting with `H` appear in the hypotheses.

```
H:OfNat.ofNat H:MonoidWithZero.toZero H:0 H:Ne T:HDiv.hDiv T:0 T:Ne ...
```

It would be desirable, however, to keep track of which symbols appear next to each other in the syntactic trees of the theorem hypotheses and its statement. Thus, we extract `bigrams` that are formed by the head symbol and each of its arguments (separated by `/` below).

```
H:Ne/OfNat.ofNat H:OfNat.ofNat/0 T:OfNat.ofNat/0 T:Ne/OfNat.ofNat ...
```

Similarly, we also consider `trigrams`, taking all paths of length 3 from the syntactic tree of the expression:

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

Table 7.1: Filters' statistics. An example is a theorem with a non-empty list of premises.

	`all`	`source`	`math`
total premises	96 915	28 784	67 462
total examples	41 755	20 571	40 187
premises per example	3.12	2.35	2.09

```
H:Ne/OfNat.ofNat/0 H:Ne/OfNat.ofNat/Zero.toOfNat0
T:HDiv.hDiv/instHDiv/GroupWithZero.toDiv T:Ne/HDiv.hDiv/OfNat.ofNat ...
```

7.2.2 Relevant premises

To obtain the list of all the premises used in the proof term it suffices to traverse the Lean expression and keep track of all the constants whose type is a proposition. For instance, the raw list of premises that appear in the proof of `le_of_pred_lt` is:

```
GroupWithZero.noZeroDivisors
mul_ne_zero
inv_ne_zero
Eq.refl
div_eq_mul_inv
```

This approach, however, results in a large number of premises including lemmas used *implicitly* by tactics, or simple facts that a user would rarely write explicitly. Three different filters are applied to mitigate this issue: `all`, `source`, and `math`. They are described below. Their overall effect is shown in Table 7.1.

1. The `all` filter preserves almost all premises from the original, raw list, removing those that were generated automatically by Lean. They can be recognized by containing a leading underscore in their names, e.g., `RingTheory.MatrixAlgebra._auxLemma.1`. In our example, there is no such a premise. Examples from this filter are not appropriate for training an ML advisor for interactive use as the suggestions would contain many lemmas used implicitly by tactics. Yet, such an advisor could be used for automated ITP approaches such as *hammers* [22].

2. The `source` filter leaves only those premises that appear in the proof's source code. The idea is to model the explicit usage of premises by a user.

Following our example, we would take the following proof as a string and list only the three mathlib premises appearing there:

```
by rw [div_eq_mul_inv]; exact mul_ne_zero ha (inv_ne_zero hb)
```

3. The `math` filter preserves only lemmas that are clearly of mathematical nature, discarding basic, technical ones. The names of all theorems and definitions from mathlib are extracted and used as a *white list*. In particular, this means that many basic lemmas from the Core library of Lean (like, e.g., `rfl`, `congr_arg`) are filtered out. In our running example, the premise `Eq.refl` would be removed.

7.3 Machine learning models

The task modelled here with ML is predicting a ranking of likely useful premises (lemmas and definitions) conditioned by the features of the statement of a theorem being proved by a user. The nature of this problem is different than common applications of classical ML: both the number of features and labels to predict (premises) is large, and the training examples are sparse in the feature space. Thus, we could not directly rely on the traditional implementations of ML algorithms, and using custom-built versions was necessary. As one of our design requirements was tight integration with the proof assistant, we implemented the ML algorithms directly in Lean 4, without a need to call external tools. This also served as a test for the maturity and efficiency of Lean 4 as a programming language.

In Subsections 7.3.1 and 7.3.2 we describe two ML algorithms implemented in this work: *k-nearest neighbours* and *random forest*.

7.3.1 k-nearest neighbours

This is a classical and conceptually simple ML algorithm [65], which has already been used multiple times for premise selection [21,80,82]. It belongs to the *lazy learning* category, meaning that it does not result in a prediction model trained beforehand on the dataset, but rather the dataset is an input to the algorithm while producing the predictions.

Given an unlabeled example, k-NN produces a prediction by extracting the labels of the k most similar examples in the data set and returning an averaged (or most frequent) label. In our case, the labels are lists of premises. We

compose multiple labels into a ranking of premises according to the frequency of appearance in the concatenated labels.

The similarity measure in the feature space calculates how many features are shared between the two data points, but additionally puts more weight on those features that are more rare in the whole training data set $\mathcal{D}$. The formula for the similarity of the two examples x_1 and x_2 associated with sets of features f_1 and f_2, respectively, is given below.

$$M(x_1, x_2) = \frac{\sum_{f \in f_1 \cap f_2} t(f)}{\sum_{f \in f_1} t(f) + \sum_{f \in f_2} t(f) - \sum_{f \in f_1 \cap f_2} t(f)},$$

where

$$t(f) = \log\left(\frac{|\mathcal{D}|}{|\mathcal{D}_f|}\right)^2,$$

where $\mathcal{D}_f$ are those training examples that contain the feature f.

The advantages of k-NN are its simplicity and the lack of training. A disadvantage, however, is the need to traverse the whole training data set in order to produce a single prediction (a ranking). This may be slow, and thus not optimal for interactive usage in proof assistants.

7.3.2 Random forest

As an alternative to k-NN, we use *random forest* [25] – an ML algorithm from the *eager learning* category, with a separate training phase resulting in a prediction model.

Random forest uses a collection of decision trees. The leaves of the trees contain labels, and their nodes decision rules based on the features. 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. The trees are trained in such a way as to avoid correlations between them, and the averaged prediction from them is of better quality than the prediction from a single tree.

Our version of random forest, adapted to deal with sparse binary features and a large number of labels, is similar to the one used in [172]. It is trained in the *online* manner, i.e., it is updated sequentially with single training examples – not with the entire training data set at once, as is typically done. The rationale for this is to make it easy to update the model with data coming from new

theorems proved by a user. This allows the model to immediately provide suggestions taking into account these recently added theorems.

Algorithm 9 provides a sketch of how a training example updates a tree – for all the details see the actual implementation in our public GitHub repository.[3] A crucial part of the algorithm is the MakeSplitRule function creating node splitting rules. Searching for the rules resulting in optimal splits would be costly, thus this function relies on heuristics.

Figure 7.1 schematically depicts how a simple decision tree from a trained random forest predicts a set of premises for an input example.

Algorithm 9 Updating the tree T with the training example e in a random forest.

```
1:  function AddExampleToTree(T, e)
2:      match T with
3:          Node(R, T_l, T_r): ▷ R – binary rule, T_l, T_r – left and right subtrees
4:              match R(e) with ▷ passing example e down the tree to a leaf
5:                  Left: return Node(R, AddExampleToTree(T_l, e), T_r)
6:                  Right: return Node(R, T_l, AddExampleToTree(T_r, e))
7:          Leaf(E): ▷ E – examples stored in the leaf
8:              E ← Append(E, e)
9:              if SplitCondition(E) then ▷ testing if the leaf should be split
10:                 R ← MakeSplitRule(E) ▷ new semi-optimized split rule
11:                 E_l, E_r ← Split(R, E) ▷ splitting examples into two parts
12:                 return Node(R, Leaf(E_l), Leaf(E_r)) ▷ new subtree
13:             else
14:                 return Leaf(E) ▷ the original leaf with the new example e
```

7.4 Evaluation setup and results

To assess the performance of the ML algorithms, the data points extracted from mathlib were split into *training* and *testing* sets. The testing examples come from the modules that are *not* dependencies of any other modules (there are 592 of them). This simulates a realistic scenario in which a user utilizing the suggestion tool develops a new mathlib module. The rest of the modules (2436) served as the source of training examples.

[3]The decision tree implementation is in a file `PremiseSelection/Tree.lean`

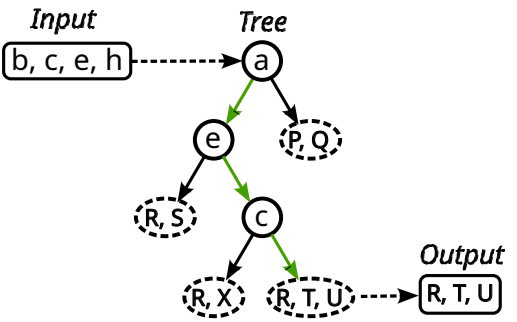


Figure 7.1: A schematic example of a decision tree from a trained random forest. Lowercase letters (`a`, `b`, `c`, ...) designate features of theorem statements, whereas uppercase letters (`P`, `Q`, `R`, ...) designate names of premises. The input (a featurized theorem statement) is being passed down the tree (along the green arrows) in such a way that each node tests for a presence of a certain single feature, and passes the input example to the left (or right) sub-tree in the negative (or positive) case. The output is a set of premises in the reached leaf.

Two measures of the quality of the rankings produced by ML are defined: Cover and Cover$_+$. 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, these measures are defined as follows:

$$\text{Cover}(T) = \frac{\left|P \cap R\texttt{[:}n\texttt{]}\right|}{n}, \qquad \text{Cover}_+(T) = \frac{\left|P \cap R\texttt{[:}n+10\texttt{]}\right|}{n},$$

where $R\texttt{[:}k\texttt{]}$ is a set of k initial premises from ranking R. Both Cover and Cover$_+$ return values in $[0, 1]$. Cover gives the score of 1 only for a "perfect" prediction where the premises actually used in the proof form an initial segment of the ranking. Cover$_+$ may also give a perfect score to less precise predictions. The rationale for Cover$_+$ is that the user in practice may look through 10 or more suggested premises. This is often more than the n premises actually used in the proof, so we consider initial segments of length $n + 10$ in Cover$_+$.

Both k-NN and random forest are evaluated on data subject to all three premise filters described in Subsection 7.2.2. For each of these variants of data, three combinations of features are tested: (1) `names` only, (2) `names` and `bigrams`, (3) `names`, `bigrams`, and `trigrams`. The hyper-parameters for the ML algorithms were selected by an experiment on a smaller data set. For k-NN, the number of neighbours was fixed to 100. For random forest, the number of

trees was set to 300, each example was used for training a particular decision tree with probability 0.3, and the training algorithm passed through the whole training data 3 times.

Table 7.2 shows the results of the experiment. In terms of the Cover metric, random forest performed better than k-NN for all data configurations. However, for Cover_+ metric, k-NN surpassed random forest for the `math` filter.

It turned out that the union of `names` and `bigrams` constitutes the best features for all the filters and both ML algorithms. It likely means that the more complex `trigrams` did not help the algorithms to generalize well but rather caused *over-fitting* on the training set.[4]

The results for the `all` filter appear to be much higher than for the other two filters. However, this is because applying `all` results in many simple examples containing just a few common, basic premises (e.g., just a single `rfl` lemma). They increase the average score.

Overall, the random forest with `names` + `bigrams` features gives the best results. 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 random forest and k-NN, respectively.

The evaluation may be reproduced by following the instructions in the source code available at `https://github.com/BartoszPiotrowski/lean-premise-selection#reproducing-evaluation`.

7.5 Interactive tool

The ML predictor is wrapped in an interactive tactic `suggest_premises` that users can type into their proof script. It will invoke the predictor and produce a list of suggestions. This list 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 as shown in Figure 7.2 where the user can select the resulting tactic to insert into the proof script. By using

[4] It means that `trigrams`, being more specific features than the other two types, likely exposed spurious correlations in the training data that were exploited by the machine learning model.

Table 7.2: Average performance of random forest and k-NN on testing data, for three premises filters and three kinds of features. The type of features is indicated by a one-letter abbreviation: `n` = `names`, `b` = `bigrams`, `t` = `trigrams`. For each configuration, Cover and Cover+ measures are reported (the latter in brackets). In each row, the best Cover result is bolded. The random forest consistently performed better than k-NN. Using the combination of `names` and `bigrams` as features was optimal.

	machine learning model					
	random forest			k-nearest neighbours		
premises	`n`	`n+b`	`n+b+t`	`n`	`n+b`	`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)

an asynchronous approach, we can display results rapidly without needing to wait for a slow tactic search to complete.

7.6 Future work

The results may be improved by augmenting the dataset with all intermediate tactic states, as well as developing better features, utilizing the well-defined structure of Lean expressions.

Applying modern neural architectures in place of the simpler ML algorithms used here is a promising path [157]. It would depart from our philosophy of a lightweight, self-contained approach as the suggestions would come from an external tool, possibly placed on a remote server. However, given the strength of the current neural architectures, we could hope for higher-quality predictions. Moreover, neural models do not require hand-engineered features.

Finally, premise selection is an important component of ITP *hammer* systems [22]. The presented tool may be readily used for a hammer in Lean, which has not yet been developed.

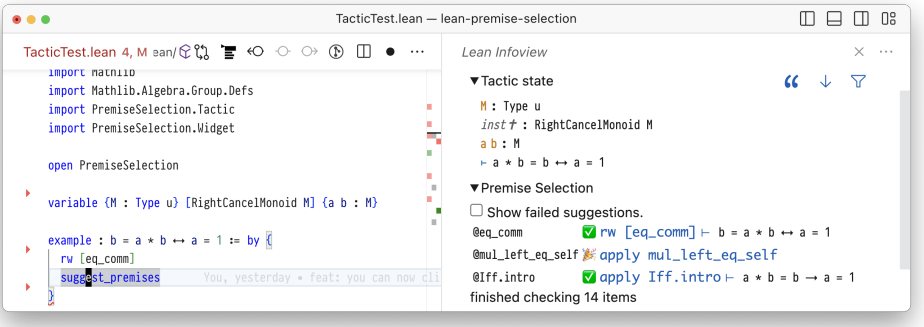


Figure 7.2: The interactive tool in Visual Studio Code. The left pane shows the source file with the cursor over a `suggest_premises` tactic. The right pane shows the goal state at the cursor position and, below, the suggested lemmas to solve the goal. Suggestions annotated with a checkbox advance the goal state, suggestions annotated with confetti close the current goal. Clicking on a suggested tactic (e.g. `apply mul_left_eq_self`) automatically appends to the proof script on the left.

Chapter 8

Symbolic rewriting with neural networks*

Abstract

This work investigates if the current neural architectures are adequate for learning symbolic rewriting. Two kinds of data sets are proposed for this research – one based on automated proofs and the other being a synthetic set of polynomial terms. The experiments with use of the current neural machine translation models are performed and its results are discussed. Ideas for extending this line of research are proposed, and its relevance is motivated.

8.1 Introduction

Neural networks (NNs) turned out to be very useful in several domains. In particular, one of the most spectacular advances achieved with use of NNs has been natural language processing. One of the tasks in this domain is a translation between natural languages – neural machine translation (NMT) systems established here the state-of-the-art performance. Recently, NMT produced first

*This chapter is based on a joint work with Josef Urban, Chad Brown and Cezary Kaliszyk. I was responsible for performing all the experiments and creating the polynomial data set. Chad Brown produced the AIM data set. I wrote the whole text. Josef Urban and Cezary Kaliszyk were advising with the work.

encouraging results in the *autoformalization task* [85, 86, 87, 167] where given an *informal* mathematical text in LaTeX the goal is to translate it to its *formal* (computer understandable) counterpart. In particular, the NMT performance on a large synthetic LaTeX-to-Mizar dataset produced by a relatively sophisticated toolchain developed for several decades [6] is surprisingly good [167], indicating that neural networks can learn quite complicated algorithms for symbolic data. This inspired us to pose a question: *Can NMT models be used in the formal-to-formal setting?* In particular: *Can NMT models learn symbolic rewriting?*

The answer is relevant to various tasks in automated reasoning. For example, neural models could compete with symbolic methods such as inductive logic programming (ILP) [112] that have been previously experimented with to learn simple rewrite tasks and theorem-proving heuristics from large formal corpora [158]. Unlike (early) ILP, neural methods can, however, easily cope with large and rich datasets without combinatorial explosion.

This work is also an inquiry into the capabilities of NNs as such, in the spirit of works like [46].

The main contributions of our project are:

1. providing two novel reasoning datasets: one extracted from real data generated by a theorem prover run on challenging problems and one synthetic,

2. investigating the capabilities of neural networks on these datasets in a reproducible and practical setting, i.e., using modest computational resources and moderate number of training examples.

8.2 Data

We prepared two data sets for our experiments – the first consists of examples extracted from proofs found by ATP (automated theorem prover) in a mathematical domain (AIM loops), whereas the second is a synthetic set of polynomial terms. We characterize both of them in detail below.

8.2.1 The AIM data set

The data consists of sets of ground and non-ground rewrites that came from Prover9[2] proofs of theorems about AIM loops produced by Veroff [89].

[2] `https://www.cs.unm.edu/~mccune/prover9/`

Table 8.1: Example of a ground rewrite in the AIM data set.

rewrite rule	`b(s(e, v1), e) = v1`
before rewriting	`k(b(s(e, v1), e), v0)`
after rewriting	`k(v1, v0)`

Table 8.2: Example of a non-ground rewrite in the AIM data set.

rewrite rule	`o(V0, e) = V0`
before rewriting	`t(v0, o(v1, o(v2, e)))`
after rewriting	`t(v0, o(v1, v2))`

Many of the inferences in the proofs are paramodulations from an equation and have the form

$$\frac{s = t \qquad u[\theta(s)] = v}{u[\theta(t)] = v}$$

where s, t, u, v are terms and θ is a substitution. For the most common equations $s = t$, we gathered corresponding pairs of terms $\big(u[\theta(s)], u[\theta(t)]\big)$ which were rewritten from one to another with $s = t$. We put the pairs to separate data sets (depending on the corresponding $s = t$): in total 8 data sets for ground rewrites (where θ is trivial) and 12 for non-ground ones. The goal will be to learn rewriting for each of these 20 rules separately.

Terms in the examples are treated as linear sequences of tokens where tokens are single symbols (variable / constant / predicate names, brackets, commas). Numbers of examples in each of the data sets vary between 251 and 34 101. Lengths of the sequences of tokens vary between 1 and 343, with the mean around 35. These 20 data sets were split into training, validation and test sets for our experiments (60%, 10%, 30%, respectively).

In Table 8.1 and Table 8.2 there are presented examples of pairs of AIM terms in TPTP [152] format, before and after rewriting with, respectively, ground and non-ground rewrite rules.[3]

[3]All the described AIM data are available at `https://github.com/BartoszPiotrowski/rewriting-with-NNs/tree/master/data/AIM`

Table 8.3: Examples in the polynomial data set.

before rewriting	after rewriting
`(x*(x+1))+1`	`x^2+x+1`
`(2*y)+(1+(y*y))`	`y^2+2*y+1`
`(x+2)*(((2*x)+1)+(y+1))`	`2*x^2+5*x+y+3`

8.2.2 The polynomial data set

This is a synthetically created data set where the examples are pairs of equivalent polynomial terms. The first element of each pair is a polynomial in an arbitrary form, and the second element is the same polynomial in a normalized form. The arbitrary polynomials are created randomly in a recursive manner from a set of available (non-nullary) function symbols, variables and constants. First, one of the symbols is randomly chosen. If it is a constant or a variable, it is returned and the process terminates. If a function symbol is chosen, its subterm(s) are constructed recursively in a similar way.

The parameters of this process are set in such a way that it creates polynomial terms of average length around 25 symbols. Terms longer than 50 are filtered out. Several data sets of various difficulties were created by varying the number of available symbols. These were quite limited – at most 5 different variables and constants being a few first natural numbers. The reason for this limited complexity of the input terms is because normalizing even a relatively simple polynomial can result in a very long term with very large constants – which is related especially to the operation of exponentiation in polynomials.

Each data set consists of different 300 000 examples – see Table 8.3 for examples. These data sets were split into training, validation and test sets for our experiments ($60\%, 10\%, 30\%$, respectively).[4]

8.3 Experiments

For experiments with both data sets, we used an established NMT architecture [102] based on LSTMs (long short-term memory cells) and implementing the attention mechanism.[5]

[4]The described polynomial data are available at `https://github.com/BartoszPiotrowski/rewriting-with-NNs/tree/master/data/polynomial`

[5]We also experimented with the transformer model [164] but the results were worse. This could be due to a limited grid search we performed as transformer is known to be very sensitive

After a small grid search, we decided to inherit most of the hyper-parameters of the model from the best results achieved in [167] where LaTeX-to-Mizar translation is learned. We used relatively small LSTM cells consisting of 2 layers with 128 units. The "scaled Luong" version of the attention mechanism was used, as well as dropout with rate equal 0.2. The number of training steps was 10 000. This setting was used for all our experiments described below.

8.3.1 AIM data set

First, NMT models were trained for each of the 20 rewrite rules in the AIM data set. It turned out that the models, as long as the number of examples was greater than 1000, were able to learn the rewriting task very well, reaching 90% of accuracy on separated test sets. This means that the task of applying a single rewrite step seems relatively easy to learn by NMT. See Table 8.4 for all the results.

We also run an experiment on the joint set of all rewrite rules (consisting of 41 396 examples). Here the task was more difficult as a model needed not only to apply rewriting correctly, but also choose "the right" rewrite rule applicable for a given term. Nevertheless, the performance was also very good, reaching 83% accuracy.

8.3.2 Polynomial data set

Then experiments on more challenging but also much larger data sets for polynomial normalization were performed. Depending on the difficulty of the data, accuracy on the test sets achieved in our experiments varied between 70% and 99%. The results in terms of accuracy are shown in Table 8.5.

This high performance of the model encouraged a closer inspection of the results. First, we checked if in the test sets there are input examples which differ from these in training sets only by renaming of variables. Indeed, for each of the data sets in test sets are 5–15% of such "renamed" examples. After filtering them out, the measured accuracy drops – but only by 1–2%.

An examination of the examples wrongly rewritten by the model was done. It turns out that the wrong outputs almost always parse (in 97–99% of cases, they are legal polynomial terms). Notably, depending on the difficulty of the data set, as much as 18–64% of incorrect outputs are wrong only with respect to the constants in the terms. (Typically, the NMT model proposes too low

to hyper-parameters.

Table 8.4: Results of experiments with AIM data. (Names of the rules correspond to folder names in the GitHub repo.) Low number of training examples implies low predictive performance.

rule	training examples	test examples	accuracy on test
`abstrused1u`	2472	1096	86.5%
`abstrused2u`	2056	960	89.2%
`abstrused3u`	1409	666	84.3%
`abstrused4u`	1633	743	87.4%
`abstrused5u`	2561	1190	89.5%
`abstrused6u`	81	40	12.5%
`abstrused7u`	76	37	0.0%
`abstrused8u`	79	39	2.5%
`abstrused9u`	1724	817	86.7%
`abstrused10u`	3353	1573	82.9%
`abstrused11u`	10230	4604	79.0%
`abstrused12u`	7201	3153	87.2%
`instused1u`	198	97	20.6%
`instused2u`	196	87	25.2%
`instused3u`	83	41	29.2%
`instused4u`	105	47	2.1%
`instused5u`	444	188	59.5%
`instused6u`	1160	531	87.5%
`instused7u`	307	144	13.8%
`instused8u`	116	54	3.7%
union of all	41396	11826	83.2%

Table 8.5: Chosen results of experiments with polynomials. Characteristic of formulas concerns the *input* polynomials. (Labels of the data sets correspond to folder names in the GitHub repo). The more complex input polynomials, the lower the predictive performance of the NMT model.

label	function symbols	constant symbols	number of variables	accuracy on test
`poly1`	$+, *$	$0, 1$	1	99.2%
`poly2`	$+, *$	$0, 1$	2	97.4%
`poly3`	$+, *$	$0, 1$	3	88.2%
`poly4`	$+, *$	$0, 1, 2, 3, 4, 5$	5	83.4%
`poly5`	$+, *, \hat{}$	$0, 1$	2	85.5%
`poly6`	$+, *, \hat{}$	$0, 1, 2$	3	71.8%

constants compared to the correct ones.) Below 1% of wrong outputs is correct modulo variable renaming.

8.4 Integration data sets from Facebook Research experiments

The work of [98] joined recently the line of research pursued by us here:[6] applying sequence-to-sequence neural architectures to symbolic rewriting tasks. In case of [98], the symbolic tasks are integration of a chosen set of functions (basically polynomials plus trigonometry, exp and log) and solving a class of differential equations. These tasks are less abstract than rewriting in the AIM loops theory and involve some more operations than our polynomial dataset. The datasets are orders of magnitude larger than ours.

The training examples were generated by a randomized procedure, similarly to our polynomial dataset. The neural model used there – transformer [164] – is also a non-modified architecture originally designed for the neural machine translation. A step in their pipe-line was preprocessing the symbolic expressions by translating them to more compact prefix notation. This preprocessing step was used by us in the first version of our earlier work on guiding theorem provers

[6]The first version of our work was submitted to AITP'19 in December 2018 [125] and presented several times at workshops and conferences in the first half of 2019. See e.g. invited talks at SAT'19 `http://grid01.ciirc.cvut.cz/~mptp/sat19.pdf` and FORMAL'19 `http://grid01.ciirc.cvut.cz/~mptp/formal19.pdf`.

Table 8.6: Prediction accuracies of the three NMT models – the model used in the previous experiments in this work, the default OpenNMT model, and the transformer model used in [98] – on the three integration data sets. The smallest model performs poorly, but the standard, off-the-shelf OpenNMT model trained with a modest computational budget already gives decent performance.

data set	small NMT	default OpenNMT	transformer
BWD	18.4%	67.7%	99.6%
FWD	14.8%	55.7%	97.2%
IBP	17.7%	64.5%	99.3%

by recurrent neural networks [128]. No experimental justification of usefulness of prefix notation is provided in [98]. In the work presented in Chapter 3 we have found that this is not always beneficial.

The performance of the trained models in [98] was quite high, reaching 99%. Here we analyze closer this experiment and compare with our methods and datasets. First, we wanted to see how the relatively small models used originally by us perform on this data. We took 3 data sets related to various kinds of integration operations (BWD, FWD, IBP) used in [98] and applied the NMT model with exactly the same hyper-parameters as described in 8.3. Additionally, we trained an NMT model implemented in OpenNMT, leaving all the hyper-parameters in their default settings.[7] Table 8.6 shows the accuracy of the trained models on the test sets, along with the reported accuracies of the transformer.

We see that the small NMT model performs much worse than the transformer model from [98], while the performance of the default OpenNMT model is already quite good. However, the transformer model in [98] was much larger. The authors do not report on how long the model was trained and what infrastructure was used. Our small NMT model was trained for about one hour on one GPU and our default OpenNMT model was trained for two hours on two GPUs. With such straightforward, unmodified training procedure and short training times the achieved performance may still be seen as surprisingly high. By tuning the hyper-parameters and increasing the number of training steps we

[7] In particular: the number of training steps was 100 000, the number of layers in the encoder and the decoder was 2, the number of units in the encoder and decoder was 500, the "scaled Luong" version of the attention mechanism was used; the predictions were generated using beam search of width 10 and 1 best output was considered only.

Table 8.7: Number of unique (not overlapping with the training set) testing examples *modulo constant* or *modulo constant and sign*. All the polynomial data sets and the integration data sets from [98] were checked. Some datasets (`FWD`, `IBP`) contain many testing examples that are similar to the training examples, which motivates a question whether such a situation should be considered as a leak in the dataset.

data set	# unique mod. constant	# unique mod. constant and sign	# all test examples
`BWD`	7421 (80%)	6999 (75%)	9319
`FWD`	4404 (44%)	3497 (35%)	9986
`IBP`	2345 (30%)	1895 (24%)	7777
`poly1`	34 877 (58%)	–	60 000
`poly2`	69 160 (77%)	–	90 000
`poly3`	82 680 (92%)	–	90 000
`poly4`	77 225 (86%)	–	90 000
`poly5`	79 185 (88%)	–	90 000
`poly6`	77 764 (86%)	–	90 000

could likely easily increase the performance.

To get more understanding of the data, we have done a simple analysis of the similarity between the training and testing sets. We substituted all the constants (i.e., digits) with `CONST` token and checked how many such modified testing examples appear in the training examples, and how many are unique to the testing set. We did this for all the polynomial data sets and the integration data sets. For the latter, we also substituted plus sign for all the minus signs to ignore the sign of integers when comparing the examples. (In the polynomial data there is no negative integers.) The results of such analysis are shown in Table 8.7.

We see that for some of the integration data sets the number of examples unique modulo constant (or constant and sign) may be as low as 24% (`IBP`). This means that for this data set 75% of the testing examples appear in the training set as very similar expressions, just with changed constants and their signs. This motivates the need for more careful/complex analysis of the performance of machine learning models in new domains like the symbolic rewriting. Reporting only accuracy on the separated testing set may not be enough. It may happen that the performance of the model is dependent on some hidden factors, e.g.,

undesired "leaks" in the data.

The analysis done here is initial and more detailed examination would be useful to measure the level of generalization and memorization done by the neural models used. Our experiments on the polynomial data show that just increasing the number of constants and variables from two to three between poly5 and poly6 decreased the testing performance from 85.5% to 71.8%. Similar ablations should be done with related experiments. Methods such as decreasing the size of the training set, making it on average more diverse, and measuring the average Levenshtein distance between the training and testing examples [167] are relatively straightforward to apply.

8.5 Conclusions and future work

NMT is not typically applied to symbolic problems, but surprisingly, it performed very well for both described tasks. The first one was easier in terms of complexity of the rewriting (only one application of a rewrite rule was performed), but the number of examples was quite limited. The second task involved more difficult rewriting – multiple different rewrite steps were performed to construct the examples. Nevertheless, provided many examples, NMT could learn normalizing polynomials.

We hope this work provides a baseline and inspiration for continuing this line of research. We see several interesting directions this work can be extended.

Firstly, more interesting and difficult rewriting problems need to be provided for better delineation of the strength of the neural models. The described data are relatively simple and with no direct relevance to the real unsolved symbolic problems. But the results on these simple problems are encouraging enough to try with more challenging ones, related to real difficulties – e.g., these from TPDB data base.[8]

Secondly, we are going to develop and test new kinds of neural models tailored for the problem of comprehending symbolic expressions. Specifically, we are going to implement an approach based on the idea of TreeNN, which may be another effective approach for this kind of tasks [31, 46, 107]. TreeNNs are built recursively from *modules*, where the modules correspond to parts of symbolic expression (symbols) and the shape of the network reflects the parse tree of the processed expression. This way model is explicitly informed on the exact structure of the expression, which in case of formal logic is always unambiguous and easy to extract. Perhaps this way the model could learn more efficiently

[8] `http://termination-portal.org/wiki/TPDB`

from examples (and achieve higher results even on the small AIM data sets). The authors have a positive experience of applying TreeNNs to learn remainders of arithmetical expressions modulo small natural numbers – TreeNNs outperformed here neural models based on LSTM cells, giving almost perfect accuracy. However, this is unclear how to translate this TreeNN methodology to the tasks with the structured output, like the symbolic rewriting task.

Thirdly, there is an idea of integrating neural rewriting architectures into the larger systems for automated reasoning. This can be motivated by the interesting contrast between some simpler ILP systems suffering from the combinatorial explosion in the presence of a large number of examples and neural methods which definitely benefit from large data sets.

We hope that this work will inspire and trigger a discussion on the above (and other) ideas.

Bibliography

[1] Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., and Urban, J. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning 52*, 2 (2014), 191–213.

[2] Avigad, J. The mechanization of mathematics. *Notices of the AMS 65*, 6 (2018), 681–90.

[3] Avigad, J. Mathematics and the formal turn. `https://www.andrew.cmu.edu/user/avigad/Papers/formal_turn.pdf`, 2023.

[4] Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *CoRR abs/2302.12433* (2023).

[5] Balunovic, M., Bielik, P., and Vechev, M. T. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (2018), S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 10338–10349.

[6] Bancerek, G., Naumowicz, A., and Urban, J. System description: XSL-based translator of Mizar to LaTeX. In Rabe et al. [133], pp. 1–6.

[7] Bansal, K., Loos, S. M., Rabe, M. N., Szegedy, C., and Wilcox, S. HOList: An environment for machine learning of higher order logic theorem proving. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA* (2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 454–463.

[8] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems – 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS* (2022), D. Fisman and G. Rosu, Eds., Springer.

[9] Barnett, M., Chang, B. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO* (2005), vol. 4111, Springer, pp. 364–387.

[10] Barrett, C. W., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Computer Aided Verification – 23rd International Conference, CAV* (2011), vol. 6806, Springer, pp. 171–177.

[11] Barrett, C. W., de Moura, L. M., Ranise, S., Stump, A., and Tinelli, C. The SMT-LIB initiative and the rise of SMT (HVC 2010 award talk). In *Hardware and Software: Verification and Testing – 6th International Haifa Verification Conference, HVC* (2010), vol. 6504, Springer, p. 3.

[12] Barrett, C. W., and Tinelli, C. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[13] Bauer, A. What makes dependent type theory more suitable than set theory for proof assistants? `https://mathoverflow.net/questions/376839/what-makes-dependent-type-theory-more-suitable-than-set-theory-for-proof-assista/376973#376973`. Accessed: 2023-04-06.

[14] Bawa, M., Condie, T., and Ganesan, P. LSH Forest: Self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005* (2005), A. Ellis and T. Hagino, Eds., ACM, pp. 651–660.

[15] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[16] Bhayat, A., and Reger, G. A combinator-based superposition calculus for higher-order logic. In *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I* (2020), N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166 of *Lecture Notes in Computer Science*, Springer, pp. 278–296.

[17] Bjørner, N., and Janota, M. Playing with quantified satisfaction. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning – Short Presentations, LPAR* (2015), vol. 35, EasyChair, pp. 15–27.

[18] Blaauwbroek, L., Urban, J., and Geuvers, H. Tactic learning and proving for the Coq proof assistant. In *Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2020* (2020), E. Albert and L. Kovács, Eds., vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 138–150.

[19] Blaauwbroek, L., Urban, J., and Geuvers, H. The Tactician – A seamless, interactive tactic learner and prover for Coq. In *Proceedings of the 13th International Conference on Intelligent Computer Mathematics CICM 2020, Bertinoro, Italy, July 26-31, 2020* (2020), C. Benzmüller and B. R. Miller, Eds., vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 271–277.

[20] Blanchette, J. C., Böhme, S., and Paulson, L. C. Extending Sledgehammer with SMT solvers. *J. Autom. Reason. 51*, 1 (2013), 109–128.

[21] Blanchette, J. C., Greenaway, D., Kaliszyk, C., Kühlwein, D., and Urban, J. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning 57*, 3 (2016), 219–244.

[22] Blanchette, J. C., Kaliszyk, C., Paulson, L. C., and Urban, J. Hammering towards QED. *J. Formaliz. Reason. 9*, 1 (2016), 101–148.

[23] Blanchette, J. C., Popescu, A., Wand, D., and Weidenbach, C. More SPASS with Isabelle – superposition with hard sorts and configurable simplification. In *ITP* (2012), L. Beringer and A. P. Felty, Eds., vol. 7406 of *Lecture Notes in Computer Science*, Springer, pp. 345–360.

[24] Bouton, T., Oliveira, D. C. B. D., Déharbe, D., and Fontaine, P. veriT: An open, trustable and efficient SMT-solver. In *Automated*

Deduction – CADE-22, 22nd International Conference on Automated (2009), vol. 5663, Springer, pp. 151–156.

[25] Breiman, L. Random forests. *Mach. Learn. 45*, 1 (2001), 5–32.

[26] Bridge, J. P., Holden, S. B., and Paulson, L. C. Machine learning for first-order theorem proving – learning to select a good heuristic. *J. Autom. Reason. 53*, 2 (2014), 141–172.

[27] Broder, A. Z. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings* (1997), B. Carpentieri, A. D. Santis, U. Vaccaro, and J. A. Storer, Eds., IEEE, pp. 21–29.

[28] Brown, C. E. Satallax: An automatic higher-order prover. In *Automated Reasoning – 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings* (2012), B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364 of *Lecture Notes in Computer Science*, Springer, pp. 111–117.

[29] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds.

[30] Buda, M., Maki, A., and Mazurowski, M. A. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks 106* (2018), 249–259.

[31] Chakraborty, S., Allamanis, M., and Ray, B. Tree2tree neural translation model for learning source code changes. *CoRR abs/1810.00314* (2018).

[32] Charton, F., Hayat, A., and Lample, G. Learning advanced mathematical computations from examples. In *9th International Conference on*

Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021 (2021), OpenReview.net.

[33] Chen, T., and Guestrin, C. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016* (2016), B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds., ACM, pp. 785–794.

[34] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (2014), A. Moschitti, B. Pang, and W. Daelemans, Eds., ACL, pp. 1724–1734.

[35] Chvalovský, K., Jakubuv, J., Suda, M., and Urban, J. ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. In *Proceedings of the 27th International Conference on Automated Deduction, CADE 27, Natal, Brazil, August 27-30, 2019* (2019), P. Fontaine, Ed., vol. 11716 of *Lecture Notes in Computer Science*, Springer, pp. 197–215.

[36] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., and Schulte, W. VCC: Contract-based modular verification of concurrent c. In *2009 31st International Conference on Software Engineering-Companion Volume* (2009), IEEE, pp. 429–430.

[37] de Moura, L., and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28 – 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings* (2021), A. Platzer and G. Sutcliffe, Eds., vol. 12699 of *Lecture Notes in Computer Science*, Springer, pp. 625–635.

[38] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.

[39] Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. Indexing by latent semantic analysis. *JASIS 41*, 6 (1990), 391–407.

[40] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: A theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[41] Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT* (2019).

[42] Domingos, P. M., and Hulten, G. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2000), R. Ramakrishnan, S. J. Stolfo, R. J. Bayardo, and I. Parsa, Eds., ACM, pp. 71–80.

[43] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. *J. Comput. Syst. Sci. 38*, 1 (1989), 86–124.

[44] Dutertre, B. Yices 2.2. In *Computer Aided Verification – 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.

[45] El Ouraou, D., Blanchette, J. C., Fontaine, P., and Kaliszyk, C. Machine learning for instance selection in SMT solving. `https://matryoshka-project.github.io/pubs/instanceselection_paper.pdf`, 2020.

[46] Evans, R., Saxton, D., Amos, D., Kohli, P., and Grefenstette, E. Can neural networks understand logical entailment? In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 – May 3, 2018, Conference Track Proceedings* (2018), OpenReview.net.

[47] Färber, M., and Kaliszyk, C. Random forests for premise selection. In *Frontiers of Combining Systems – 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings* (2015),

C. Lutz and S. Ranise, Eds., vol. 9322 of *Lecture Notes in Computer Science*, Springer, pp. 325–340.

[48] Farzan, A., and Kincaid, Z. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang. 2*, POPL (2018), 61:1–61:30.

[49] Fawcett, T. An introduction to ROC analysis. *Pattern Recognit. Lett. 27*, 8 (2006), 861–874.

[50] Fidora, A., and Sierra, C., Eds. *Ramon Llull: From the Ars Magna to Artificial Intelligence*. Artificial Intelligence Research Institute, 2011.

[51] Fischer, M. J., and Rabin, M. O. Super-exponential complexity of presburger arithmetic. In *Texts and Monographs in Symbolic Computation*. Springer Vienna, 1998, pp. 122–135.

[52] Freitag, M., and Al-Onaizan, Y. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation, NMT@ACL 2017, Vancouver, Canada, August 4, 2017* (2017), T. Luong, A. Birch, G. Neubig, and A. M. Finch, Eds., Association for Computational Linguistics, pp. 56–60.

[53] Gauthier, T. Deep reinforcement learning for synthesizing functions in higher-order logic. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020* (2020), E. Albert and L. Kovács, Eds., vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 230–248.

[54] Gauthier, T., and Kaliszyk, C. Premise selection and external provers for HOL4. In *Proceedings of the 4th Conference on Certified Programs and Proofs (CPP'15)* (2015), X. Leroy and A. Tiu, Eds., ACM, pp. 49–57.

[55] Gauthier, T., Kaliszyk, C., and Urban, J. Initial experiments with statistical conjecturing over large formal corpora. In *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2016 co-located with the 9th Conference on Intelligent Computer Mathematics (CICM 2016), Bialystok, Poland, July 25-29, 2016* (2016), A. Kohlhase, P. Libbrecht, B. R. Miller, A. Naumowicz, W. Neuper, P. Quaresma, F. W. Tompa, and M. Suda, Eds., vol. 1785 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 219–228.

[56] Gauthier, T., Kaliszyk, C., and Urban, J. TacticToe: Learning to reason with HOL4 tactics. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017* (2017), T. Eiter and D. Sands, Eds., vol. 46 of *EPiC Series in Computing*, EasyChair, pp. 125–143.

[57] Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., and Norrish, M. TacticToe: Learning to prove with tactics. *J. Autom. Reason. 65*, 2 (2021), 257–286.

[58] Ge, Y., and de Moura, L. M. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Computer Aided Verification, 21st International Conference, CAV* (2009), pp. 306–320.

[59] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds., Morgan Kaufmann, pp. 518–529.

[60] Grabowski, A., Kornilowicz, A., and Naumowicz, A. Mizar in a nutshell. *J. Formalized Reasoning 3*, 2 (2010), 153–245.

[61] Hales, T. C. A proof of the Kepler conjecture. *Annals of Mathematics 162* (2005), 1063–1183.

[62] Hales, T. C., Adams, M., Bauer, G., Dang, D. T., Harrison, J., Hoang, T. L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., Nguyen, T. Q., Nipkow, T., Obua, S., Pleso, J., Rute, J. M., Solovyev, A., Ta, A. H. T., Tran, T. N., Trieu, D. T., Urban, J., Vu, K. K., and Zumkeller, R. A formal proof of the Kepler conjecture. *CoRR abs/1501.02155* (2015).

[63] Har-Peled, S., Indyk, P., and Motwani, R. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory Comput. 8*, 1 (2012), 321–350.

[64] Harrison, J. HOL Light: A tutorial introduction. In *FMCAD* (1996), M. K. Srivas and A. J. Camilleri, Eds., vol. 1166 of *Lecture Notes in Computer Science*, Springer, pp. 265–269.

[65] Hastie, T., Tibshirani, R., and Friedman, J. H. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition.* Springer Series in Statistics. Springer, 2009.

[66] Herbrand, J. *Recherches sur la théorie de la démonstration.* Doctorat d'état, La Faculté des Sciences de Paris, 1930.

[67] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[68] Hoder, K., and Voronkov, A. Sine qua non for large theory reasoning. In *Automated Deduction – CADE-23 – 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 – August 5, 2011. Proceedings* (2011), N. S. Bjørner and V. Sofronie-Stokkermans, Eds., vol. 6803 of *Lecture Notes in Computer Science*, Springer, pp. 299–314.

[69] Holden, S. B. Machine learning for automated theorem proving: Learning to solve SAT and QSAT. *Found. Trends Mach. Learn. 14*, 6 (2021), 807–989.

[70] Huzaifah, M., and Wyse, L. Deep generative models for musical audio synthesis. *CoRR abs/2006.06426* (2020).

[71] Irving, G., Szegedy, C., Alemi, A. A., Eén, N., Chollet, F., and Urban, J. DeepMath – Deep sequence models for premise selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain* (2016), D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., pp. 2235–2243.

[72] Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., and Urban, J. ENIGMA Anonymous: Symbol-independent inference guiding machine (system description). In *Automated Reasoning – 10th International Joint Conference, IJCAR* (2020), vol. 12167, Springer, pp. 448–463.

[73] Jakubuv, J., and Urban, J. BliStrTune: Hierarchical invention of theorem proving strategies. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017* (2017), Y. Bertot and V. Vafeiadis, Eds., ACM, pp. 43–52.

[74] Jakubuv, J., and Urban, J. ENIGMA: Efficient learning-based inference guiding machine. In *Intelligent Computer Mathematics – 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings* (2017), H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, Eds., vol. 10383 of *Lecture Notes in Computer Science*, Springer, pp. 292–302.

[75] Janota, M., Barbosa, H., Fontaine, P., and Reynolds, A. Fair and adventurous enumeration of quantifier instantiations. In *Formal Methods in Computer-Aided Design* (2021).

[76] Janota, M., Piepenbrock, J., and Piotrowski, B. Towards learning quantifier instantiation in SMT. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel* (2022), K. S. Meel and O. Strichman, Eds., vol. 236 of *LIPIcs*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 7:1–7:18.

[77] Jech, T. J. *The Axiom of Choice*. Courier Corporation, 2008.

[78] Jones, K. S. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation 60*, 5 (2004), 493–502.

[79] Kaliszyk, C., Chollet, F., and Szegedy, C. HolStep: A machine learning dataset for higher-order logic theorem proving. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (2017), OpenReview.net.

[80] Kaliszyk, C., and Urban, J. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning 53*, 2 (2014), 173–213.

[81] Kaliszyk, C., and Urban, J. FEMaLeCoP: Fairly efficient machine learning connection prover. In *LPAR-20* (2015), pp. 88–96.

[82] Kaliszyk, C., and Urban, J. MizAR 40 for Mizar 40. *J. Autom. Reasoning 55*, 3 (2015), 245–256.

[83] Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada* (2018), S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 8836–8847.

[84] Kaliszyk, C., Urban, J., and Vyskočil, J. Efficient semantic features for automated reasoning over large theories. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, (IJCAI'15)* (2015), Q. Yang and M. Wooldridge, Eds., AAAI Press, pp. 3084–3090.

[85] Kaliszyk, C., Urban, J., and Vyskočil, J. Learning to parse on aligned corpora (rough diamond). In *Interactive Theorem Proving – 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings* (2015), C. Urban and X. Zhang, Eds., vol. 9236 of *Lecture Notes in Computer Science*, Springer, pp. 227–233.

[86] Kaliszyk, C., Urban, J., and Vyskočil, J. Automating formalization by statistical and semantic parsing of mathematics. In *Interactive Theorem Proving – 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings* (2017), M. Ayala-Rincón and C. A. Muñoz, Eds., vol. 10499 of *Lecture Notes in Computer Science*, Springer, pp. 12–27.

[87] Kaliszyk, C., Urban, J., Vyskočil, J., and Geuvers, H. Developing corpus-based translation methods between informal and formal mathematics: Project description. In *Intelligent Computer Mathematics – International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings* (2014), S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, Eds., vol. 8543 of *Lecture Notes in Computer Science*, Springer, pp. 435–439.

[88] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems* (2017), pp. 3146–3154.

[89] Kinyon, M. K., Veroff, R., and Vojtechovský, P. Loops with abelian inner mapping groups: An application of automated deduction. In *Automated Reasoning and Mathematics – Essays in Memory of William W. McCune* (2013), M. P. Bonacina and M. E. Stickel, Eds., vol. 7788 of *Lecture Notes in Computer Science*, Springer, pp. 151–164.

[90] Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations* (Vancouver, Canada, July 2017), Association for Computational Linguistics, pp. 67–72.

[91] Korovin, K. iProver – An instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings* (2008), A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195 of *Lecture Notes in Computer Science*, Springer, pp. 292–298.

[92] Kovács, L., and Voronkov, A. First-order theorem proving and Vampire. In *CAV* (2013), N. Sharygina and H. Veith, Eds., vol. 8044 of *Lecture Notes in Computer Science*, Springer, pp. 1–35.

[93] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems* (2012), pp. 1106–1114.

[94] Kucik, A. S., and Korovin, K. Premise selection with neural networks and distributed representation of features. *CoRR abs/1807.10268* (2018).

[95] Kuehlwein, D., and Urban, J. Learning from multiple proofs: First experiments. In *PAAR-2012* (2013), P. Fontaine, R. A. Schmidt, and S. Schulz, Eds., vol. 21 of *EPiC Series*, EasyChair, pp. 82–94.

[96] Kühlwein, D., and Urban, J. MaLeS: A framework for automatic tuning of automated theorem provers. *J. Autom. Reason. 55*, 2 (2015), 91–116.

[97] Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., and Heskes, T. Overview and evaluation of premise selection techniques for large theory mathematics. In *Automated Reasoning – 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings* (2012), B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364 of *Lecture Notes in Computer Science*, Springer, pp. 378–392.

[98] Lample, G., and Charton, F. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (2020), OpenReview.net.

[99] Lenat, D. B. *AM, an artificial intelligence approach to discovery in mathematics as heuristic search.* PhD thesis, Stanford University, USA, 1976.

[100] Letz, R., Mayr, K., and Goller, C. Controlled integration of the cut rule into connection tableau calculi. *J. Autom. Reasoning 13* (1994), 297–337.

[101] Loos, S. M., Irving, G., Szegedy, C., and Kaliszyk, C. Deep network guided proof search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017* (2017), T. Eiter and D. Sands, Eds., vol. 46 of *EPiC Series in Computing*, EasyChair, pp. 85–105.

[102] Luong, M., Brevdo, E., and Zhao, R. Neural machine translation (seq2seq) tutorial. `https://github.com/tensorflow/nmt`. Accessed: 2017.

[103] Luong, T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015* (2015), L. Màrquez, C. Callison-Burch, J. Su, D. Pighin, and Y. Marton, Eds., The Association for Computational Linguistics, pp. 1412–1421.

[104] mathlib Community, T. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2020), CPP 2020, Association for Computing Machinery, p. 367–381.

[105] Megill, N. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina.

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

[107] Miao, N., Wang, H., Le, R., Tao, C., Shang, M., Yan, R., and Zhao, D. Tree2tree learning with memory unit, 2018.

[108] Mikula, M., Antoniak, S., Tworkowski, S., Jiang, A. Q., Zhou, J. P., Szegedy, C., Kucinski, L., Milos, P., and Wu, Y. Magnushammer: A transformer-based approach to premise selection. *CoRR abs/2303.04488* (2023).

[109] Mitchell, T. M. *Machine Learning*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[110] Monk, J. D. *Mathematical Logic*. Springer Science & Business Media, 2012.

[111] Moy, Y., and Wallenburg, A. Tokeneer: Beyond formal program verification. *Embedded Real Time Software and Systems 24* (2010).

[112] Muggleton, S., and De Raedt, L. Inductive logic programming: Theory and methods. *The Journal of Logic Programming 19* (1994), 629–679.

[113] Nagashima, Y., and He, Y. PaMpeR: Proof method recommendation system for Isabelle/HOL. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018* (2018), M. Huchard, C. Kästner, and G. Fraser, Eds., ACM, pp. 362–372.

[114] Nagashima, Y., and Kumar, R. A proof strategy language and proof script generation for Isabelle/HOL. In *Proceedings of the 26th International Conference on Automated Deduction, CADE 26* (2017), L. de Moura, Ed., vol. 10395 of *Lecture Notes in Computer Science*, Springer, pp. 528–545.

[115] Nejati, S., Frioux, L. L., and Ganesh, V. A machine learning based splitting heuristic for divide-and-conquer solvers. In *Principles and Practice of Constraint Programming – 26th International Conference, CP* (2020), vol. 12333, Springer, pp. 899–916.

[116] Niemetz, A., Preiner, M., Reynolds, A., Barrett, C. W., and Tinelli, C. Solving quantified bit-vectors using invertibility conditions. In *Computer Aided Verification – 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018* (2018), vol. 10982, Springer, pp. 236–255.

[117] Niemetz, A., Preiner, M., Reynolds, A., Barrett, C. W., and Tinelli, C. Syntax-guided quantifier instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS* (2021), vol. 12652, Springer, pp. 145–163.

[118] Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., and Tinelli, C. Towards bit-width-independent proofs in SMT

solvers. In *International Conference on Automated Deduction* (2019), Springer, pp. 366–384.

[119] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM 53*, 6 (2006), 937–977.

[120] Olaleye, E. How to win any ML contest, 2021. Published as `https://medium.com/machine-learning-insights/how-to-win-any-ml-contest-244a12c62f30`.

[121] Olsák, M., Kaliszyk, C., and Urban, J. Property invariant embedding for automated reasoning. In *ECAI 2020 – 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)* (2020), G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 1395–1402.

[122] Otten, J., and Bibel, W. leanCoP: Lean connection-based theorem proving. *J. Symb. Comput. 36*, 1-2 (2003), 139–161.

[123] Paulson, L. C. The foundation of a generic theorem prover. *J. Autom. Reason. 5*, 3 (1989), 363–397.

[124] Pimpalkhare, N., Mora, F., Polgreen, E., and Seshia, S. A. Medleysolver: Online SMT algorithm selection. In *Theory and Applications of Satisfiability Testing – SAT 2021 – 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings* (2021), C. Li and F. Manyà, Eds., vol. 12831 of *Lecture Notes in Computer Science*, Springer, pp. 453–470.

[125] Piotrowski, B., Brown, C., Urban, J., and Kaliszyk, C. Can neural networks learn symbolic rewriting? In *Proceedings of AITP 2019* (2019).

[126] Piotrowski, B., Mir, R. F., and Ayers, E. W. Machine-learned premise selection for Lean. *CoRR abs/2304.00994* (2023).

[127] Piotrowski, B., and Urban, J. ATPboost: Learning premise selection in binary setting with ATP feedback. In *Automated Reasoning – 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated*

Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (2018), D. Galmiche, S. Schulz, and R. Sebastiani, Eds., vol. 10900 of *Lecture Notes in Computer Science*, Springer, pp. 566–574.

[128] Piotrowski, B., and Urban, J. Guiding theorem proving by recurrent neural networks. *CoRR abs/1905.07961* (2019).

[129] Piotrowski, B., and Urban, J. Guiding inferences in connection tableau by recurrent neural networks. In *Intelligent Computer Mathematics – 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings* (2020), C. Benzmüller and B. R. Miller, Eds., vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 309–314.

[130] Piotrowski, B., and Urban, J. Stateful premise selection by recurrent neural networks. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020* (2020), vol. 73, EasyChair, pp. 409–422.

[131] Piotrowski, B., Urban, J., Brown, C. E., and Kaliszyk, C. Can neural networks learn symbolic rewriting? *CoRR* (2019).

[132] Piskac, R., Wies, T., and Zufferey, D. GRASShopper. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2014), Springer.

[133] Rabe, F., Farmer, W. M., Passmore, G. O., and Youssef, A., Eds. *Intelligent Computer Mathematics – 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings* (2018), vol. 11006 of *Lecture Notes in Computer Science*, Springer.

[134] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners.

[135] Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. Hierarchical text-conditional image generation with CLIP latents. *CoRR abs/2204.06125* (2022).

[136] Reynolds, A., Barbosa, H., and Fontaine, P. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems* (2018), vol. 10806, pp. 112–131.

[137] Reynolds, A., King, T., and Kuncak, V. Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods Syst. Des. 51*, 3 (2017), 500–532.

[138] Reynolds, A., Tinelli, C., and de Moura, L. M. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014* (2014), IEEE, pp. 195–202.

[139] Risch, R. H. The problem of integration in finite terms. *Transactions of the American Mathematical Society 139* (1969), 167–189.

[140] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022* (2022), IEEE, pp. 10674–10685.

[141] Saffari, A., Leistner, C., Santner, J., Godec, M., and Bischof, H. On-line random forests. In *12th IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2009, Kyoto, Japan, September 27 – October 4, 2009* (2009), IEEE Computer Society, pp. 1393–1400.

[142] Scholze, P. Liquid tensor experiment. *Exp. Math. 31*, 2 (2022), 349–354.

[143] Scholze, P., and Buzzard, K. Half a year of the liquid tensor experiment: Amazing developments. `https://xenaproject.wordpress.com/2021/06/05/half-a-year-of-the-liquid-tensor-experiment-amazing-developments/`. Accessed: 2023-04-06.

[144] Schulz, S., Cruanes, S., and Vukmirovic, P. Faster, higher, stronger: E 2.3. In *Automated Deduction – CADE 27 – 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings* (2019), P. Fontaine, Ed., vol. 11716 of *Lecture Notes in Computer Science*, Springer, pp. 495–507.

[145] Scott, J., Niemetz, A., Preiner, M., Nejati, S., and Ganesh, V. MachSMT: A machine learning-based algorithm selector for SMT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS* (2021), vol. 12652, Springer, pp. 303–325.

[146] Selsam, D., and Bjørner, N. Guiding high-performance SAT solvers with unsat-core predictions. In *Theory and Applications of Satisfiability*

Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal (2019), M. Janota and I. Lynce, Eds., vol. 11628, Springer, pp. 336–353.

[147] Slind, K., and Norrish, M. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings* (2008), O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds., vol. 5170 of *Lecture Notes in Computer Science*, Springer, pp. 28–32.

[148] Steen, A., and Benzmüller, C. The higher-order prover Leo-III. In *Automated Reasoning – 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings* (2018), D. Galmiche, S. Schulz, and R. Sebastiani, Eds., vol. 10900 of *Lecture Notes in Computer Science*, Springer, pp. 108–116.

[149] Suda, M. Vampire getting noisy: Will random bits help conquer chaos? (system description). In *Automated Reasoning – 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings* (2022), J. Blanchette, L. Kovács, and D. Pattinson, Eds., vol. 13385 of *Lecture Notes in Computer Science*, Springer, pp. 659–667.

[150] Sutcliffe, G. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning 43*, 4 (2009), 337–362.

[151] Sutcliffe, G. The TPTP world – infrastructure for automated reasoning. In *LPAR (Dakar)* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, pp. 1–12.

[152] Sutcliffe, G. The TPTP problem library and associated infrastructure – from CNF to TH0, TPTP v6.4.0. *J. Autom. Reason. 59*, 4 (2017), 483–502.

[153] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (2014), Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 3104–3112.

[154] Szegedy, C. A promising path towards autoformalization and general artificial intelligence. In *Intelligent Computer Mathematics – 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings* (2020), C. Benzmüller and B. R. Miller, Eds., vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 3–20.

[155] The Coq Development Team. The Coq proof assistant, version 8.11.0, Oct 2019.

[156] Tsivtsivadze, E., Urban, J., Geuvers, H., and Heskes, T. Semantic graph kernels for automated reasoning. In *SDM* (2011), SIAM / Omnipress, pp. 795–803.

[157] Tworkowski, S., Mikula, M., Odrzygóźdź, T., Czechowski, K., Antoniak, S., Jiang, A. Q., Szegedy, C., Łukasz Kuciński, Miłoś, P., and Wu, Y. Formal premise selection with language models. In *The 7th Conference on Artificial Intelligence and Theorem Proving, AITP 2022, September 4-9, 2022, Aussois and online, France* (2022).

[158] Urban, J. Experimenting with machine learning in automatic theorem proving. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 1998. English summary at `https://www.ciirc.cvut.cz/~urbanjo3/MScThesisPaper.pdf`.

[159] Urban, J. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning 37*, 1-2 (2006), 21–43.

[160] Urban, J., and Jakubuv, J. First neural conjecturing datasets and experiments. In *Intelligent Computer Mathematics – 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings* (2020), C. Benzmüller and B. R. Miller, Eds., vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 315–323.

[161] Urban, J., Sutcliffe, G., Pudlák, P., and Vyskočil, J. MaLARea SG1 – machine learner for automated reasoning with semantic guidance. In *Automated Reasoning – 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings* (2008), A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195 of *Lecture Notes in Computer Science*, Springer, pp. 441–456.

[162] Urban, J., Vyskočil, J., and Štěpánek, P. MaLeCoP: Machine learning connection prover. In *TABLEAUX* (2011), K. Brünnler and

G. Metcalfe, Eds., vol. 6793 of *Lecture Notes in Computer Science*, Springer, pp. 263–277.

[163] Vanegue, J., and Lahiri, S. Towards practical reactive security audit using extended static checkers. In *IEEE Symposium on Security and Privacy (Oakland'13)* (May 2013).

[164] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 6000–6010.

[165] Vukmirovic, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., and Tourret, S. Making higher-order superposition work. *J. Autom. Reason. 66*, 4 (2022), 541–564.

[166] Vukmirovic, P., Blanchette, J., and Schulz, S. Extending a high-performance prover to higher-order logic. In *Tools and Algorithms for the Construction and Analysis of Systems – 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II* (2023), S. Sankaranarayanan and N. Sharygina, Eds., vol. 13994 of *Lecture Notes in Computer Science*, Springer, pp. 111–129.

[167] Wang, Q., Kaliszyk, C., and Urban, J. First experiments with neural translation of informal to formal mathematics. In Rabe et al. [133], pp. 255–270.

[168] Wiedijk, F., Ed. *The Seventeen Provers of the World*, vol. 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[169] Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. In *NeurIPS* (2022).

[170] Yildirim, S. How to tune the hyperparameters for better performance, 2020. Published as `https://towardsdatascience.com/how-to-tune-the-hyperparameters-for-better-performance-cfe223d398b3`.

[171] Zhang, C., Zhang, Y., Shi, X., Almpanidis, G., Fan, G., and Shen, X. On incremental learning for gradient boosting decision trees. *Neural Process. Lett. 50*, 1 (2019), 957–987.

[172] Zhang, L., Blaauwbroek, L., Piotrowski, B., Cerný, P., Kaliszyk, C., and Urban, J. Online machine learning techniques for Coq: A comparison. In *Intelligent Computer Mathematics – 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings* (2021), F. Kamareddine and C. S. Coen, Eds., vol. 12833 of *Lecture Notes in Computer Science*, Springer, pp. 67–83.

[173] Zinn, C. *Understanding informal mathematical discourse.* PhD thesis, University of Erlangen-Nuremberg, 2004.

[174] Zombori, Z., Urban, J., and Brown, C. E. Prolog technology reinforcement learning prover (system description). In *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II* (2020), N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12167 of *Lecture Notes in Computer Science*, Springer, pp. 489–507.

Research data management

The following research datasets have been produced during this PhD research:

- **Chapter 2:** code and data are stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/ATPboost`
- **Chapter 3:** code and data are stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/stateful-premise-selection-with-RNNs`
- **Chapter 4:** code and data are stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/guiding-connection-tableau-by-RNNs`
- **Chapter 5:** code and data are stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/ml-guidance-for-instantiation-in-CVC5`
- **Chapter 6:** code is stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/random-forest`
- **Chapter 7:** code is stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/lean-premise-selection`
- **Chapter 8:** data is stored in a GitHub repository available at `https://github.com/BartoszPiotrowski/rewriting-with-NNs`

Summary

This thesis develops various machine-learning-based methods that improve the success rate of automated theorem provers and facilitate constructing formal proofs in proof assistants. These methods are based on classical machine learning (ML), as well as on modern neural network approaches, emerging as a promising research direction that may further advance automated reasoning.

The first part of the thesis focuses on the premise selection task in automated theorem proving. This is a critical task when an automated theorem prover (ATP) is used over a large theory where typically only a small fraction of the available facts are relevant for proving a new conjecture. Giving too many redundant premises to the ATP significantly decreases the chances of proving the conjecture.

The ATPboost system addressing this problem is introduced. It solves sets of large-theory problems by interleaving ATP runs with machine learning of premise selection from the proofs. Unlike many approaches that use a multi-label setting, the learning is implemented as a binary classification that estimates the pairwise relevance of (*theorem, premise*) pairs. ATPboost uses for this the gradient boosting decision tree algorithm. Learning in the binary setting, however, requires negative examples, and accumulating them is nontrivial due to many alternative proofs. We implement several solutions in the context of the ATP/ML feedback loop and show significant improvement over the multi-label approach.

Later, a new method for premise selection is developed based on recurrent neural networks (RNNs). Unlike the previous method which chooses sets of facts independently of each other by their rank, the new method uses the notion of *state* that is updated each time a choice of a fact is made. The new method is combined with data augmentation techniques. The evaluation shows improvements in terms of the number of new problems solved in comparison to the previous approach.

The second part of the thesis focuses on internal guidance for ATPs. Certain parts of their algorithms require non-deterministic choices to be made. These choices are normally either randomized or governed by pre-designed heuristics. The goal is to provide there machine-learned advice instead, and by this improve the performance.

In this spirit, experiments with applying RNNs for guiding clause selection in the connection tableau proof calculus are performed. The RNN encodes a sequence of literals from the current branch of the partial proof tree to a hidden vector state; using it, the system selects a clause for extending the proof tree. Additionally, a conjecturing experiment is performed where the RNN does not select an existing clause but completely generates the next tableau goal.

Later, we develop an approach of applying ML to solve quantified satisfiability modulo theories (SMT) problems more efficiently. We focus on the enumerative instantiation method of solving quantified formulas. The task is to select the right ground terms to be instantiated. In ML parlance, this means learning to rank ground terms. We devise a series of features of the considered terms and train on them using gradient boosted decision trees. The experiments demonstrate that the ML-guided solver enables us to solve more problems than the base solver and reduce the number of quantifier instantiations.

The third part of the thesis develops ML-based automation for proof assistants. Formalizing mathematics using proof assistants is a laborious task requiring expert knowledge. The formal proofs need to deal with low-level reasoning steps. Also, a mastery of the existing formal library is required in order to reuse formalized theorems. To make proof assistants more user-friendly various forms of automation need to be developed. Here, ML-based approaches learning from already completed proofs are developed.

First, we focus on the Coq proof assistant. Its proofs consist of sequences of *tactics* that modify *proof states*. The goal is to learn to suggest the next tactic in a given proof state. We build on top of Tactician, a plugin for Coq that provides a framework for learning from proofs written by the user to synthesize new proofs. Learning happens in an online manner, meaning that the ML model is updated every time the user performs a step in an interactive proof. This provides the user with a seamless, interactive experience, and it takes advantage of the locality of proof similarity: proofs similar to the current proof are likely to be found close by. Two online methods are implemented: k-nearest neighbors based on locality sensitive hashing and custom online random forest. We compare the relative performance of these methods on Coq's standard library.

Later, we introduce an ML-based tool for the Lean proof assistant that suggests *relevant premises* for a theorem being proved by a user. The tool is

based on a modification of the custom random forest model used in the Coq project. It is implemented directly in Lean, which was possible thanks to the rich and efficient metaprogramming features of Lean 4. The random forest is trained on data extracted from mathlib – Lean's formal library. The advice from the trained model is accessible to the user via a command that can be called while constructing a proof interactively.

The last part of the thesis investigates the capabilities of neural language models in the context of mathematics. More specifically, we investigate if the current neural architectures are adequate for learning symbolic rewriting. Two kinds of data sets are proposed for this investigation – one derived from automated proofs and the other being a synthetic set of polynomial terms. The experiments with neural machine translation models are performed and their (surprisingly) good results are discussed. These were one of the very first experiments on applying neural language models to symbolic tasks.

Samenvatting

Dit proefschrift ontwikkelt verschillende op machinaal leren gebaseerde methoden die de succespercentages van geautomatiseerde stellingbewijzers verbeteren en het construeren van formele bewijzen in bewijsassistenten vergemakkelijken. Deze methoden zijn gebaseerd op klassiek machinaal leren (ML), maar ook op moderne neurale netwerkbenaderingen, een opkomende, veelbelovende onderzoeksrichting die geautomatiseerd redeneren verder kan verbeteren.

Het eerste deel van het proefschrift richt zich op de premisselectietaak in geautomatiseerde stellingbewijzers. Dit is een belangrijke taak wanneer een geautomatiseerde stellingbewijzer (ATP, van het Engelse *automated theorem prover*) wordt gebruikt over een grote theorie, waar typisch slechts een kleine fractie van de beschikbare feiten relevant is voor het bewijzen van een nieuwe stelling. Te veel overbodige premissen geven aan de ATP, vermindert aanzienlijk de kans op het bewijzen van de beoogde stelling.

Het systeem ATPboost wordt geïntroduceerd om dit probleem aan te pakken. Het lost collecties van grote theorie-problemen op door stellingbewijzers te combineren met machinaal leren van premisselectie uit eerdere bewijzen. In tegenstelling tot vele benaderingen die een multi-label opzet gebruiken, wordt het leren geïmplementeerd als een binaire classificatie die de paarsgewijze relevantie van (*stelling, premisse*) voorspelt. ATPboost gebruikt hiervoor het gradient boosting beslissingsboomalgoritme. Leren in de binaire opzet vereist echter negatieve voorbeelden, wat niet triviaal is door de vele alternatieve bewijzen. Wij implementeren verschillende oplossingen in de context van de ATP/ML feedback loop en tonen aanzienlijke verbetering ten opzichte van de multi-label aanpak.

Later wordt een nieuwe methode voor premisselectie ontwikkeld op basis van *recurrent neural networks* (RNNs). In tegenstelling tot de vorige methode, die reeksen feiten kiest onafhankelijk van elkaar, gebruikt de nieuwe methode een bewaarde staat, die wordt bijgewerkt telkens wanneer een keuze voor een feit

wordt gemaakt. De nieuwe methode wordt gecombineerd met strategieën voor data-augmentatie. De evaluatie toont verbeteringen in termen van het aantal opgeloste nieuwe problemen in vergelijking met de vorige aanpak.

Het tweede deel van het proefschrift richt zich op de interne begeleiding van ATPs. Voor bepaalde delen van hun algoritmen moeten niet-deterministische keuzes worden gemaakt. Deze keuzes zijn normaal gesproken ofwel willekeurig ofwel gestuurd door vooraf ontworpen heuristiek. Het doel is om daar machinaal aangeleerd advies te geven en daardoor de prestaties te verbeteren.

Met dit doel worden experimenten gedaan met het toepassen van RNNs voor het begeleiden van clausule-selectie in de *connection tableau* bewijscalculus. Het RNN encodeert een reeks literalen van de huidige tak van de incomplete bewijsboom naar een vector; met behulp daarvan selecteert het systeem een clausule om de bewijsboom uit te breiden. Daarnaast wordt een experiment uitgevoerd waarbij het RNN niet een bestaande clausule selecteert, maar het volgende tableau doel volledig genereert.

Later ontwikkelen we een aanpak waarbij ML wordt toegepast om gekwantificeerde satisfiability modulo theorieën (SMT) efficiënter op te lossen. We richten ons op de enumeratieve instantiatie-methode voor het oplossen van gekwantificeerde formules. De taak is het selecteren van de juiste termen te selecteren om mee te instantiëren. In ML-taal betekent dit het leren rangschikken van termen. Wij ontwerpen een reeks kenmerken van de beschikbare termen en trainen daarop met gradient boosted beslissingsbomen. De experimenten tonen aan dat het ML-geleide systeem ons in staat stelt meer problemen op te lossen dan de basisoplosser en het aantal benodigde instantiaties kan verminderen.

Het derde deel van het proefschrift ontwikkelt ML-gebaseerde automatisering voor bewijsassistenten. Het formaliseren van wiskunde met behulp van bewijsassistenten is een bewerkelijke taak, waarvoor expertise nodig is. De formele bewijzen moeten omgaan met zeer precieze redeneerstappen. Ook is beheersing van de bestaande formele bibliotheek vereist om geformaliseerde stellingen te kunnen hergebruiken. Om bewijsassistenten gebruiksvriendelijker te maken, moeten verschillende vormen van automatisering worden ontwikkeld. Hier worden ML-gebaseerde benaderingen ontwikkeld die leren van reeds voltooide bewijzen.

Eerst richten we ons op de Coq-bewijsassistent. In deze software bestaan bewijzen uit reeksen van tactieken die de bewijstoestand wijzigen. Het doel is om te leren de volgende tactiek voor te stellen in een gegeven bewijstoestand. We bouwen voort op Tactician, een plugin voor Coq die een raamwerk biedt voor het leren op basis van bewijzen geschreven door de gebruiker om nieuwe

bewijzen te synthetiseren. Het leren gebeurt online, wat betekent dat het ML-model wordt bijgewerkt telkens wanneer de gebruiker een stap uitvoert in een interactief bewijs. Dit biedt de gebruiker een naadloze, interactieve ervaring, en het maakt gebruik van de localiteit van bewijsovereenkomsten: bewijzen die vergelijkbaar zijn met het huidige bewijs zijn waarschijnlijk in de buurt te vinden. Twee online methodes zijn geïmplementeerd: k-nearest neighbors gebaseerd op localiteitsgevoelige hashing en aangepaste online random forest. We vergelijken de relatieve prestaties van deze methoden op de standaardbibliotheek van Coq.

Vervolgens introduceren we een op ML gebaseerde tool voor de Lean bewijsassistent die suggesties doet voor relevante premissen voor een stelling die door een gebruiker wordt bewezen. Het systeem is gebaseerd op een aanpassing van het random forest model gebruikt in het Coq project. Het is rechtstreeks geïmplementeerd in Lean, wat mogelijk was dankzij de rijke en efficiënte metaprogrammeerfuncties van Lean 4. Het random forest-algoritme wordt getraind op gegevens uit mathlib – de formele bibliotheek van Lean. Het advies van het getrainde model is toegankelijk voor de gebruiker via een commando dat kan worden opgeroepen tijdens de interactieve constructie van een bewijs.

Het laatste deel van het proefschrift onderzoekt de mogelijkheden van neurale taalmodellen in de context van wiskunde. Meer specifiek onderzoeken we of de huidige neurale architecturen geschikt zijn voor het leren van symbolisch herschrijven. Voor dit onderzoek worden twee soorten datasets voorgesteld – één afgeleid van geautomatiseerde bewijzen en de andere is een synthetische set van polynomiale termen. De experimenten met neurale automatische vertaalmodellen modellen worden uitgevoerd en hun (verrassend) goede resultaten worden besproken. Deze waren enkele van de allereerste experimenten met de toepassing van neurale taalmodellen op symbolische taken.

Contributions

Chapter 1 is an introduction to the thesis that has not appeared elsewhere and was written entirely by me. The historical part was motivated and informed mainly by the first three chapters of the monograph "Ramon Llull: From the Ars Magna to Artificial Intelligence" edited by Alexander Fidora and Carles Sierra [50]. The text was proofread by Josef Urban, Mikoláš Janota, and Herman Geuvers, and corrections were applied based on their suggestions.

Chapter 2 is based on a conference paper "ATPboost: Learning Premise Selection in Binary Setting with ATP Feedback" authored by me together with Josef Urban, which was published in the Proceedings of 9th International Joint Conference on Automated Reasoning (IJCAR 2018). Josef Urban was advising with the work and provided the data set used for the experiments. The experiments and initial writing were done by me, and Josef Urban improved and proofread the text.

Chapter 3 is based on a conference paper "Stateful Premise Selection by Recurrent Neural Networks" authored by me together with Josef Urban, which was published in the Proceedings of 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2020). Josef Urban was advising with the work and provided the data set used for the experiments. The experiments and initial writing were done by me, and Josef Urban improved and proofread the text.

Chapter 4 is based on a conference paper "Guiding Inferences in Connection Tableau by Recurrent Neural Networks" authored by me together with Josef Urban, and was published in the Proceedings of 13th International Conference Intelligent Computer Mathematics (CICM 2020). Josef Urban advised with the work and provided the data used for the experiments. The experiments and initial writing were done by me. Josef Urban improved and proofread the text.

Chapter 5 is based on a conference paper "Towards Learning Quantifier Instantiation in SMT" authored by me together with Mikoláš Janota and Jelle

Piepenbrock, which was published in the Proceedings of 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). I was involved in designing features used by machine learning, and I was responsible for designing, implementing and running looping-style experiments. Mikoláš Janota was responsible for implementing featurizer within cvc5, and implementing an interface between cvc5 and LightGBM. I wrote Section 5.4 and helped to write other sections as well.

Chapter 6 is based on a conference paper "Online Machine Learning Techniques for Coq: A Comparison" authored by me together with Liao Zhang, Lasse Blaauwbroek, Prokop Cerný, Cezary Kaliszyk, and Josef Urban, which was published in the Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (CICM 2021). The Tactician framework for machine learning experiments with Coq was implemented by Lasse Blaauwbroek. I was responsible for implementing and evaluating the online random forest algorithm, and I advised Liao Zhang on how to perform experiments with gradient-boosted trees. I wrote Subsection 6.3.2 describing the random forest algorithm, whereas the other sections were written mostly by Lasse Blaauwbroek and Liao Zhang. Josef Urban and Cezary Kaliszyk advised with the work and proofread the paper.

Chapter 7 is based on a conference paper "Machine-Learned Premise Selection for Lean" authored by me together with Ramon Fernández Mir and Edward Ayers, which was published in the Proceedings of 32nd International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2023). I led the project. I implemented and evaluated custom versions of the random forest and k-nearest neighbours algorithms. Ramon Fernández Mir implemented the data extraction tool. Edward Ayers implemented the interface allowing to use the tool in Visual Studio Code interactively. The paper, except Section 7.2, was written by me, and it was complemented and improved together with the other authors.

Chapter 8 is based on a workshop paper "Can Neural Networks Learn Symbolic Rewriting?" authored by me together with Josef Urban, Chad Brown and Cezary Kaliszyk, presented at the Learning and Reasoning with Graph-Structured Representations workshop at the 36th International Conference on Machine Learning (ICML 2019). I was responsible for performing all the experiments and creating the polynomial data set. Chad Brown produced the AIM data set. I wrote the paper. Josef Urban and Cezary Kaliszyk were advising with the work.

Curriculum vitae

Education

- **PhD in Computer Science** 2017 – 2023
- **Research visit**, Carnegie Mellon University 03.2022–09.2022
- **MSc in Philosophy**, University of Warsaw 2014 – 2021
- **Student exchange**, University of Amsterdam 09.2016 – 02.2017
- **MSc in Mathematics**, University of Warsaw 2014 – 2016
- **BSc in Mathematics**, University of Warsaw 2011 – 2014

Employment

- **Junior research scientist**, Czech Institute of Informatics, Robotics and Cybernetics in Prague 2017 – 2022
- **Internship**, Interdisciplinary Centre for Mathematical and Computational Modelling in Warsaw 2015 – 2016

Publications

1. Mikoláš Janota, Bartosz Piotrowski, Karel Chvalovský: Towards Learning Infinite SMT Models. SYNASC 2023
2. Bartosz Piotrowski, Ramon Fernández Mir, Edward Ayers: Machine-Learned Premise Selection for Lean. TABLEAUX 2023

3. Zhangir Azerbayev, Bartosz Piotrowski, Jeremy Avigad: ProofNet: A Benchmark for Autoformalizing and Formally Proving Undergraduate-Level Mathematics Problems. MATH-AI@NeurIPS 2022
4. Mikoláš Janota, Jelle Piepenbroeck, Bartosz Piotrowski: Towards Learning Quantifier Instantiation in SMT. SAT 2022
5. Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Cerný, Cezary Kaliszyk, Josef Urban: Online Machine Learning Techniques for Coq: A Comparison. CICM 2021
6. Jan Jakubův, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, Josef Urban: ENIGMA Anonymous: Symbol-Independent Inference Guiding Machine (System Description). IJCAR 2020
7. Bartosz Piotrowski, Josef Urban: Stateful Premise Selection by Recurrent Neural Networks. LPAR 2020
8. Bartosz Piotrowski, Josef Urban: Guiding Inferences in Connection Tableau by Recurrent Neural Networks. CICM 2020
9. Bartosz Piotrowski, Josef Urban, Chad Brown, Cezary Kaliszyk: Can Neural Networks Learn Symbolic Rewriting? GNN@ICML 2019
10. Bartosz Piotrowski, Josef Urban: ATPboost: Learning Premise Selection in Binary Setting with ATP Feedback. IJCAR 2018
11. Bartosz Piotrowski, Miron Kursa: Fully Automatic Classification of Flow Cytometry Data. ISMIS 2018

Grants & awards

- **Research grant** from the National Science Centre in Poland (28 000 EUR, 2019–2022)
- **Research visit** in Carnegie Mellon University (6 months, 2022)
- **Research visit** in University of Oxford (1 week, 2019)
- **Research visit** in University of Manchester (1 week, 2019)
- **Research visit** in University of Innsbruck (3 weeks, 2018)
- **Finalist** of the 62nd Polish National Mathematical Olympiad (2011)

Acknowledgments

My PhD journey was enriching and adventurous. I am grateful to all these people who made the experience so positive.

I would like to thank my supervisor, Herman Geuvers, for accepting me as an external PhD candidate at Radboud University, and for his professional guidance and invaluable help in the whole process.

I am deeply grateful to Josef Urban, my daily supervisor and mentor, who introduced me to the fascinating world of automated reasoning. He is a hacker and a visionary in one person, and his optimistic research style was inspiring to me. Additionally, he created at CIIRC a stimulating and friendly research environment, and I was fortunate to be a part of it.

Mikoláš Janota overtook the role of my daily supervisor when I joined his project later in my PhD. I want to warmly thank him for guiding me through the arcana of SMT solving, and for our scientifically illuminating discussions.

I am grateful to Henryk Michalewski who initiated my PhD adventure, and with whom I traveled to Prague by night trains to learn about proof assistants.

During my PhD, I had an opportunity to go for a few short – but inspiring – scientific visits. I wanted to thank Cezary Kaliszyk, Konstantin Korovin, and Michael Benedikt, all of whom were great hosts.

I am greatly indebted to Jeremy Avigad who was so supportive when I visited his group at CMU for six months, and who taught me about Lean.

I want to thank all the anonymous reviewers of the publications underlying this thesis for their valuable feedback. I am also grateful to Tom Heskes, Mateja Jamnik, Jasmin Blanchette, Stephan Schulz, and Konstantin Korovin, for assessing this thesis and for providing me with suggestions not only useful for improving the manuscript but which I can also reuse in my future research.

During my research endeavors I interacted with many colleagues, which was greatly enriching – both scientifically and personally; some of the people I want to mention here are: Chad Brown, Karel Chvalovský, Martin Suda (here spe-

cial thanks for our numerous climbing sessions), Jan Hůla, Lasse Blaauwbroek, Zar Goertzel, Jelle Piepenbrock, Thibault Gauthier, Filip Bártek, Jan Jakubův, Mirek Olšák, Adam Pease, Yutaka Nagashima, Liao Zhang, Fernando Larrain Langlois, Ramon Fernández Mir, Ed Ayers, Wojciech Nawrocki, Zhangir Azerbayev, Mario Carneiro, Michael Färber.

Finally, I want to thank my family for their constant support and love. Special thanks to my wonderful wife Olga and my little son Tadeusz (whose arrival slowed me down considerably with wrapping up this thesis, but gave me joy and fresh motivation instead).