Towards automating formalisation of theorem statements using large language models

Siddhartha Gadgil[*]
Department of Mathematics
Indian Institute of Science
Bangalore, India
gadgil@iisc.ac.in

Anand Rao Tadipatri[†]
Indian Institute of Science Education and Research
Pune, India
anand.tadipatri@students.iiserpune.ac.in

Ayush Agrawal
Microsoft Research
Bangalore, India
t-agrawalay@microsoft.com

Ashvni Narayanan,
London School of Geometry and Number Theory
London, UK
a.narayanan20@imperial.ac.uk

Navin Goyal
Microsoft Research
Bangalore, India
navingo@microsoft.com

Abstract

Mathematics formalisation is the task of writing mathematics (i.e., definitions, theorem statements, proofs) in natural language, as found in books and papers, into a formal language that can then be checked for correctness by a program. It is a thriving activity today, however formalisation remains cumbersome. In this paper, we explore the abilities of a large language model (Codex) to help with formalisation in the Lean theorem prover. We find that with careful input-dependent prompt selection and postprocessing, Codex is able to formalise short mathematical statements at undergrad level with about 65% accuracy for 120 theorem statements.

1 Introduction

Mathematics (definitions, theorems, proofs, remarks) as found in books and papers is written in a semi-formal style combining natural language with formal language in specialized notation. We refer to the language of this style of writing mathematics as *natural language* or NL. *Formalisation of mathematics* consists of writing mathematics in a *formal language* that can then be checked and manipulated by a computer. NL mathematics writing, while being more rigorous than writing in most other domains, falls far short of the standard of detail and rigour required for full formalisation. Formalisation is done with the help of *proof assistants*. A proof assistant consists of a formal language in which mathematical statements can be encoded along with a piece of software that assists in writing and checking proofs in the formal language up to the foundational axioms. See under Prompt in Figure 1 for some examples. Formalisation is an old endeavour that is thriving with several actively developed libraries of formalised mathematics for major proof assistants including Coq, Isabelle, Lean and Mizar. A major use of proof assistants is in software and hardware verification but here we are concerned with their applications in mathematics: checking formalised mathematics automatically

[*]http://math.iisc.ac.in/ gadgil/
[†]https://0art0.github.io/lambda-cube/

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on MATH-AI.

results in a much higher degree of confidence in the correctness of proofs. Formalisation promises to open up new possibilities in mathematical exposition, teaching, research and collaboration [Massot, 2021, Buzzard, 2022]; in addition, it can facilitate automated proof discovery, e.g. [Lample et al., 2022].

Formalisation of mathematics today poses a barrier to entry because of the need to learn to use proof assistants. *Autoformalisation* [Wang et al., 2018] is the task of (semi-)automatically turning a piece of mathematics in natural language into a formalised one. An autoformalisation tool that speeds-up formalisation or fully automates it would be of great value by enabling the above advantages of formalisation and opening up new ones [Szegedy, 2020].

Autoformalisation is challenging: mathematics retains much of the complexity of natural language while presenting additional challenges such as semantically mapping concepts in the informal description to those in the formal corpus [Ganesalingam, 2013, Massot, 2021]; and the the amount of formalised mathematics available is much smaller than code in major programming languages.

In this paper we worked with **Lean 4** [de Moura and Ullrich, 2021] – the latest version of the popular Lean theorem prover. Lean 4 is (in addition to an interactive theorem prover) a full-fledged programming language with a fast runtime. This allows a seamless integration of proofs, programs and meta-programs.The rapidly evolving Lean mathematical library (abbreviated mathlib) is one of the largest libraries of formal mathematics. mathlib is currently 226MB in size. mathlib is monolithic by design, ensuring that formalisations of different parts of mathematics can be combined easily. The resulting standardization of terminology in mathlib and its good coverage make Lean an attractive target for autoformalisation.

Our contributions. In this paper we apply a large language model (specifically, Codex) to the problem of autoformalisation. We focused on translating theorem statements of a form similar to docstrings of mathlib to theorems in Lean 4.

For the evaluation dataset, we chose 120 theorem statements at the undergrad level so that the relevant concepts (background theory and definitions) were mostly already in mathlib. Since mathlib is substantial (it has a significant fraction of undergrad mathematics curriculum apart from many advanced results), this is not a restriction. We focused on theorem statements at the undergrad level from various areas of mathematics. These statements tend to be more challenging for autoformalisation compared to mathematics competition problems studied in prior work [Wu et al., 2022] as they often assume more in terms of implicit context and draw from a much larger background [Wu et al., 2022].

We experimented with using input-dependent prompting, with mathlib as a database. Specifically, we chose our few-shot prompts to consist of theorem-docstring pairs from mathlib where the docstring is close in a sentence similarity metric to the statement to be formalised. We also experimented with filtering outputs generated at high temperatures by checking validity in Lean 4 and some other post-processing.

Our results showed that *there is a strong effect of both prompt engineering and selection, and even more when used in combination* and that *a reasonably large fraction are elaborated (i.e., give type-correct Lean terms) when both prompt engineering and selection is done* (the results improve further when more prompts are used). Further, we see that a high fractions of the completions that were elaborated were indeed correct, showing that elaboration is a good proxy measure for correctness.

In the context of autoformalisation, we are the first to use input-dependent prompting. Our use of elaboration for postprocessing is novel. Both of these are greatly facilitated by the availability of mathlib, and the nature of Lean 4, which gives easy access to its internals and in Lean 4 itself – the latter allowing shared code and avoiding context switching. Further the easy access of Lean 4 internals allows *efficient programmatic checking for elaboration*, which can hence be used to filter outputs at high temperatures and also gives a *scalable measure of performance* to refine prompt engineering and for fine-tuning.

Related work While the term *autoformalisation* was coined in Wang et al. [2018], the problem itself has a long history; see Wang et al. [2020]. Wang et al. [2020] applied deep learning-based methods to autoformalisation by treating it as a language translation problem. The recent work Wu et al. [2022] is closest to ours and stimulated our work. They considered statement autoformalisation in Isabelle/HOL using LLMs. For their quantitative results, their statements were from middle school

to undergrad mathematical competitions [Zheng et al., 2022]. These problems use only elementary concepts. Their quantitative studies are for fixed few-shot prompts. While a direct comparison with their results is not possible due to the use of different proof assistants and datasets, our method compares favourably with their method (fixed few-shot prompting with greedy decoding) as shown in the next section. Our input-dependent prompting is not applicable on their dataset due to the lack of availability of aligned data at the elementary level of statements in their datasets. Lean Chat is a fixed-prompt autoformalisation tool for Lean 3 theorem statements based on Codex.

Future work. Using docstrings from mathlib in the present form does not give adequate examples of complex LaTeX formulas and of some mathematical idioms. An additional database of prompts targeting these could address this. Further, we can make use of Lean's easily extensible syntax to incorporate more mathematical notation.

Better equality testing for theorem statements will also result in better filtering. Unlike program synthesis, for theorem autoformalisation, there is no obvious counterpart of unit tests. Better equality testing with the correct Lean formal statement, however, can serve the role of unit tests.

2 Evaluation datasets

We used three test sets with 40 natural language statements each. The natural language statements were of the same form as typical doctrings in mathlib: single sentences often with Lean code fragments (including names and formulas not in LaTeX but in unicode) enclosed in backticks. We call such strings **docstring-style** strings.

Our first set consisted of mathematical theorems (and some conjectures) in areas well-represented by mathlib, such as undergraduate-level number theory, group theory and topology. These were chosen to represent various areas of mathematics and various levels in a typical undergraduate curriculum.

The other two sets were designed to minimize contamination due to similar results being in the training of Codex. Our second set consisted of what we called *silly statements*, such as *every vector space with dimension `2` is finite dimensional*. While being true, these were easy and/or absurdly specific, so unlikely to appear in this precise form anywhere else. We created this set based on theorems proved in mathlib – with a *silly* versions of a statement mostly involving the same concepts and terminology as the statement but with the hypothesis or conclusion modified so that the silly version is obvious and/or bizarrely specific.

The third set consisted of *false statements*: these obviously cannot appear in any library. The statements in this set were closely related to those in mathlib or our first dataset: for example, while our first dataset had the statement *every field is a ring* our third dataset had its (false) converse *every ring is a field*. In general given a statement in mathlib or our collection of theorems, we created a related false statement by taking the converse, weakening the hypothesis or strengthening the conclusion, ensuring that the modified statement is false.

3 Techniques

We used Codex to translate an input text in natural language to code in Lean 4. Codex takes as input a prompt and returns *completion(s)*, i.e., a guess as to how the text continues until a given stop token (in our case `:=`) or token limit. We generated a prompt from the input text and post-processed completions as described below. Figure 1 is an example of a prompt, the initial result (with one completion shown) and the result after post-processing. We remark that this example needs prompt engineering, as we see in Section 4.

Prompt engineering. Given an input text to be translated, we chose example prompts from mathlib whose docstrings are similar to the input text. We used two notions of similarity: proximity in sentence embeddings (described further in Section B.4) and keyword matching (described further in Section B.5), with the number of sentences chosen by the user. This style of prompt design appears in the previous work, e.g., Jain et al. [2022]. The docstrings and the corresponding Lean code were extracted from mathlib documentation. From these a prompt was constructed as in Figure 1 following a template, with the prompt consisting of the example doc-strings followed by theorem statements essentially in the same syntax as Lean Code and doc-strings, followed by the statement to be translated in the format of a doc-string and an incomplete line with just the word "theorem".

Input text: "If a vector space has dimension '2' then it is finite dimensional."

Prompt:

```
/-- If a vector space has a finite basis, then it is finite-dimensional. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] {ι : Type w} [fintype ι] (h : basis ι K V) :
  finite_dimensional K V :=

...

/-- A vector space has dimension at most '1' if and only if there is a single vector of which all vectors are multiples. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] : module.rank K V ≤ 1 ↔ ∃ (v₀ : V), ∀ (v : V), ∃
  (r : K), r • v₀ = v :=

/-- If a vector space has dimension '2' then it is finite dimensional. -/
theorem
```

Codex Completion:

```
{K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] (h : module.rank K V = 2) : finite_dimensional K V
```

Post-processed code in Lean 4:

```
theorem ∀ {K : Type u} {V : Type v} [inst : DivisionRing K] [inst_1 : AddCommGroup V] [inst_2 : Module K V],
Module.rank K V = 2 → FiniteDimensional K V
```

Figure 1: Example of a prompt, the initial result and the result after post-processing. Part of the prompt was elided to save space; full prompt appears in Appendix B.1.

Post processing. Lean 4 code is compiled in two phases: a **parser** converts a string into a *Syntax* object, and an **elaborator** converts a *Syntax* object into a type-correct expression. The elaboration step is a much stricter analogue of type-checking in a strongly-typed language. It is roughly a formal analogue of supplying all the implicit details in an NL theorem statement. Lean 4 is unique among proof assistants in being implemented in Lean 4 and providing an interpreter API, which facilitates our implementation.

We parsed the Codex completions, translated from Lean 3 to Lean 4 and auto-corrected (as described in Section B.2) to obtain Syntax objects corresponding to (syntactically valid) completions. We attempted to elaborate each of these. We see that restriction to completions which are successfully parsed and elaborated gives a **strong filter**.

4 Results

We tested the effects of the prompt engineering and post-processing as well as the final quality of translations for the datasets described in Section 2.

Success rates for the Elaborator. We begin with quantitative results showing the utility of both prompt engineering and elaboration filtering for the datasets described in Section 2. By elaboration filtering we mean that of the many (typically 15-20) completions returned by Codex, we only consider those which are successfully parsed and elaborated. We emphasize that by using the features of Lean 4 the elaboration filtering was programmatic and efficient (which would not be case, for instance, if each completion was used to generate a program which was checked by an external compiler).

We summarize the number of statements for which some completion was elaborated for each of the three sets of statements in Table 1. For each set, we considered results with 4 fixed prompts (those used by Lean Chat) and 4 prompts chosen by sentence similarity. For each of these cases we considered answers chosen greedily (i.e., temperature 0 and 1 completion) and those obtained by choosing several completions at temperature 0.8 with filtering and selection. We made three runs for each configuration, and the result reported is the *median*. We also ran a configuration with the Codex recommended default temperature 0.2 and with fixed prompts. The results of this are included in parentheses in the entries for the greedy case. As 11 of the theorem statements were present in mathlib we also ran all the configurations excluding these and obtained similar results as above: in particular 23 of the 29 statements were elaborated with prompt engineering and selection. We see in the next section that elaboration is a good proxy measure for accuracy. Thus, we can justify the claims made in the Introduction.

	Theorems		Silly Statements		False Statements	
	Fixed	Input-dependent	Fixed	Input-dependent	Fixed	Input-dependent
Greedy	20 (18)	21	19 (21)	28	15 (16)	23
Filtered	25	29	29	34	24	30

Table 1: Numbers of elaborated statements; numbers in parenthesis are for temperature 0.2 (instead of 0) with one completion

	false statements	silly statements	theorem statements
Elaborated	**32**	**34**	**33**
Correct	21	26	30
Some correct	28	32	30
All wrong	4	2	3

Table 2: Correctness of elaborated statements

The example in Figure 1 illustrates the effect of prompt engineering. None of the 15 completions (per run) were elaborated in the three runs with the fixed (Lean Chat) prompts. The completions often used the wrong name from mathlib or assumed a definition was at a different level of abstraction (e.g., modules versus vector spaces) from that of mathlib. We also saw that a larger number of examples did lead to more sentences being elaborated, but the effect was not strong enough to quantify robustly.

Correctness of elaboration. We analysed how often completions that were elaborated were correct. In the case where more than one completion was elaborated, we considered both whether the chosen completion was correct and whether any of the elaborated completions were correct.

For each of the three sets, we considered a configuration with high temperature and prompt engineering – specifically, we considered the configuration with the highest number of elaborated statements[3], as our goal was to test elaboration as a proxy measure for correctness. We manually checked the correctness of the completions for the elaborated completions, as reported in Table 2.

Further, the statements where all completions were wrong involved some concept for which we had very few prompts available, in part due to the incomplete state of the binary port of mathlib, also suggesting that elaboration is a good proxy measure.

References

Kevin Buzzard. What is the point of computers? a question for pure mathematicians. In *International Congress of Mathematicians*, 2022. URL `https://arxiv.org/pdf/2112.11598.pdf`.

Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Mário Jorge, Célia Nunes, and Adam Jatowt. A text feature based automatic keyword extraction method for single documents. In *European conference on information retrieval*, pages 684–691. Springer, 2018.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL `https://doi.org/10.1007/978-3-030-79876-5_37`.

Mohan Ganesalingam. *The Language of Mathematics - A Linguistic and Philosophical Investigation*, volume 7805 of *Lecture Notes in Computer Science*. Springer, 2013. ISBN 978-3-642-37011-3. doi: 10.1007/978-3-642-37012-0. URL `https://doi.org/10.1007/978-3-642-37012-0`.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1219–1231,

[3]All configurations had temperature 0.8; we used 10 sentence similarity prompts and 4 keyword based prompts and obtained 15 completions for theorems and silly statements and used 12 sentence similarity prompts and 8 keyword based prompts and obtained 20 completions for false statements.

New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510203. URL https://doi.org/10.1145/3510003.3510203.

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *CoRR*, abs/2205.11491, 2022. doi: 10.48550/arXiv.2205.11491. URL https://doi.org/10.48550/arXiv.2205.11491.

Patrick Massot. Why formalize mathematics. 2021. URL https://www.imo.universite-paris-saclay.fr/~pmassot/files/exposition/why_formalize.pdf.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL http://arxiv.org/abs/1908.10084.

Christian Szegedy. A promising path towards autoformalization and general artificial intelligence. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020. doi: 10.1007/978-3-030-53518-6_1. URL https://doi.org/10.1007/978-3-030-53518-6_1.

Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2018. doi: 10.1007/978-3-319-96812-4_22. URL https://doi.org/10.1007/978-3-319-96812-4_22.

Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *International Conference on Certified Programs and Proofs*, 2020.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *NeurIPS 2022*, abs/2205.12615, 2022. doi: 10.48550/arXiv.2205.12615. URL https://doi.org/10.48550/arXiv.2205.12615.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A The Lean interactive theorem prover

Lean is an *Interactive Theorem Prover* based on the *Calculus of Inductive Constructions* (CIC), i.e., a system in which results are proved interactively and checked by the system. Lean has a large number of features that automate or assist in the discovery of proofs, but the verification is done by a small *trusted kernel* and can also export proofs to be verified by external checkers (at least three such independent checkers have been implemented for Lean). Other interactive theorem provers like Coq, Isabelle and Mizar also are built on similar principles and have been used for formalisation of mathematics.

The latest version of Lean, Lean 4, is a full-fledged programming language with a fast runtime in addition to being an interactive theorem prover. Lean 4 is largely implemented in Lean 4, and gives easy access to its internals to Lean 4 programs – the latter allowing shared code and avoiding context switching.

The Lean mathematical library (mathlib) includes most of the standard undergraduate curriculum. Many advanced results have also been formalised building on mathlib. mathlib is monolithic by design, ensuring that formalisations of different parts of mathematics can be combined easily. The presence of this library to use and also the resulting standardization of terminology makes Lean an attractive target for autoformalisation.

The structure of typical proofs in Lean differs from typical ones in the literature. Mathematical proofs in the literature usually use *forward reasoning*, where a series of conclusions are deduced starting with the hypotheses from previous conclusions and known results. A notable exception is proof by induction, where we begin with the goal and reduce to sub-goals for the base case and the induction step. Reasoning starting with goals is called *backward reasoning*.

In Lean (and similar systems), proofs can use both forward and backward reasoning. However, backward reasoning allows for much more powerful automation within the *tactic mode*, at the cost of readability. Lean has powerful tactics like `rw` (applies an equation or if and only if statement) and `apply`(tries to match the goal against the conclusion of the lemma being used) to deal with these.

We illustrate the different styles of Lean proofs by proving in Lean in various ways the result $3 < 7$ using only the results $\forall n \in \mathbb{N}, 0 < n + 1$ and $\forall n, m \in \mathbb{N}, n \leq m \implies n + 1 \leq m + 1$. In Lean these are the theorems `Nat.zero_lt_succ` and `Nat.succ_lt_succ`. Four proofs (in Lean 4, which also work in Lean 3) are shown in Figure 2.

```
theorem three_lt_seven₁ : 3 < 7 :=
    have l₁ : 0 < 4 :=  Nat.zero_lt_succ
    3
    have l₂ : 1 < 5 :=  Nat.succ_lt_succ
    l₁
    have l₃ : 2 < 6 :=  Nat.succ_lt_succ
    l₂
    Nat.succ_lt_succ l₃
```

```
theorem three_lt_seven₂ : 3 < 7 :=
Nat.succ_lt_succ (
Nat.succ_lt_succ (Nat.succ_lt_succ
    (Nat.zero_lt_succ 3)))

theorem three_lt_seven₃ : 3 < 7 :=
by  repeat (apply Nat.succ_lt_succ)
    apply Nat.zero_lt_succ

theorem three_lt_seven₄ : 3 < 7 := by
    decide
```

Figure 2: Four proofs of $3 < 7$ in Lean

The first proof is a typical forward reasoning proof making deductions from known results and previous deductions. The second proof is simply this in a more concise form, something a person will typically not be able to write correctly except in the simplest cases (indeed the proof given here was obtained by using the `#print` command on the third proof). Note that these are complete proofs in the foundations of Lean.

The third and fourth proof use backward reasoning in tactic mode. Tactics are powerful algorithms for finding proofs. In the third proof we use the `apply` tactic to apply results, with `repeat` saying that a tactic is to be applied as long as it is valid. The fourth proof uses the `decide` tactic which depends

on a decision procedure. with proofs being implemented for a class of problems. (in this case, as we checked from the Lean source, the decision procedure only uses the results mentioned above in proofs).

It is evident that the above backward proofs are more concise and will be easier for a user to produce. However, in practice a complex mathematical proof in Lean will have *mixed* forward and backward reasoning, with forward reasoning taking the form of a sequence of lemmas (in the form of `have` statements) leading to the main theorem and backward reasoning used in the proof of each lemma.

B Theorem Statement Translation : Further Details

We sketch more details of the various steps in translating sentences.

B.1 Full Example prompt

The full prompt for Figure 1 is in Figure 3. As mentioned earlier, no completion elaborated when we used fixed prompts.

Input text: "If a vector space has dimension ‘2‘ then it is finite dimensional."

Prompt:

```
/-- If a vector space has a finite basis, then it is finite-dimensional. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] {ι :
    Type w} [fintype ι] (h : basis ι K V) : finite_dimensional K V :=

/-- A finite dimensional space is nontrivial if it has positive ‘finrank‘. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] (h : 0
    < finite_dimensional.finrank K V) : nontrivial V :=

/-- A finite dimensional space that is a subsingleton has zero ‘finrank‘. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] [h :
    subsingleton V] : finite_dimensional.finrank K V = 0 :=

/-- A vector space has dimension at most ‘1‘ if and only if there is a single vector of which all
    vectors are multiples. -/
theorem {K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] :
    module.rank K V ≤ 1 ↔ ∃ (v₀ : V), ∀ (v : V), ∃ (r : K), r • v₀ = v :=

/-- If a vector space has dimension ‘2‘ then it is finite dimensional. -/
theorem
```

Codex Completion:

```
{K : Type u} {V : Type v} [division_ring K] [add_comm_group V] [module K V] (h :
  module.rank K V = 2) : finite_dimensional K V
```

Post-processed code in Lean 4:

```
theorem ∀ {K : Type u} {V : Type v} [inst : DivisionRing K] [inst_1 : AddCommGroup V]
  [inst_2 : Module K V],
Module.rank K V = 2 → FiniteDimensional K V
```

Figure 3: Complete prompt used in Figure 1.

B.2 Parsing, translation and auto-correction

Given a Codex completion, we first (attempt to) parse this and extract *identifiers* from the syntax. These are *translated* and *auto-corrected* before re-parsing. The translation step is necessary as the

prompt data we have available is in Lean 3, as is most of the data in GitHub on which Codex is trained. Thus the completions usually use Lean 3/mathlib terminology. Using a prebuilt dictionary, we translate the Lean 3/mathlib identifiers to those used by the binary port (binport) of mathlib, with auto-correction attempted for those that do not have valid translations. Both the dictionary and auto-correction are based on transformations of two forms: case transformations (for example camel-case versus snake-case) and dropping or adding segments of the form `is` or `has`.

B.3 Selection

If more than one completion is correctly elaborated (which is typical when at least one completion is elaborated), we select the best completion by *voting*. Namely, we first group elaborated completions together into groups whose members can be proved to be equal using a certain tactic. The tactic we use is one that slightly extends *reflexivity* (i.e., *definitional equality*). The chosen answer is the first member of the largest group. In practice, as the present tactic for proving equality is weak, in most cases this simply picked the first completion of Codex that is valid (i.e., that elaborated).

B.4 Sentence similarity

We use Sentence-Similarity library [Reimers and Gurevych, 2019] for calculating the sentence embeddings of the doc-strings. We use all-mpnet-base-v2 model, a pretrained transformer model finetuned over 1 Billion sentence pairs from multiple datasets. This model provided the best quality embeddings among the hosted models on the library at the time of writing this paper. We compute the cosine similarity of the sentence-embeddings generated from the input docstring with the collection of mathlib docstrings and select the top k similar docstrings based on the similarity scores and their corresponding Lean statements for the example prompts.

B.5 Keyword-based prompting

The purpose of input-dependent prompting for theorem statements is to retrieve a collection of examples that contain all the relevant details to formalise a given statement, and this is achieved to a large extent using sentence similarity. However, in optimising for overall similarity, this approach may leave out smaller details that are nevertheless crucial for formalising the statement correctly. To address this, we introduce a method of prompting based on keywords that complements sentence-similarity based retrieval. We used the YAKE keyword extraction tool [Campos et al., 2018] to extract the keywords from mathlib and store them in a convenient format. When preparing the prompt for formalising a sentence, we extract the keywords and retrieve a few examples for each keyword, in addition to using sentence similarity.