Tackling Students' Coding Assignments with LLMs

Adam Dingle
dingle@ksvi.mff.cuni.cz
Charles University
Prague, Czechia

Martin Kruliš
krulis@d3s.mff.cuni.cz
Charles University
Prague, Czechia

ABSTRACT

State-of-the-art large language models (LLMs) have demonstrated an extraordinary ability to write computer code. This ability can be quite beneficial when integrated into an IDE to assist a programmer with basic coding. On the other hand, it may be misused by computer science students for cheating on coding tests or homework assignments. At present, knowledge about the exact capabilities and limitations of state-of-the-art LLMs is still inadequate. Furthermore, their capabilities have been changing quickly with each new release. In this paper, we present a dataset of 559 programming exercises in 10 programming languages collected from a system for evaluating coding assignments at our university. We have experimented with four well-known LLMs (GPT-3.5, GPT-4, Codey, Code Llama) and asked them to solve these assignments. The evaluation results are intriguing and provide insights into the strengths and weaknesses of the models. In particular, GPT-4 (which performed the best) is currently capable of solving 55% of all our exercises and achieved an average score of 86% on exercises from the introductory programming course (using the best of five generated solutions).

CCS CONCEPTS

• **Computing methodologies → Natural language processing**; • **General and reference → Evaluation**; • **Applied computing → Education**.

KEYWORDS

LLM, large language model, coding, programming, student assignment, teaching

ACM Reference Format:
Adam Dingle and Martin Kruliš. 2024. Tackling Students' Coding Assignments with LLMs. In *2024 International Workshop on Large Language Models for Code (LLM4Code '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643795.3648389

1 INTRODUCTION

Recent breakthroughs in large language models (LLMs) have caused a wave of excitement that has splashed well beyond the scientific community. The emergent abilities of LLMs [19] such as language translation, creative text generation, and solving programming tasks have raised many questions regarding their limits and capabilities. In this research, we focus on the ability to solve coding tasks, and so we have performed extensive experiments with real assignments solved by university students.

It is well known that LLMs such as ChatGPT can handle smaller coding tasks [2, 4]. In fact, AI-powered tools designed to assist programmers by generating or fixing small portions of code (such as Copilot [20]) are generally available. Although these tools may significantly improve coding efficiency in the hands of an experienced programmer, their benefits for the learning process in the domain of computer science are still unclear [3]. On one hand, having a tireless assistant that is capable of correcting student mistakes in real time is a tempting prospect. On the other hand, relying blindly on AI tools may dull the programmer's wits and may lead to new types of mistakes since LLMs are not infallible. Furthermore, it is undesirable if an AI tool can solve the student's work completely since it may easily become a tool for cheating.

In this paper, we focus on the empirical evaluation of LLMs from the perspective of solving coding assignments designed specifically for education. We attempt to answer the following questions:

(1) Can LLMs solve assignments completely in place of a student?
(2) How complex are the assignments that LLMs can solve?
(3) Do currently available LLMs differ greatly in their performance on this task?

One significant contribution of our research is that we use a dataset consisting of real coding exercises assigned to students at our university. The exercises were selected from various courses covering the first seven semesters of instruction. They vary greatly in complexity, are written in 10 different programming languages, and focus on various aspects of programming, such as coding principles, algorithms, data structures, and specific technologies.

The paper is organized as follows. Section 2 describes our dataset of exercises and their evaluation. Section 3 elaborates on the LLMs and how we use them to solve the exercises. Our experimental results and their interpretations are summarized in Section 4. Section 5 presents related work and Section 6 concludes our paper.

2 DATASET

Our dataset was collected from the ReCodEx system[1] deployed at our university. It is a web application in which teachers may prepare coding exercises and assign them to students. Each exercise has a specification (formatted text in Markdown) and optionally some associated files (e.g. images or template source code). Furthermore, the teacher provides test cases for evaluating the correctness of solutions along with time and memory limits for each test.

The students are supposed to follow the exercise specifications and submit their solutions in the form of source code. ReCodEx compiles each solution, executes it for each test case, and imposes

[1] https://github.com/recodex

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LLM4Code '24, April 20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0579-3/24/04
https://doi.org/10.1145/3643795.3648389

the time and memory limits. The produced outputs are compared with the outputs provided by the teacher to determine whether the solution is working correctly.

At present, ReCodEx supports 17 programming languages and is being used in over 30 courses with various types of exercises ranging from simple basic coding tasks to fairly advanced exercises in specialized classes (e.g., designing machine learning models in TensorFlow or writing a custom compiler).

One of the greatest benefits of ReCodEx is quick feedback to students: simple assignments are evaluated in a matter of seconds. We exploit this advantage in our work by submitting large numbers of LLM-generated solutions to ReCodEx, which provides quick and impartial evaluations.

2.1 Data extraction and filtering

At present, there are nearly 2,000 exercises stored in ReCodEx. Unfortunately, the exercises are of varying quality — some have vague specifications or mistakes in testing data, and some may be completely or nearly identical to others. We imposed a series of filtering rules to select only exercises that we considered sufficiently good for our experiments:

- At least one student solved the exercise (100% correct) in the past two years, so it is valid and sufficiently up-to-date.
- The specification has at least 150 characters[2].
- The specification does not contain hyperlinks to external materials which are essential for solving the assignment.

Some of our exercises support multiple programming languages. For simplicity, we decided that each exercise should be solved by only one programming language in our experiments and we selected the language that was used by the most students in the past two years (for each exercise).

Many of our exercises have specifications written both in Czech[3] and in English. English may be better suited for LLMs since it is the language on which they are trained the most. On the other hand, almost all our teachers are native Czechs so the English specifications may be of lower quality. We decided to choose the longer of the available specifications for each exercise since it presumably describes the problem more thoroughly.

2.1.1 Deduplication. Many of our teachers often duplicate exercises, so we put in a considerable effort to reduce the duplicates in our dataset. We automatically eliminated exercises with identical specifications. We also manually examined exercises with identical names, or whose specifications were very similar (measured using the normalized Levenshtein distance), and eliminated those we judged to be duplicates. This deduplication process eliminated 113 exercises. We manually eliminated 28 further exercises since their specifications were found to be incomplete or otherwise unsuitable.

2.2 Basic properties and availability

The final dataset holds 559 individual exercises from 17 distinct courses spanning 7 semesters of our computer science curriculum up to the first semester of our master's program. The distribution is not even: 365 exercises are from the first year of study, 177 from the second year and only 17 from the remaining years.

The dataset covers 10 different programming languages. Most of the exercises (280) are in Python since we use that language in our first programming classes. The second most frequent language is C# (105 exercises), and the third is C/C++ (67). It is worth mentioning that we have 36 exercises in Prolog and 27 in Haskell; these languages are unusual in LLM coding evaluation datasets. Finally, we include 5 C++ exercises from a class about Arduino programming which are also highly uncommon in evaluation datasets.

The complete list of exercises along with a more detailed overview is available in an associated Git repository[4]. We have not placed the exercise specifications into the repository since many of them should not be publicly available to students (especially exercises used in exams), and also so that the models cannot use them for training (since we would like to continue to test with this dataset in the future). However, we are willing to share them privately under nondisclosure terms.

3 EXPERIMENTS

We evaluated four LLMs on the exercises in our dataset:

GPT-3.5 [15] is a family of models from OpenAI, with an initial release in March 2022. The very popular and well-known web service ChatGPT uses a fine-tuned model from this family. We specifically tested `gpt-3.5-turbo-1106` ("Updated GPT-3.5 Turbo"), released on November 6, 2023.

GPT-4 [16] is a family of models from OpenAI, with an initial release in March 2023. The ChatGPT Plus service is based on a model from this family. We tested `gpt-4-1106-preview` ("GPT-4 Turbo"), released on November 6, 2023.

Codey [11] is a family of models from Google, first released in May 2023. Codey models are built on the large foundation model PaLM 2 and fine-tuned on a large dataset of permissively licensed source code. We tested the `code-bison-32k` model, a preview version that was current when we made our queries via Google's API in November 2023.

Code Llama [17] is a family of models from Meta, fine-tuned from its Llama 2 model on a large dataset of source code and released in August 2023. Unlike the other models above, Code Llama is openly licensed with weights publicly available. We used the `codellama-34b` model, which has 34 billion parameters.

Table 1 shows some characteristics of these models.

Many of our exercises have descriptions in Czech. We found that all the models could solve exercises in Czech, and in fact every model's average score on exercises in Czech was higher than on exercises in English. This does not necessarily mean that the models are as capable of understanding Czech as well as English, since our Czech exercises may have been easier on average. Nevertheless, our overall impression was that the choice of language had little impact on the performance of any of these models.

We were originally unsure whether all these models would be able to generate code in all of the programming languages of our exercises. In the end, we found that all models were able to solve at least one exercise in every language, except that only GPT-3.5 and

[2]This threshold was determined by manual examination of the shortest specifications.
[3]The native language of most of our students.
[4]https://github.com/medovina/llm4code-2024-recodex (`exercises` folder)

Model	Parameters	Fine-tuned for code?	Context window
`gpt-3.5-turbo-1106`	175B?	No	16K tokens
`gpt-4-1106-preview`	1.8T?	No	128K tokens
`code-bison-32k`	340B?	Yes	32K tokens
`codellama-34b`	34B	Yes	100K tokens

Table 1: Characteristics of the models we evaluated. Question marks indicate values that have not been confirmed by the models' authors.

GPT-4 solved any Arduino C++ exercises, and only GPT 4 solved any Node.JS (JavaScript) exercises.

We queried the GPT and Codey models through OpenAI's and Google's APIs. Facebook does not offer an API for Code Llama. We chose to query it through Replicate, a Web service that hosts openly available models.

All these models' APIs offer a *'temperature'* parameter for controlling the degree of stochasticity in the output, as well as other parameters for controlling output qualities such as repetitiveness. In all our queries we used the default values of these parameters.

3.1 Exercise input format

The exercises in our dataset have descriptions with Markdown formatting. We decided to submit the exercises to the models including this formatting, rather than stripping it and submitting it in plain text. That is because we found the models were generally not confused by Markdown formatting and the formatting can be useful for understanding exercise structure. We observed only one exercise where Markdown formatting seemed to cause a significant ambiguity. This exercise contained the following text (which we have translated from Czech):

```
The name of the input file is __file.txt.gz__
```

The double underscores above are Markdown formatting intended to emphasize the filename; however, models sometimes generated solutions in which the input filename literally included the underscore characters.

Many of our exercises include attachment files, which may hold images with diagrams, code templates to be filled in, or sample data for smoke testing the solution. When submitting exercises to the models, we ignored attachments with images, because the models we were evaluating would not be able to understand them anyway[5]. We appended the text-based attachments to the exercise specification, preceding each attachment with a header line such as "`=== data.in ===`" indicating its filename.

In some exercises, the attachments are very long, and would in aggregate exceed the models' token limits. We decided to submit these exercises anyway along with as many attachments as we could reasonably inline in the query. Specifically, we ignored all attachments that were longer than $8K$ characters, as well as any other attachments that would cause the total size of the query to exceed 70% of the token limit for the model being queried (reserving 30% for the model's response).

[5]OpenAI has recently released a multimodal model `gpt-4-version-preview` that can understand images, but we did not test it.

3.2 Output format

Some of the exercises require solutions with multiple source files, and in some cases, those files must have certain specific names (such as C++ headers which are included by name). So we needed to ask the models to produce output in a format that could be unambiguously parsed into a set of named source files. We could have chosen a structured format such as JSON, in which the filename and source text could be object attributes. However, we decided to ask the models to produce a simpler format in which each source file's text is preceded by a header line containing the file's name, such as "`=== SortArray.java ===`". This format avoids issues with escaping text inside JSON strings or with the JSON structure itself, which could be potential sources of error.

We asked the models to output only source code, without any explanatory text. We did this because we found they have a tendency to intersperse code and textual output. Such mixing would prevent us from finding the source code in the output unless we used heuristics to distinguish code from text, which could be complicated to implement reliably for all our programming languages.

Based on these decisions, here is the exact prompt that we used when requesting a solution in Python:

```
Write a program in Python that solves the
given exercise. Your program should not prompt
the user for input.
Your program will consist of one or more source
files. For each source file, output a filename
in the format "=== filename ===", followed by
the file's contents. After you have output all
source files, output a final line "====".
Do not output any explanatory text.
```

We used this same English-language prompt even for exercises with specifications in Czech.

Despite our request not to produce explanatory text, the models sometimes did so anyway, and in some cases, this output was interspersed with code. For example, GPT-4 often outputs explanatory text when it judges an exercise to be too difficult to solve completely and explains that it is only solving a simplified version of the requested task.

3.3 Few-shot prompting

In addition to describing our desired output format in our prompt, in each request, we also provided examples of exercises to be solved along with their intended solutions. This approach is called *few-shot prompting* and is intended to increase the quality of the models' output. Without these examples, the models had a tendency to produce code with extraneous behaviors such as hard-coding particular test

cases or producing extra unwanted output. The examples reduced but did not entirely eliminate such behaviors.

Our example exercises and solutions were specific to each programming language from our exercise dataset. For most languages, we provided two example exercises: one involving reading input and writing output, and one in which the solution consists only of functions to be written, with no main function or I/O. The examples were trivial (e.g., writing a function to add two numbers).

The GPT Chat Completions API allows a request to contain a series of messages marked as either "user" messages, containing an instruction to the model, or as "assistant" messages, containing an expected response. These are explicitly intended for few-shot prompting, and so we used them for providing our examples to GPT. The Codey API does not contain this feature, so we invented custom delimiters (`[EXERCISE]` and `[RESPONSE]`) that we used for few-shot prompts to Codey.

We attempted to encode few-shot prompts for Code Llama using certain tags (e.g. `[INST]` and `«SYS»`) mentioned in Meta's Code Llama documentation. For this initial experiment, we used the `codellama-34b-instruct` model, which is fine-tuned for instruction input and should support these tags. However, the results were unsatisfactory: we found that the model would not consistently produce output in our expected format. So we fell back on using the `codellama-34b` model with the custom delimiters that we used for prompting Codey, which worked much better. We made all our queries to Code Llama through Replicate's API, which could conceivably have had some effect on these tags.

3.4 Number of queries

The models we evaluated are stochastic, so we were interested in gathering multiple solutions for each exercise. We quickly realized that the GPT models were much stronger than Codey and Code Llama, and therefore of more interest. In the end, we decided to gather 5 solutions to each exercise for GPT-3.5 and GPT-4, and only 1 solution to each exercise for Codey and Code Llama.

4 RESULTS

In this section, we present a selected subset of results with our explanations. All collected data along with the scripts and additional plots are available in an associated Git repository[6].

4.1 Metrics

In this section, we define the metrics avg@k and pass@k that we used for evaluating models' solutions to our exercises.

When ReCodEx evaluates a solution to an exercise, it assigns it a *score*, which is the fraction of test cases that passed. (In some exercises test cases have different weights, which are considered in computing this fraction.)

The models are stochastic, so for any exercise, they may produce solutions with various scores. For a single exercise, we define the avg@k to be the maximum score earned by k independent solutions from a model for the exercise. The metric avg@k over a set of exercises is defined to be the average value of avg@k over the set.

For GPT models we collected 5 solutions for each exercise. When computing avg@k for any exercise when $k < 5$, we computed the average of the maximum scores of all $\binom{5}{k}$ possible subsamples of size k. That produces a statistic with lower variance than if we had chosen only one arbitrary subsample of this size for each exercise.

We say that a solution *passes* if its score is 1.0, i.e. it passes all the test cases. For a single exercise, we define the metric pass@k to be 1.0 if any of k independent solutions from a model have passed, otherwise 0.0. The metric pass@k over a set of exercises is defined to be the average value of pass@k over the set.

Our definition of pass@k matches that of other researchers. [7] The metric pass@k is commonly used in evaluating the performance of LLMs on benchmarks such as HumanEval [7] and MBPP. Our metric avg@k is more lenient in that a model can earn points for a solution that does not pass all the test cases. This reflects the grading model that most of our teachers use, in which a student may earn partial credit for such a solution. The statistic avg@k for a model represents the score that such a student could expect to achieve by asking the model for k different solutions to an exercise, then submitting all these solutions to ReCodEx, which will award the student the highest score of all submitted solutions.

In most of the following results, we consider the average score avg@k. In some results, we also present the pass rate pass@k.

4.2 All exercises

Figure 1 shows the average score and pass rate achieved by each model over all exercises in our entire dataset. We see that the GPT models are much stronger than Code Llama and Codex and that GPT-4 significantly outperforms GPT-3.5. These relative rankings will be evident throughout the results in the following sections.

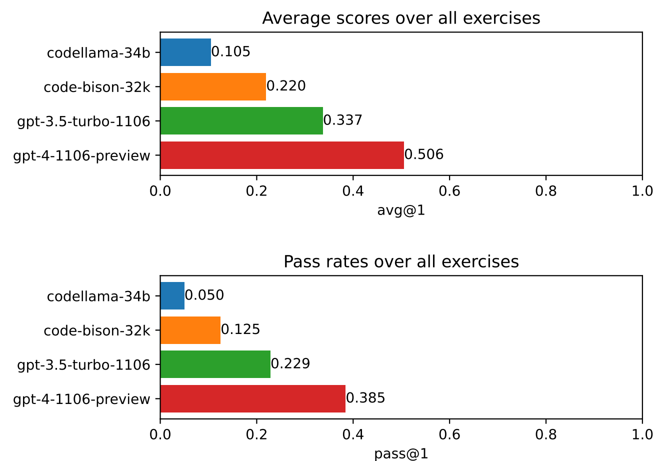

Figure 1: The average score (avg@1) and pass rate (pass@1) achieved by each model over all exercises in the dataset.

Figure 2 shows how the statistics avg@k and pass@k vary with increasing k for GPT models, i.e. how much performance improves with repeated sampling. We observe that taking even just two solutions (instead of one) to each exercise dramatically increases average scores (e.g., by over 9% for GPT-4).

[6] https://github.com/medovina/llm4code-2024-recodex

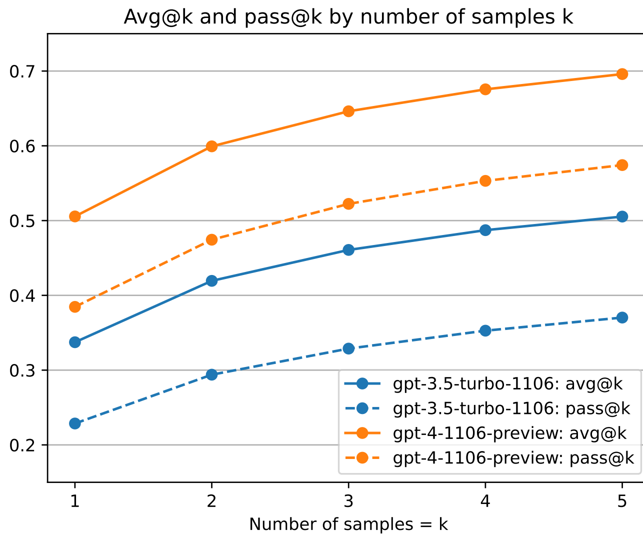

Figure 2: Average scores and pass rates of GPT-3.5 and GPT-4 over all exercises, by the number of samples k for each exercise.

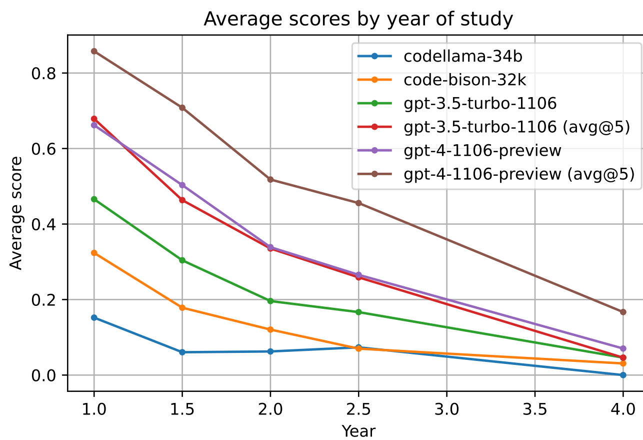

Figure 3: Average scores by course year. On the x-axis, 1.0 **is the first semester (1st year of BSc program),** 1.5 **is the second semester,** 2.0 **is the first semester of the 2nd year, and so on.** 4.0 **is the first year of master's studies. (The graph excludes data for semesters in year** 3**, since we have only a few exercises for those semesters, too few for a meaningful average.)**

4.3 Average scores by year

Figure 3 shows each model's average scores over all courses that are normally taken by students in a given semester of their bachelor's or master's studies. The graph shows a consistent downward trend. We also see that GPT-3.5 with 5 samples consistently performs about as well as GPT-4 with a single sample. Our dataset contains many more exercises from the earlier semesters of study than from later semesters (since the assignments in later semesters are more complex). In fact over half of the exercises in our dataset are from first-semester courses. For this reason, the average scores over all

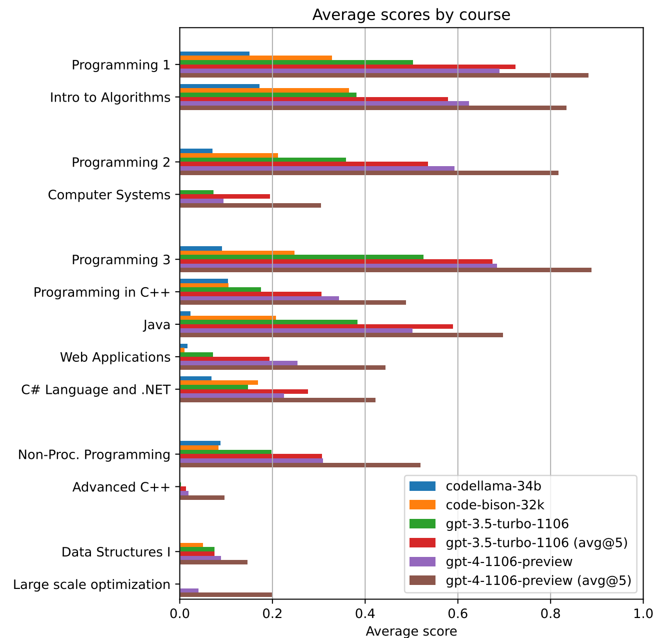

Figure 4: Average scores by course. Courses offered in a single semester are grouped together, with first-semester courses at the top. (This chart excludes some courses with only a small number of exercises.)

exercises in the previous section are much higher than the average scores in the median year of study (2.5) that we see here.

4.4 Average scores by course

Figure 4 shows each model's average score in different courses that we offer. The minimum required homework score to pass a class varies among teachers at our university. In our introductory classes (Programming 1/2/3 and Introduction to Algorithms) many teachers require a minimum score of 70%. It seems likely that a student could use GPT-4 to generate a set of homework solutions that would pass these classes, especially if they query it several times for each assignment. On the other hand, for most of our courses in the second year of study and beyond, even the repeated use of GPT-4 would probably not be enough for a passing score.

4.5 Average scores by programming language

Figure 5 shows each model's average score for all programming languages in our exercise set. The performance among programming languages differed greatly. For example, the average scores in C++ and Haskell are relatively low. However, our assignments in these languages are in courses in later semesters and may be inherently more difficult, so this does not necessarily mean that producing code in these languages is more difficult for the models.

4.6 Lines of code

In this section, we consider the number of lines of code in each model's solutions. In counting lines, we excluded blank lines and single-line comments in all languages. We did not bother to exclude

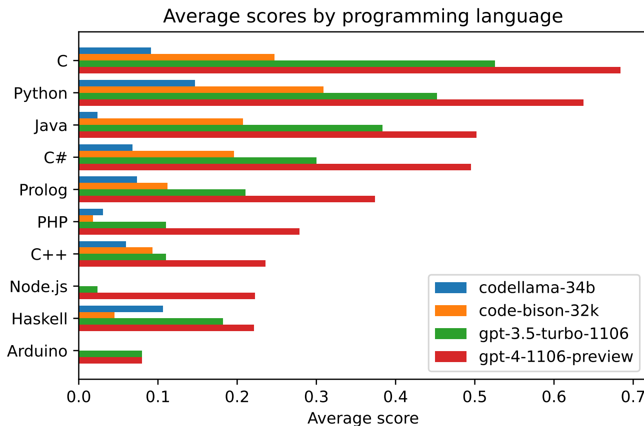

Figure 5: Average scores by programming language

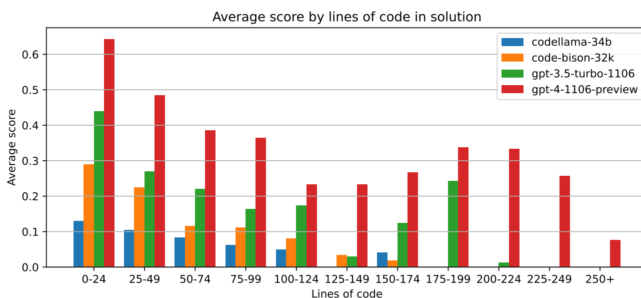

Figure 6: Average scores by lines of code. All solutions from each model are bucketed into groups by number of lines, and the plot shows the average score for each group.

multi-line comments, though we found that models rarely emit these, so this probably does not matter too much.

Figure 6 shows the average score of each model's solutions as a function of the number of lines in a solution. We observe an overall downward trend in scores as the number of lines increases. This trend is especially consistent in solutions with fewer than 150 lines, for which the data is relatively plentiful.

Figure 7 shows the distribution of the number of lines of code of each model's solutions that passed, i.e. earned a score of 1.0. Most successful programs were short: we see that for all models at least 75% of solutions were shorter than 50 lines. Neither Code Llama nor Codey produced any successful solutions of over 100 lines. However, GPT-3.5 generated two successful programs that were more than 150 lines long, and GPT-4 had several successful programs over 200 lines.

Figure 8 compares the number of lines in GPT-4's solutions and our reference solutions for the same exercises. Each blue **x** represents a single exercise where GPT's solution passed with a score of 1.0. We can see a positive correlation between the number of lines in GPT's and our solutions, though in many cases GPT's solution was significantly shorter or longer.

Our reference solutions were written by our teachers, and their nature varies widely. In some exercises, teachers have submitted code golf-style solutions in which an entire problem is solved on a single line. In others, the teachers' solutions have a lot of object-oriented scaffolding that is not strictly necessary for solving the exercise. We chose the smallest of the available reference solutions for each exercise in an attempt to avoid these verbose solutions.

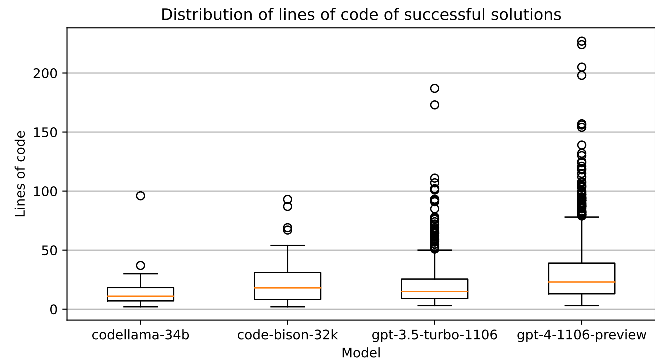

Figure 7: The distribution of the number of lines of code in each model's solutions that earned a perfect score of 1.0. In each box plot, the orange line indicates the median and the box spans the quartiles Q_1 and Q_3. The whiskers extend to points lying 1.5 IQR (= interquartile range) from the box. Outliers are plotted individually.

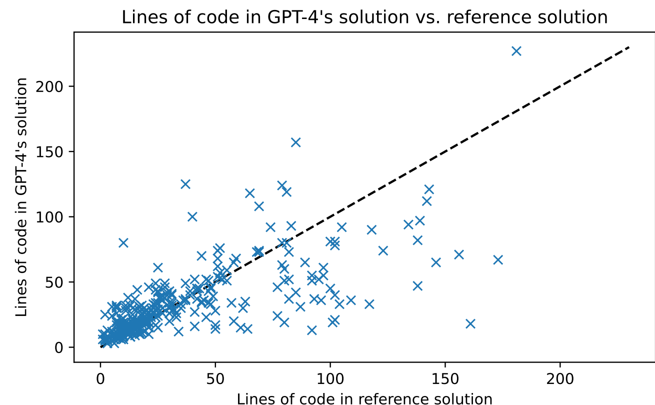

Figure 8: Lines of code in GPT-4's passing solutions plotted versus lines of code in our teachers' reference solutions for the same exercises. The dashed line is the line $x = y$. The plot omits four outlier points.

4.7 Unsuccessful solutions

Although the strongest model GPT-4 produced successful solutions for an impressive variety of problems, in many cases its solutions were unsuccessful (did not earn the maximum score of 1.0). In this section, we take a closer look at GPT-4 solutions that failed.

Figure 9 classifies the outcome of all solutions generated by GPT-4 for all our exercises, grouped by course. The outcome "no program" means that GPT did not produce a program that could be evaluated. Often this was because GPT produced code interspersed with text explaining that it was unable to solve the problem completely. In some cases, GPT gave up and did not output any code at all. The

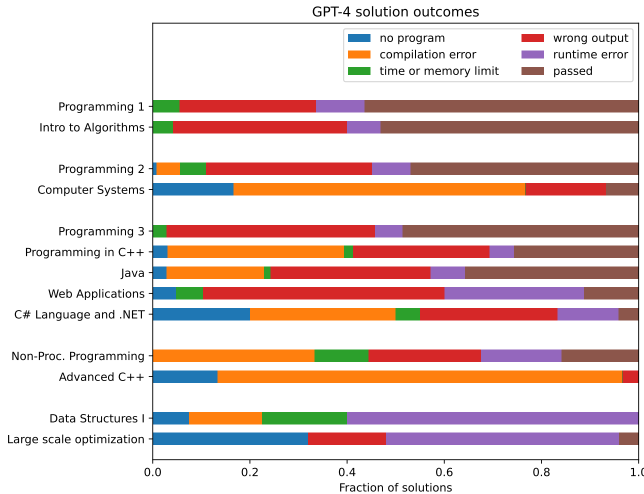

Figure 9: GPT-4 solution outcomes, grouped by course. Courses offered in a single semester are grouped together, with first-semester courses at the top. Any solution that scored 1.0 has the outcome "passed", and other solutions are classified by failure mode. (This chart excludes some courses with only a small number of exercises.)

"compilation error" means that the solution failed to compile. If it compiled, but failed on some tests, the failure is classified as "time or memory limit", "wrong output" or "runtime error", indicating why the majority of test cases failed. If all test cases succeeded, then the solution was successful and is classified as "passed".

We see that the reasons why solutions failed varied a lot from course to course. Our first-semester courses use the dynamically typed language Python, so there could be no compilation errors there. In these courses, the most common failure mode was that the program produced incorrect output. In more advanced courses, GPT often had trouble producing programs that would compile, especially in courses using C++. We also see that in more advanced courses GPT often failed to produce a program. Finally, we see that in two courses (Advanced C++ and Data Structures, a graduate-level course) there were no passing solutions at all.

5 RELATED WORK

One of the most discussed models for code generation is Codex [7, 9, 10, 18], a GPT model fine-tuned on publicly available source code from GitHub. Codex was also used for the well-known developer tool Copilot [10, 20], an AI-powered IntelliSense. Various attempts have been made to create a more elaborate model that focuses specifically on coding, such as Poly-Coder [21], which tried incorporating a variety of programming languages into the training, and AlphaCode [13], which targeted primarily assignments from coding contests. At present, the most powerful model is considered to be GPT-4 [5], which also performed the best in our evaluation.

Most LLMs are evaluated on natural language processing (NLP) problems and there are many datasets in this domain [4]. When focusing on coding assignments, two datasets appear frequently in model evaluations: HumanEval [7] from OpenAI and Google's *Mostly Basic Python Problems* (MBPP) [2]. HumanEval was essential for training Codex [7] and contains 164 original problems in Python. A revised HumanEvalPlus [14] dataset was created later to overcome deficiencies in HumanEval's test cases. MBPP is larger, with about 1000 problems. Amazon introduced the MBXP benchmark [1], a translation of MBPP into many popular programming languages. Similarly, the MultiPL-E [6] tool attempted to translate MBPP and HumanEval into 18 different languages to create a more polyglot dataset. Other datasets focus on specific domains such as OOP [8], coding contests [13], or data science problems [12].

From the perspective of computer science education, there are both concerns and opportunities [3] in using AI tools. Even the early generation of generative models performed quite well on simple programming tasks [2]. Finnie-Ansley et al. focused on evaluating the Codex model, comparing its performance against human students in first (CS1) [9] and second (CS2) [10] introductory computer science courses. Sarsa et al. [18] used Codex to generate exercises and code explanations for CS1 and CS2 courses. A more high-level approach was taken by Cipriano et al. [8], who studied GPT's capabilities to design OOP abstractions. They also outlined guidelines for educators to prevent students from abusing AI for academic assignments. Most of the aforementioned evaluations were done using some level of automation, which raises another question — whether the solutions produced by AI are correct or whether they are just passing (incomplete) tests [14].

Our approach is unique in several ways. Most notably, we present hundreds of exercises from a real curriculum that spans over seven semesters. Furthermore, our selection of languages includes slightly more exotic ones such as Prolog and Haskell.

6 CONCLUSIONS AND FUTURE WORK

Our objective was to evaluate the performance of major LLMs in solving a diverse set of programming assignments designed for students at our university. Let's return to the three research questions we posed in the Introduction.

We first asked: (1) *Can LLMs solve assignments completely in place of a student?* We found that in many cases LLMs can solve our students' assignments completely. In particular, Figure 1 shows that the strongest model GPT-4 can solve almost 40% of the exercises in our dataset with a single query per exercise. Taking the best of 5 queries raises this number to over 55%.

Next: (2) *Do currently available LLMs differ greatly in their performance on this task?* We found that the large general models GPT-3.5 and GPT-4 greatly outperformed the smaller models Codey and Code Llama, which are specialized for code.

Finally: (3) *How complex are the assignments that LLMs can solve?* We will focus our discussion on the strongest model GPT-4, which seems quite capable of solving most of the introductory programming assignments: its average score on our first-semester assignments, taking the best of 5 solutions for each exercise, was 0.86. However by the fourth semester, GPT-4's avg@5 score falls below 0.5, and for our master's-level assignments, it is less than 0.2. This suggests that as a student passes through our bachelor's program, they will progress from a point where GPT can mostly solve their assignments to a point where it will only be a helpful assistant.

The most significant limitation we have seen in the models' ability to solve our assignments is the size of the generated program. Even GPT-4 can rarely produce a complete successful program of more than about 150-200 lines, even though its context window of 128K tokens would theoretically allow producing a program that is many thousands of lines long. Indeed, when faced with many of our advanced exercises which would require a solution of hundreds of lines, GPT-4 insisted on sketching a solution or solving the problem only partially. Generating long programs seems to be especially difficult in languages with complex type systems such as C++ and Haskell. In these languages, GPT-4's solutions often failed to compile in our experiments.

Based on these observations, we believe that a teacher who is concerned about students' use of GPT-like tools may want to place more weight on larger programming projects than on programming exercises such as those in this dataset. Even in the first year of a computer science program, most students should be able to tackle projects that involve writing hundreds of lines of code. GPT will likely be able to assist with writing such projects, but our experiments suggest that it will not be able to write them entirely based on a single prompt. That means that even a student who uses GPT (whether the teacher allows that or not) will need to understand the code that GPT has written at some level of detail to complete the work. That itself may be a valuable learning experience.

We could perform various further experiments with our dataset. In particular, we might be able to improve performance by making a series of LLM queries for each exercise. If the first solution attempt fails, we could provide the LLM with feedback about compiler errors or failing test cases, and then let it try to improve its solution. Furthermore, when an LLM provides only a sketch or a partial solution to an exercise, we could make a series of followup queries to ask it to provide more detail. We could even allow an LLM to debug its own solution by providing a function that it can call to run its program on any input that it likes. The GPT API includes a function callback mechanism that could be useful for this purpose. A combination of these these approaches could allow an LLM to interact with a program under development in much the same way as a human programmer. It would be interesting to find out how much this interactivity could raise scores over our dataset.

ACKNOWLEDGMENTS

This research was supported by Charles University grants SVV-260698 and SVV-260699.

REFERENCES

[1] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).
[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[3] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard - or at least it used to be: educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 500–506.
[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: early experiments with GPT-4. *arXiv preprint arXiv:2303.12712* (2023).
[6] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).
[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[8] Bruno Pereira Cipriano and Pedro Alves. 2023. GPT-3 vs object oriented programming assignments: an experience report. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1.* 61–67.
[9] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: exploring the implications of OpenAI Codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference.* 10–19.
[10] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My AI wants to know if this will be on the exam: testing OpenAI's Codex on CS2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference.* 97–104.
[11] Google. 2023. *Google Cloud launches new AI models.* Retrieved December 8, 2023 from https://cloud.google.com/blog/products/ai-machine-learning/google-cloud-launches-new-ai-models-opens-generative-ai-studio
[12] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning.* PMLR, 18319–18345.
[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.
[14] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).
[15] OpenAI. 2022. *Models - OpenAI API.* Retrieved December 8, 2023 from https://platform.openai.com/docs/models/gpt-3-5
[16] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[17] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[18] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1.* 27–43.
[19] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).
[20] Michel Wermelinger. 2023. Using GitHub Copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 172–178.
[21] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming.* 1–10.