2024

Emergent Capabilities Of Llms For Software Engineering

Conor O'Brien

College of William and Mary - Arts & Sciences, csxobrien@gmail.com

Emergent Capabilities of LLMs for Software Engineering

Conor S. O'Brien

Alexandria, Virginia, United States

Bachelor of Science, College of William & Mary, 2022

A Thesis presented to the Graduate Faculty
of the College of William and Mary in Virginia in
Candidacy for the Degree of Master of Science

Department of Computer Science

The College of William and Mary in Virginia
August 2024

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

_____

Conor S. O'Brien

Approved by the Committee, July 2024

_____

Committee Chair
Professor Denys Poshyvanyk, Computer Science
College of William & Mary

_____

Professor Adwait Nadkarni, Computer Science
College of William & Mary

_____

Professor Oscar Chaparro, Computer Science
College of William & Mary

ABSTRACT

A growing interest for Large Language Models (LLMs) is how increasing their size might result in changes to their behavior not predictable from relatively smaller-scaled models. Analyzing these emergent capabilities is therefore crucial to understanding and developing LLMs. Yet, whether LLMs exhibit emergence, or possess emergent capabilities, is a contested question. Furthermore, most research into LLM emergence has focused on natural language processing tasks and models suited for them.

We focus on investigating emergence in the context of software engineering, and recontextualize the discussion of emergence in the context of prior research. We propose a multifaceted pipeline for evaluating and reasoning about emergent capabilities of LLMs in any context and instantiate this pipeline to analyze the emergent capabilities of the CodeGen1-multi model across four scales ranging from 350M parameters to 16.1B parameters. We examine the model's performance on the software engineering tasks of automatic bug fixing, code translation, and commit message generation. We find no evidence of emergent growth at this scale on these tasks and consequently discuss the future investigation of emergent capabilities.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

In memoriam Raghu "razetime" Ranganathan, a remarkable K programmer, whose life was taken from us too soon.

LIST OF TABLES

LIST OF FIGURES

Emergent Capabilities of LLMs for Software Engineering

Chapter 1

Introduction

1.1 Overview

1.1.1 Motivation

Whether Large Language Models (LLMs) can exhibit capabilities which appear unpredictably upon scaling them large enough is both an alarming and powerful prospect. This phenomenon has the potential to both unlock new planes of improvement on tasks, as well as unprecedented harmful capabilities. Therefore, establishing a reliable methodology of investigating emergent capabilities is crucial to responsible and continued development of LLMs.

While much research has been conducted on LLM's emergent capabilities in the domain of natural language processing (NLP), most of the discourse until now has focused primarily on assessing whether researchers have correctly demonstrated the phenomenon as a property of models, or if it is the case that researchers have merely induced these findings by choice of metric [21], or if emergent capabilities are better explained through its prowess in the more generic ability to learn from examples in its input, or as a description of capabilities which do not strongly require reasoning [11].

Our study seeks to investigate emergent capabilities exclusively in the context of software engineering tasks, a novelty in the discourse. Software engineering tasks and the corresponding capabilities required of LLMs to solve them differ notably from those required to solve natural

language processing tasks. Furthermore, our study analyzes capabilities relevant to the field of software engineering, such as automatic bug fixing and commit message generation, data about which would be relevant to LLMs targeting this domain designed to assist programmers in this field.

1.2   Background

A principal objective of deep learning is to solve problems for which classical programming is inadequate or exorbitantly difficult. Instead of, say, writing code which solves a problem, deep learning employs a model which attempts to solve a target goal; this model can be thought of a function, which ideally transforms the input to the desired output. The model undergoes training, where it attempts to learn general patterns and knowledge from what it is shown, which has the effect of refining this function iteratively. The exact composition of this model varies greatly depending on many factors, such as the task trying to be solved and which data is available to train the model.

The transformer model was first articulated by Vaswani et al. [25]. Transformer-based models use a mechanism called attention, which enables models to efficiently examine and prioritize information from various points in the input.

Since its conception, the Transformer model has become the backbone of many Large Language Models, and the basis for much research in model design. Transformer-based models typically are either *autoregressive* (or unidirectional) [25] or *bidirectional*, as in BERT [4]. In autoregressive Transformers, the model's attention can only project to prior tokens, whereas attention in bidirectional Transformers projects both before and after the current token being processed.

Although state-of-the-art LLMs have proven increasingly capable on a wide variety of tasks [2] [9], especially in the domain of Natural Language Processing, many models and model architectures are trained on and adapted to the realm of software engineering: T5 [16] spawned CodeT5 [26], BERT[4] spawned CodeBERT [5], and LLaMA [24] spawned Code Llama [20].

3

In other words, there is widespread precedent for developing LLMs specific to the domain of software engineering. Likewise, though research on emergence in LLMs hitherto has been focused on NLP tasks, virtually none has been directed predominately towards the field of software engineering.

1.3   Related Work

The notion of emergent capabilities has long been a subject of inquiry in the field of machine learning. As early as 2020 and 2021, researchers were concerned about the negative side effects of increasing model scale, particularly with increasingly larger models adopting the explicit and implicit biases featured in their training sets, and their ability to mimic coherent human speech and articulation [1], based on concerns and research done into the various kinds of bias found in models such as BERT, GPT-2, and GPT-3 [6] [7].

Researchers have used the term *emergence* to refer to capabilities models acquired through training that they were not explicitly trained for, as in Nijkamp et al.'s paper documenting the CodeGen family, where the term is used to describe the model's capability to synthesize programs from comment descriptions [13]. The rigorous study of emergent capabilities, however, was made most prominent with Wei et al., where emergent capabilities are framed not as capabilities acquired without explicit intentions, but as sharp, unpredictable jumps in performance [27]. Their findings suggest that LLMs, through certain tasks and prompting methods, when scaled high enough (e.g. by training compute or parameter count), can unexpectedly and noticeably break plateaued performance.

However, various researchers contest the claim that these models exhibited emergent capabilities whatsoever. Shaeffer et al. [21] suggest the appearance of emergence is better explained by the metrics used to assess model performance, rather than as some property of the model itself; they implicate nonlinear and discontinuous metrics as a confounding factor in Wei et al.'s results. Other research by Lu et al. suggests what appears to be emergent capabilities are better explained as the results of in-context learning, that is, the model's ability to derive

crucial information from the context it is prompted with and apply that knowledge to the task at hand [11].

1.4 Definition of Emergence

Broadly, a model-task-metric-prompt quadruplet exhibits emergence if and only if the model performs poorly on the task at lower scales, well on higher scales, and the improvement in performance does not linearly correspond with the increased scale; in other words, emergent performance is characterized by unexpected and unpredictable jumps in performance. In this section, we provide a high-level definition of emergence, which can be instantiated with particular models, tasks, evluative metrics, and prompting techniques to establish emergence.

More specifically, we define emergence to be a property of (model, task, metric, prompt) quadruplets. A model may perform non-emergently on a variety of tasks, so it is important to distinguish between them when discussing emergence. As per Wei et al., the good performance of different prompting strategies may themselves be emergent capabilities [27]. Last, the inclusion of metrics in this definition is informed by observations that emergence may appear only under certain metrics [21].

If a model exhibits emergent under a variety of different, relevant metrics, especially continuous ones as per Schaeffer et al. [21], we may also say emergence is a property of (model, task, prompt) triplets, and that the model is emergent under that task with that prompting strategy. Likewise, if a model is emergent under a variety of different prompting strategies and metrics, we may say emergence is a property of (model, task) pairs, and that the model is emergent under that task.

As varying model scale is definitionally required to observe emergence, when we talk about a *model* in our definition, we implicitly require a set or family of models which differ only by some measure of scale. Therefore, emergence is also a property of a subset of model scales; in theory, new models in the family could be produced, either of larger or smaller scale outside the observed range, or as intermittent scales between observed model scales. Therefore, by our

methods, a classification of emergence is a label only upon a view of particular, existing scales, and not a classification of all the possible views of the model.

1.5   Research Goal

Our overall research goal is to investigate emergence by varying parameter count on publicly available LLMs as it manifests in particular models on downstream tasks, assessed by metrics, enumerated in Section 2.2.

RQ$_1$ *What effect does varying the scale of CodeGen1-multi models have upon its performance on tasks of code repair, code translation, and commit message generation?*

RQ$_2$ *Between which model scales, if any at all, do emergent capabilities appear?*

To examine these questions, we first establish a general methodology in Chapter 2; this pipeline can be applied to a variety of circumstances where one wishes to ascertain whether an LLM exhibits emergence. We then instantiate this pipeline specifically with the CodeGen1-multi model, our tasks, and our metrics. In furtherance of answering **RQ**$_1$, we provide the results of applying this pipeline in Chapter 3, and in furtherance of answering **RQ**$_2$, we analyze whether our results suggest emergence in Chapter 4. Finally, we summarize our findings on these research questions in Chapter 5.

Chapter 2

Methodology

To establish more precise confidence in verdicts of emergence, we propose a formalized system of evaluating emergence which uses two methods for visualizing whether models display emergence on the observed scale: By fitting the data to linear models to observe emergent growth, and by leveraging Ribeiro et al.'s CheckList method to explore model performance beyond mere accuracy assessments [19]. By assessing the models from multiple angles, we will be able to assert whether the model-task-metric-prompt quadruplet exhibits emergence.

2.1   Models

We analyze how the model family "CodeGen1-multi" with parameter counts 350M, 2.7B, 6.1B, and 16.1B performs on various software engineering tasks (see Section 2.3). CodeGen1-multi is built upon the natural language model CodeGen1-nl by training it further on open-source code snippets in multiple programming languages; CodeGen1-nl is predominately trained on English text [13]. The ability to perform well on software engineering tasks other than program synthesis is crucially important to detecting emergence.

We invoke the models deterministically with a temperature of 0, using the full width of the provided inputs as the context window, and truncating experiments after the generated output exceeds 500 tokens. Our proposed pipeline can naturally measure the effect of temperature on model performance in the same way it showcases variance in bootstrapping.

2.2 Metrics

To evaluate the quality of the outputs the model gives on each of the software engineering-related tasks, we use the following metrics: Exact Match, BLEU (along with specializations B-Moses and B-Norm), and CodeBLEU.

Exact Match (EM) is a discontinuous metric which simply computes the proportion of answers the model gives when completing a task which exactly match the reference answer. EM grades the model's responses with a score from 0 to 1, where higher is better. For example, if the model produces exactly the reference solution in 37 of the 100 given cases, the EM metric is given as $37/100 = 0.37$.

BLEU is a metric designed to simulate human evaluation of machine translated text [15], which has since seen widespread use in NLP and ML [17]. The metric computes the proportion of N-grams (i.e., runs of N consecutive words) that appear in the model's answer against the number of N-grams which appear in the reference answer. We adapt Lu et al.'s [12] implementation of the BLEU metric to support the variant metrics such B-Moses and B-Norm. These metrics are specializations of BLEU that normalize their inputs to be lowercased text before grading. Although BLEU and B-Norm apply Lin et al. 2004 smoothing [10], B-Moses does not. Said smoothing operates by incrementing both the numerator and denominator before calculating the precision of how well a given N-gram matches the reference. We configure BLEU and its metric specializations with N-grams sized up to $N = 4$. The BLEU family of metrics grades the model on a score from 0 to 1, where higher is better.

CodeBLEU is a metric devised by Ren et al. [18] which adapts BLEU to process computer code rather than natural language text by considering the Abstract Syntax Tree (AST) structure of the code of the model's answer and the reference code. We use the implementation `codebleu==0.6.1` hosted on Pypi [3], which in turn is based on the aforementioned paper. Like BLEU, CodeBLEU grades the model on a score from 0 to 1, where higher is better.

In furtherance of $\mathbf{RQ}_1$, we select EM both as a baseline metric measuring the model's ability to produce the expected results exactly, and we select BLEU and CodeBLEU as more nuanced

metrics which might better exhibit growth curves. As for assessing emergent capabilities in furtherance of $\mathbf{RQ}_2$, we consider Schaeffer et al. contention that the emergence Wei et al. observed is better explained as a mirage induced by discontinuous metrics [21] by choosing EM (a discontinuous metric) as a basis for discussing how metric continuity may affect emergence and contrasting it with the continuous metrics BLEU and CodeBLEU. If EM were to exhibit emergence while BLEU and/or CodeBLEU did not, this would corroborate this claim of Schaeffer et al.

2.3   Tasks

Code repair: To evaluate the Code repair task (automatic bug fixing), we evaluate the tested models over CodeXGLUE's [12] Bugs2fix task (aka Code-refinement), using the EM, BLEU, and CodeBLEU metrics. The Bugs2fix task consists of Java methods consolidated onto a single line, with the names of variables, types, strings, etc., consistently replaced by generic names such as `TYPE_1` and `STRING_3`. The task is to provide the corrected version of this code in a similar format. For example, given

```
private TYPE_1 getType ( TYPE_2 VAR_1 ) { TYPE_3 VAR_2 = new TYPE_3 (
    ↪ STRING_1 ) ; return new TYPE_1 ( VAR_2 , VAR_2 ) ; }
```

the bug fixer is expected to produce

```
private TYPE_1 getType ( TYPE_2 VAR_1 ) { TYPE_3 VAR_2 = new TYPE_3 (
    ↪ STRING_1 ) ; return new TYPE_1 ( VAR_2 , VAR_2 , this , VAR_1 ) ; }
```

as output.

   Code translation: To evaluate the Code translation task, we evaluate the tested models over CodeXGLUE's [12] CodeTrans task (aka code-to-code-trans), using the EM, BLEU, and CodeBLEU metrics. The CodeTrans task consists of Java methods consolidated onto a single line. The task is to provide equivalent $C^\sharp$ code in a similar format. For example, given

```
public void serialize(LittleEndianOutput out) {out.writeShort(
    ↪ field_1_vcenter);}
```

the code translator is expected to produce

```
public override void Serialize(ILittleEndianOutput out1){out1.WriteShort(
    ↪ field_1_vcenter);}
```

as output.

Commit message generation: To evaluate the Commit message generation task, we evaluate the tested models over CoDiSum's data [28] using BLEU, as well as a subset of the metrics used by Zhang et al. [29], namely, B-Moses and B-Norm, described in detail in [23]. We also considered testing upon Zhang et al.'s data [29] but elected not to, as the length of the prompted data surpassed the tenable resources allocated for running the model. See the elaboration in Section 3.3.

2.4   Evaluating Emergence

To evaluate emergence and answer RQ_2, we utilize a *preponderance of evidence* model of argumentation, which aims to put forward enough evidence that suggests emergence in a particular case is more likely true than not, without asserting so incontrovertibly. Given the difficulty in meaningfully interpreting LLMs based on their composition, which makes black-box testing the most approachable method for evaluating LLMs, purely analytical solutions for emergence are elusive, so we introduce two objective measures of emergence which can be used to help establish a preponderance of evidence in favor of or against a verdict of emergence in an LLM: Regression modeling the graded results of the LLM on the task, and perturbation via Checklist. In short, failing to match a regression (in our case, a linear regression) suggests emergent growth, and a model performing comparably under the perturbed test cases indicates robust knowledge, which in turn strengthens a suggestion of emergent growth; flourishing under those circumstances may itself be an indicator of further emergent capabilities gained specifically because of the perturbations. Taking these suggestions into consideration may establish sufficient evidence to indicate that it is more likely for the model to exhibit emergent capabilities than not.

We adapt Wei et al.'s technique of scaling curve analysis, which relies on visual inspection of the linearity of logarithmic plots to demonstrate emergence [27], to rely on black-box regression tests on the attained data to test for predictable growth. Although the exact nature of these tests can vary depending on the experimental setup, we choose to evaluate linear regressions on the attained data (model performance graded by various metrics) to assess for non-linear growth. Non-linear growth is a proxy variable unpredictability; as we only have a limited view of the true shape of the performance graph as seen through four different model configurations, we choose to evaluate a linear regression model rather logarithmic or exponential regression models to avoid overfitting the curves to the limited datapoints. This restriction does not exist generally, and using this pipeline in circumstances where more data is available demands considering additional regressions.

Considering the results of these evaluation tactics in tandem suggests emergence on a model-task-metric-prompt quadruplet, and consequently on a model-task pair. Following Schaeffer et al. [21], the most important metrics to consider are continuous metrics, as discontinuous metrics may induce the mirage of emergence.

2.4.1 Linear regression evaluation

First, we employ a statistics-driven approach which is more suitable for the specifics of our experiments to ascertain if the model exhibits emergent growth anywhere in our observed range, as per $\mathbf{RQ}_2$, as we do not have such fine-grained control over model scale as Wei et al had. We instead evaluate the emergence of a model-task-metric-prompt quadruplet by fitting the evaluated results to a linear regression model and then measuring the resultant error.

Specifically, let y be the observed performances of the model across family sizes x. We compute the series of predicted values from a linear model $\hat{y}$ using a linear regression. To assess how well this linear model fits our observations, we employ Root Mean Square Deviation (RMSD) and Mean Absolute Error (MAE), the formulae for which are as follows:

$$\text{RMSD}(x, y) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{2.1}$$

$$\text{MAE}(x, y) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2.2}$$

After computing these metrics, we employ a binary cutoff at some threshold T, below or equal to which model-task-metric-prompt quadruplets are considered to have linear output, and above which they are considered to be potentially emergent. In our case, we employ a cutoff of $T = 0.10$ for both metrics.

2.4.2 Checklist

The Checklist utility designed by Ribeiro et al. offers more robust methods to assess NLP models beyond mere accuracy assessments [19]. Although suited more for NLP tasks, we adapt the Checklist Editor template functionality to generate novel test cases for software engineering tasks based on perturbing existing test cases. By observing how the model performs on the modified tests as compared to the original tests, as well as accounting for the amount the modified test deviates from the original, we attain a better understanding of how exactly the model comprehends the task. These insights, in turn, may strengthen or weaken a verdict of emergence and consequently inform the decision made for $\textbf{RQ}_2$; consider the different cases of how the perturbations affect the model's performance:

i. **Maintains performance.** If a model performs well on the original test cases while performing comparably on the modified test cases, this suggests that the capabilities the model has learned are quite general and robust, which in turn would strengthen a verdict of emergent capabilities existing.

(This also technically covers the case of maintaining poor performance; this would only further weaken a verdict of emergence.)

ii. **Degrades performance.** Contrarily, if a model performs well initially but performance deteriorates drastically on the modified data, it may suggest that, even in the face of apparent emergence, the capabilities the model has acquired are in fact quite limited.

iii. **Improves performance.** Although it may seem unusual, there is also a third case worth mentioning, where the model does not perform well on the original test cases, and performance improves on the modified test cases; this may suggest the modifications themselves might enable the model to do well, either emergently or not.

Furthermore, we may assess the relationship between degree of perturbation (i.e., Levenshtein distance between original and modified test cases) and performance attained for a more granular picture of the model's resiliency to degrees of changes.

Chapter 3

Results

To answer $\mathbf{RQ}_1$ and examine the effect of varying the scale of the CodeGen1-multi model, we run 100 experiments on models with parameter sizes 350M, 2.7B, 6.1B, and 16.1B, for each of the various prompting strategies we explored on each of the tested software engineering tasks: Code repair (Section 3.1), code translation (Section 3.2), and commit message generation (Section 3.3). Each experiment consists of a test case passed as input through a prompt template, which in turn is passed through each of the four model scales of CodeGen1-multi, and the output is obtained as the first full line of output the model produces in response. This output is then compared with the reference answer using one of a few metrics (Exact Match, BLEU or BLEU variants with different parameters, and/or CodeBLEU).

3.1 Bugs2fix

CodeXGLUE's [12] Bugs2fix (aka Code-Code/code-refinement) paired with the four model families in CodeGen1 multilingual (parameter sizes 350M, 2B, 6B, and 16B) serve as the basis for testing our pipeline.

For this prototype, we look at the first 100 test case pairs in small/test.buggy-fixed.buggy and small/test.buggy-fixed.fixed. Although our pipeline is phrased in terms of multiple trials per datapoint, as we are running the cases deterministically, there is no difference between running each test case once and each test case multiple times.

3.1.1 Prompting strategy

We evaluated two different prompts, which we term `prompt0` and `prompt1`. We enumerate our prompts, indicating where line breaks exist in the prompt by also including the literal \n; we use `{buggy_code}` to indicate where the test case, the buggy code that needs to be repaired, is inserted into the prompt, before everything is passed through to the model. Where the line is too long to display all on one line, and must be broken, we indicate that with ↪.

Prompt ID	Prompt Template

prompt0	```// the buggy version of the code\n```
	```{java_code}\n```
	```// the fixed version of the code\n```

prompt1	```// You are given a piece of buggy code. Your task```
	``` ↪ is to fix the error, and generate the```
	``` ↪ corrected code. Fix the following code:\n```
	```{buggy_code}\n```

prompt2	```// You are given a piece of buggy code. Your task```
	``` ↪ is to fix the error, and generate the```
	``` ↪ corrected code. Fix the following code:\n```
	```{buggy_code}\n```
	```// The following code is correct:\n```

We decided on `prompt0` after small-scale experimentation as to how the model responds to various similar prompting techniques. We include `prompt1` as a more traditional prompting technique. We introduce `prompt2` as a combination of both `prompt0` and `prompt1`, inspired by manually examining some of the more intelligible results that `prompt1` emitted, which emitted the exact phrase `// The following code is correct:\n` in a few instances.

3.1.2 Results

To answer RQ_1 and examine the effect of varying the scale of the CodeGen1-multi model, we run 100 experiments on each of the prompts prompt0, prompt1, and prompt2 for the code repair task.
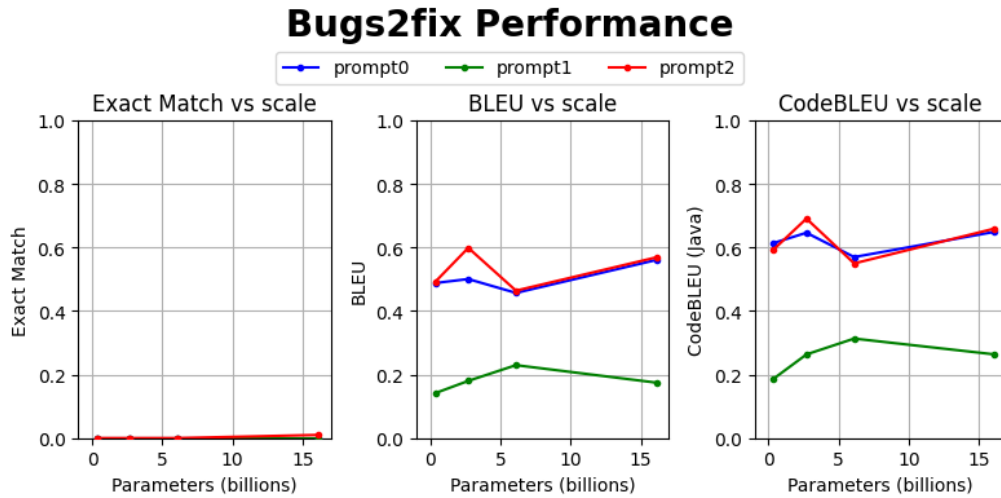
Figure 3.1: CodeGen1-multi performance on first 100 CodeXGLUE Bugs2fix test cases, assessed by the metrics Exact Match, BLEU, and CodeBLEU (operating on Java ASTs), and prompted by prompt0, prompt1, and prompt2 described in Section 3.1.1.

Metric	Prompt	350M	2.7B	6.1B	16.1B	RMSD	MAE
EM	prompt0	0	0	0	0	0	0
	prompt1	0	0	0	0	0	0
	prompt2	0	0	0	**0.0100**	0.0015	0.0012
BLEU	prompt0	0.4882	0.5006	0.4571	0.5607	0.0252	0.0218
	prompt1	0.1416	0.1801	0.2297	0.1747	0.0307	0.0256
	prompt2	0.4920	**0.5989**	0.4642	0.5689	0.0526	0.0449
CodeBLEU (Java)	prompt0	0.6139	0.6464	0.5699	0.6492	0.0304	0.0248
	prompt1	0.1867	0.2634	0.3133	0.2636	0.0412	0.0375
	prompt2	0.5934	**0.6909**	0.5499	0.6595	0.0537	0.0453

Table 3.1: Results for pairs on CodeGen1 multi. Bolded results indicate maximum attained performance for best prompt. CodeBLEU operated on Java ASTs.

3.1.3   Checklist

The format of the Bugs2fix dataset employs dummy sequential variable names such as `TYPE_1`, `TYPE_2`, `METHOD_1`, etc. We mutate the same 100 test cases used for the main task via a modified version of the Checklist perturb utility [19]. Please see Appendix A.1.1.1 for an elaboration of how this works.

3.1.3.1   Results

To answer $\mathbf{RQ}_1$ and examine the effect of varying the scale of the CodeGen1-multi model, we run 100 experiments on each of the prompts `prompt0`, `prompt1`, and `prompt2` for the code repair task, varying the corresponding experiments from the baseline code repair task using Checklist perturbations.

Bugs2fix (Checklist) Performance
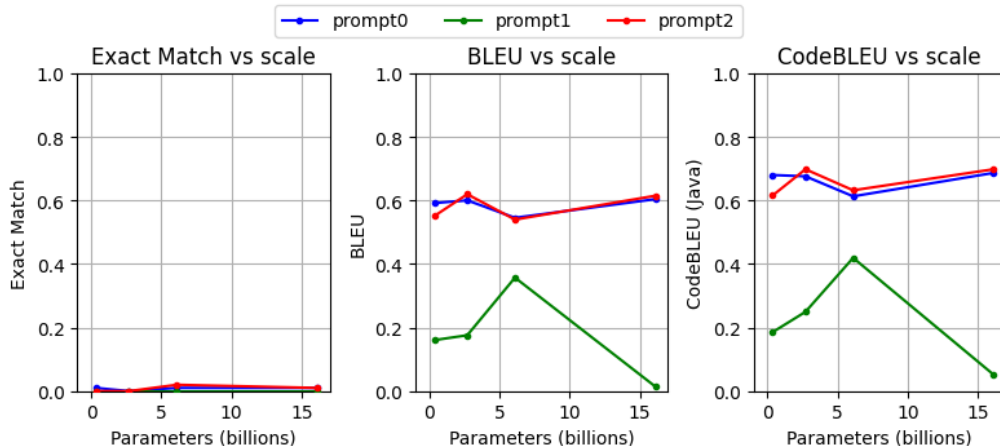
Figure 3.2: CodeGen1-multi performance on first 100 CodeXGLUE Bugs2fix test cases (perturbed according to the transformations described in Section 3.1.3), assessed by the metrics Exact Match, BLEU, and CodeBLEU (operating on Java ASTs), and prompted by prompt0, prompt1, and prompt2 described in Section 3.1.1.

Metric	Prompt	350M	2.7B	6.1B	16.1B	RMSD	MAE
EM	prompt0	0.0100	0	0.0100	0.0100	0.0041	0.0033
	prompt1	0	0	0	0	0	0
	prompt2	0	0	**0.0200**	0.0100	0.0073	0.0063
BLEU	prompt0	0.5918	0.6004	0.5456	0.6042	0.0231	0.0199
	prompt1	0.1605	0.1759	0.3572	0.0145	**0.1037**	0.0890
	prompt2	0.5509	**0.6197**	0.5399	0.6149	0.0326	0.0278
CodeBLEU (Java)	prompt0	0.6802	0.6761	0.6131	0.6867	0.0294	0.0254
	prompt1	0.1853	0.2495	0.4194	0.0521	**0.1145**	0.0952
	prompt2	0.6154	**0.6983**	0.6321	0.6983	0.0312	0.0264

Table 3.2: Results for pairs on CodeGen1 multi. Bolded results indicate maximum attained performance for best prompt, and bolded linear statistics suggests nonlinear data. CodeBLEU operated on Java ASTs.
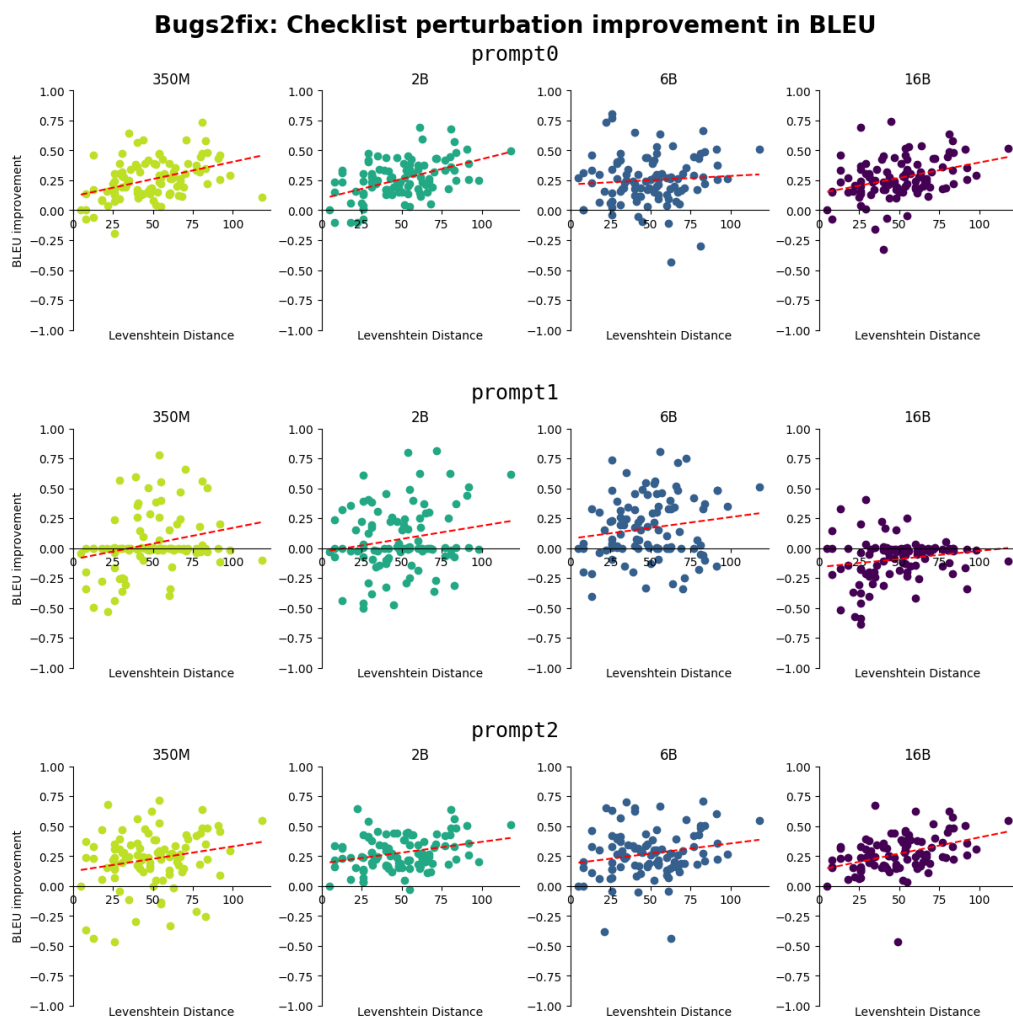
Figure 3.3: Scatterplot of the relationship between the Levenshtein distance between the original, unmodified test case and the corresponding perturbed test case on the x-axis, and the relative performance increase from the BLEU score of the unmodified test case to the BLEU score of the perturbed test case; positive y-values indicate improvement.

Bugs2fix: Checklist perturbation improvement in CodeBLEU
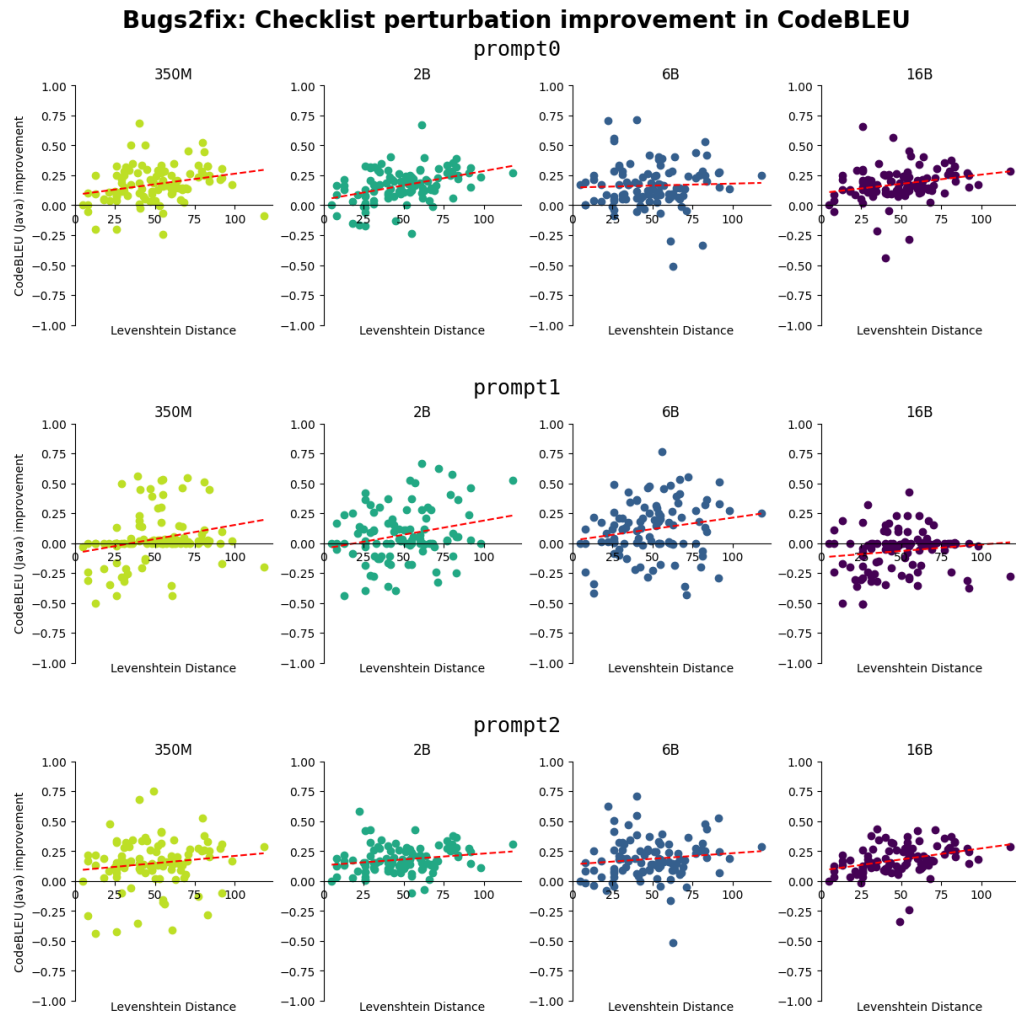
Figure 3.4: Scatterplot of the relationship between the Levenshtein distance between the original, unmodified test case and the corresponding perturbed test case on the x-axis, and the relative performance increase from the CodeBLEU score of the unmodified test case to the CodeBLEU score of the perturbed test case; positive y-values indicate improvement.

Metric	Prompt	Scale	b_1	b_0	s	r^2
BLEU	prompt0	350M	0.00289	0.11581	0.00066	0.40513
		2B	0.00332	0.09547	0.00062	0.47775
		6B	0.00071	0.21571	0.00090	0.07892
		16B	0.00258	0.14091	0.00070	0.34689
BLEU	prompt1	350M	0.00264	-0.09425	0.00103	0.25031
		2B	0.00219	-0.03233	0.00116	0.18676
		6B	0.00180	0.07927	0.00114	0.15795
		16B	0.00135	-0.15816	0.00077	0.17407
BLEU	prompt2	350M	0.00207	0.12434	0.00098	0.20815
		2B	0.00180	0.18934	0.00059	0.29702
		6B	0.00169	0.18680	0.00086	0.19569
		16B	0.00272	0.13498	0.00063	0.39901
CodeBLEU	prompt0	350M	0.00182	0.08116	0.00064	0.27576
		2B	0.00245	0.04318	0.00058	0.39084
		6B	0.00033	0.14633	0.00082	0.04129
		16B	0.00155	0.10037	0.00061	0.24874
CodeBLEU	prompt1	350M	0.00237	-0.08471	0.00094	0.24636
		2B	0.00238	-0.04852	0.00096	0.24202
		6B	0.00192	0.02296	0.00099	0.19237
		16B	0.00106	-0.11905	0.00078	0.13647
CodeBLEU	prompt2	350M	0.00124	0.08723	0.00087	0.14225
		2B	0.00100	0.13036	0.00047	0.21161
		6B	0.00095	0.13864	0.00074	0.12805
		16B	0.00185	0.08718	0.00050	0.34904

Table 3.3: Summary results for linear regressions performed on perturbed results. The x-axis is the Levenshtein distance between the original prompt and the perturbed prompt. The y-axis is the amount the perturbed prompt improved the model's performance relative to the original prompt. b_1 is the regression's slope, b_0 is its intercept, s is the standard error, and r^2 is coefficient of determination.

Across the prompts, we can see from the positive b_1 slope values that there is a positive but tenuous correlation between Levenshtein distance and performance improvement across all metrics; furthermore, we can also see that in many cases, especially in `prompt0` and `prompt2`, the perturbations had a uniquely positive impact on most of the tests.

Worth noting is the marked decrease in performance for `prompt1` at the 16.1B parameter scale: The decrease is so drastic, that it is classified as sufficiently non-linear by our models, and could suggest *negative* emergence if read in a vacuum. However, manually inspecting the outputs under these configurations, we hypothesize this is a reflection of the limitations of our testing harness which secures the output from the model. See Section 4.2.3 for an extended discussion.

3.2   Code translation

CodeXGLUE's [12] CodeTrans (aka Code-Code/code-to-code-trans) paired with the four model families in CodeGen1 multilingual (parameter sizes 350M, 2B, 6B, and 16B) serve as the basis for testing our pipeline.

For this prototype, we look at the first 100 test case pairs in test.java-cs.txt.java and test.java-cs.txt.cs. Although our pipeline is phrased in terms of multiple trials per datapoint, as we are running the cases deterministically, there is no difference between running each test case once and each test case multiple times.

3.2.1   Prompting strategy

We evaluated three different prompts, which we term `prompt0`, `prompt1`, and `prompt2`. We enumerate our prompts, indicating where line breaks exist in the prompt by also including the literal \n; we use {java_code} to indicate where the test case, the Java code that needs to be translated into C$^\sharp$, is inserted into the prompt, before everything is passed through to the model. Where the line is too long to display all on one line, and must be broken, we indicate that with ↪.

Prompt ID	Prompt Template

prompt0	```
	// original code.java\n
	{java_code}\n
	// code.cs version of code.java\n
	``

prompt1	```
	// code.java\n
	{java_code}\n
	// code.cs\n
	``

prompt2	```
	// This code is written in Java. Reproduce the
	    ↪ same exact code in C#.\n
	{java_code}\n
	``

prompt3	```
	// original code.java\n
	{java_code}\n
	\n
	// code.cs version of code.java\n
	``

prompt4	```
	// This code is written in Java. Reproduce the
	    ↪ same exact code in C#.\n
	{java_code}\n
	// This code is written in C#.\n
	``

3.2.2 Results

To answer $\mathbf{RQ}_1$ and examine the effect of varying the scale of the CodeGen1-multi model, we run 100 experiments on each of the prompts prompt0, prompt1, prompt2, prompt3, and prompt4 for the code translation task.
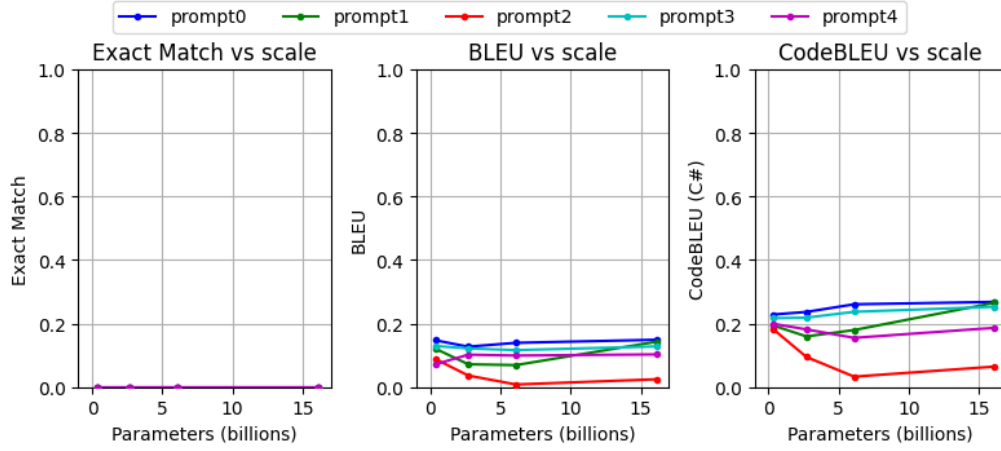
Figure 3.5: CodeGen1-multi performance on first 100 CodeXGLUE Code2code test cases, assessed by the metrics Exact Match, BLEU, and CodeBLEU (operating on C♯ ASTs), and prompted by prompt0, prompt1, prompt2, prompt3, and prompt4 described in Section 3.2.1.

Metric	Prompt	350M	2.7B	6.1B	16.1B	RMSD	MAE
EM	prompt0	0	0	0	0	0	0
	prompt1	0	0	0	0	0	0
	prompt2	0	0	0	0	0	0
	prompt3	0	0	0	0	0	0
	prompt4	0	0	0	0	0	0
BLEU	prompt0	0.1474	0.1273	0.1394	**0.1491**	0.0078	0.0063
	prompt1	0.1212	0.0719	0.0693	0.1429	0.0269	0.0254
	prompt2	0.0886	0.0364	0.0083	0.0247	0.0245	0.0227
	prompt3	0.1295	0.1218	0.1164	0.1281	0.0052	0.0046
	prompt4	0.0717	0.1021	0.0994	0.1029	0.0103	0.0092
CodeBLEU (C$^\sharp$)	prompt0	0.2286	0.2366	0.2607	**0.2682**	0.0074	0.0064
	prompt1	0.1936	0.1596	0.1796	0.2655	0.0201	0.0190
	prompt2	0.1814	0.0948	0.0330	0.0646	0.0442	0.0401
	prompt3	0.2169	0.2187	0.2370	0.2529	0.0038	0.0031
	prompt4	0.1992	0.1816	0.1551	0.1866	0.0159	0.0129

Table 3.4: Results for pairs on CodeGen1 multi. Bolded results indicate maximum attained performance for best prompt. CodeBLEU operated on C$^\sharp$ ASTs.

3.3 Commit message generation

Whereas with other model-task pairs where we tested multiple prompts per task, we elected not to move forward with prompting techniques beyond our preliminary explorations of prompt0. Fitting the entire diff in working memory for the model proved infeasible for many prompts, exhausting the GPU resources we had available to test the models. Furthermore, the datasets we examined (CoDiSum [28] and those by Zhang et al. [29]) are given as outputs of the diff command and the corresponding expected natural language commit message. Although the CodeGen-multi and CodeGen-mono models were built atop the corresponding CodeGen-nl

natural language models, we theorize the combination of the unfamiliar syntax of `diff` with the less-comfortable natural language output explains the model's abysmal performance on this task. Using Large Language Models to generate helpful summaries of commit messages is an area of growing research [8], which motivated our initial pursuits, but to pursue this subdomain of research with this model, datasets phrased as providing separate code snippets representing the state of the commit before and after the change might be a more fruitful approach.

3.3.1   Prompting strategy

We evaluated one prompt, `prompt0`. We enumerate our prompt, indicating where line breaks exist in the prompt by also including the literal \n; we use {diff_output} to indicate where the test case, the raw output of running the pertinent `diff` command, is inserted into the prompt before being passed through to the model.

Prompt ID	Prompt Template

prompt0	```/* diff of changes\n```
	```{diff_output}\n```
	```*/\n```
	```// a summary of the above diff is:\n```
	```// -```

3.3.2   Results

To answer RQ_1 and examine the effect of varying the scale of the CodeGen1-multi model, we run 100 experiments on `prompt0` for the commit message generation task.

26
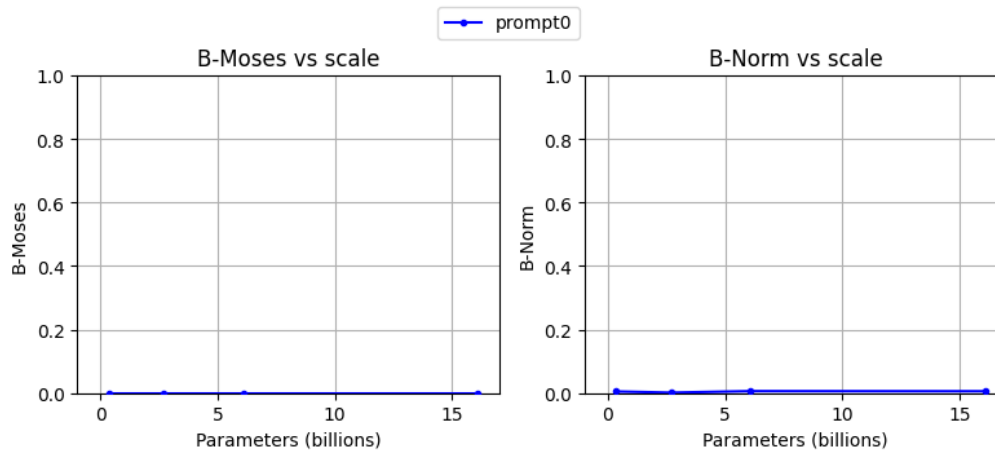
Figure 3.6: CodeGen1-multi performance on the first 100 test cases from the CoDiSum dataset, assessed by the metrics B-Moses and B-Norm, and prompted by `prompt0` described in Section 3.3.1.

Metric	Prompt	350M	2.7B	6.1B	16.1B	RMSD	MAE
B-Moses	prompt0	0	0	0	0	0	0
B-Norm	prompt0	0.0051	0.0014	**0.0063**	0.0059	0.0017	0.0015

Table 3.5: Results for pairs on CodeGen1 multi. Bolded results indicate maximum attained performance for best prompt.

Chapter 4

Analysis

4.1 Bootstrapping

To analyze our results, we employ bootstrapping to resample our data. This gives us insights as to the underlying structure of our data beyond what is visible by merely graphing the aggregate statistics. Bootstrapping works by taking a random subsample of S data points from the primary experimental sample, and performing the same statistic on that subsample. This process is repeated N times. In our case, we apply bootstrapping to each parameter size, forming a line graph of box plots showcasing the distribution of the bootstrapped resampling. For all of our bootstrapping, we take subsample sizes $S = 50$ a total of $N = 500$ iterations.

4.1.1 Bugs2fix
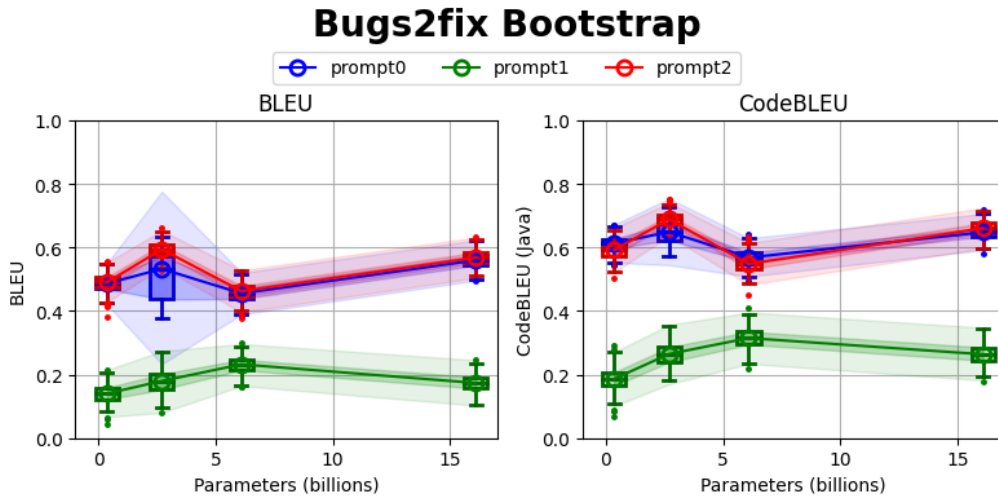
Figure 4.1: Bootstrapping with $S = 50$ and $N = 500$ over the CodeXGLUE Bugs2fix task, assessed by metrics BLEU and CodeBLEU, over the results of prompting CodeGen1-multi with prompts `prompt0`, `prompt1`, and `prompt2` as described in Section 3.1.1. (The Exact Match metric is omitted, as it is 0 across all datapoints.)

4.1.2 Bugs2fix (Checklist)
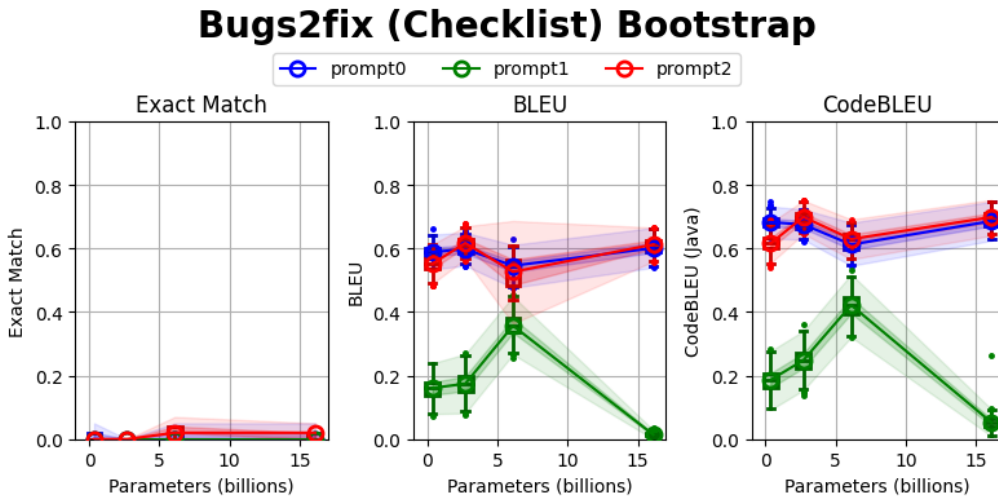
Figure 4.2: Bootstrapping with $S = 50$ and $N = 500$ over the CodeXGLUE Bugs2fix Checklist perturbed task, assessed by metrics Exact Match, BLEU, and CodeBLEU, over the results of prompting CodeGen1-multi with prompts `prompt0`, `prompt1`, and `prompt2` as described in Section 3.1.1.

4.1.3 Code2code
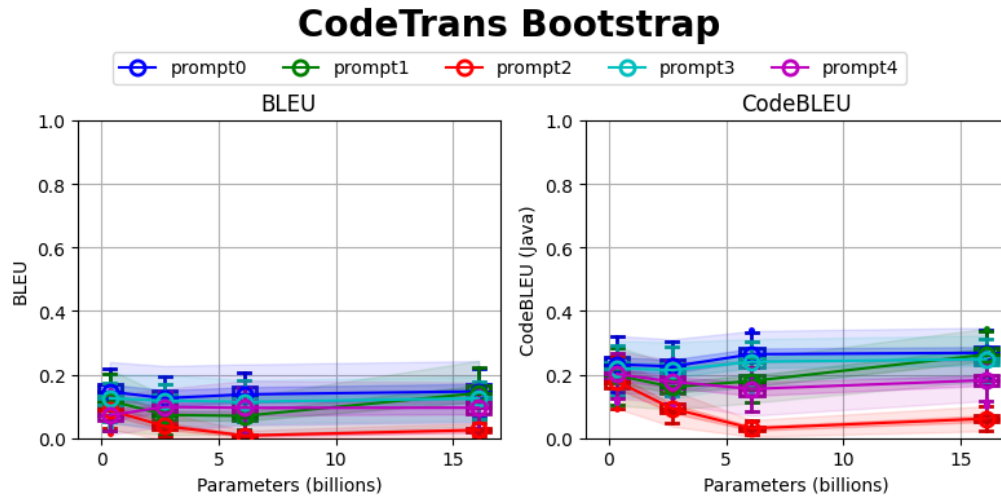
Figure 4.3: Bootstrapping with $S = 50$ and $N = 500$ over the CodeXGLUE CodeTrans task, assessed by metrics Exact Match, BLEU, and CodeBLEU, over the results of prompting CodeGen1-multi with prompts prompt0, prompt1, prompt2, prompt3, and prompt4 as described in Section 3.2.1.

4.2 Discussion

4.2.1 Insights about CodeGen1-multi

Throughout the various experiments and prompting methods, we extracted insights relevant to future prompt engineering and testing of this model. We established experimentally that many conventional prompting techniques and wisdoms that work well with modern LLMs such as ChatGPT and LLaMA do not translate to CodeGen1-multi, such as natural language descriptions following a structure which states the goal, describes what the model should and should not do, and summarizing succinctly what the model should do; in fact, this approach appears unilaterally deleterious towards extracting meaningful output.

We found that methods which appeared more like code enabled the model to not only respond more coherently, but also perform better across our given metrics. Generally, the most successful of our prompts phrase the task as a code comment, followed by the code, followed by

an additional code comment reiterating what the goal is. Furthermore, prompts can be refined by inspecting prompt output for such code comments the model volunteers naturally.

4.2.2 Limitations of Metrics

There is a broad issue worth mentioning, that the metrics BLEU (including B-Moses and B-Norm) and CodeBLEU are more concerned with the model's outputs being *apparently similar* to reference solutions, instead of them being *correct* or *useful*. Of course, to assess the model's output in this way would likely require extensive and involved human surveying; even supposing willing and able human graders, the issues posed by inferring a rating with sufficient granularity so as to extract meaningful verdicts of emergence is pressing and significant.

Perhaps, even though Shaeffer et al. claim emergent capabilities disappear when using continuous metrics [21], those metrics lose sight of our actual goals of model correctness and usefulness. Metrics are not designed to assess this, but to assess how close the produced output matches, token-wise, the expected output, not allowing for significant deviation, even if, for all relevant purposes, it might be equally acceptable. A general solution to this problem remains elusive and/or taxing, but it could perhaps be most easily addressed by examining the question of emergent capabilities with respect to the software engineering domain of code generation against test cases (or the inverse: test cases generation against existing code). In these scenarios, models are afforded the option for a diversity of answers, and grade the extent of how functional the given answer is via code execution, which is exactly the metric that corresponds to the desired properties of correctness and usefulness.

4.2.3 Limitations of the Testing Harness

As the CodeGen family of models does not respond well to conventional prompting phrased as natural language task descriptions, and was prone to ramble indecisively far beyond a concise answer, we made the decision to mirror the strict input format of the test cases for how we expected the model to respond. Since the input test cases consisted of Java methods condensed to a single line, we iteratively prompted the model in chunks until it produced a non-empty line,

or signaled end-of-text. Thus, we ended up considering only a single line as output from the model for each test case. See Appendix A.2 for an elaboration on the specifics of our approach.

The model would likely perform differently if prompted with more conventionally formatted code input (which spans multiple lines), and if we more leniently selected output from the model as well-balanced Java code (again allowed to span multiple lines). Following manual inspection of the model's answers, we found that many of its outputs consisted of a single comment line, or the beginnings of a code snippet with clear intentions to complete the code in a format with more typical whitespace (e.g., an open curly brace { followed by a line break). Thus, not only does our testing examine the model's ability to, say, automatically fix bugs, but to do so while complying with the simple, restricted input/output format.

This phenomenon was especially pronounced in our study of the Checklist perturbations of the CodeXGLUE Bugs2fix task in Section 3.1.3.1; the model's performance at the highest scale of 16.1B parameters under `prompt1` drops drastically, so much so that the overall shape of the graph is considered non-linear under our metrics for evaluating the linearity of emergent jumps. As it turns out, the majority of the 16.1B outputs are some variation of `// The buggy code is:` or sometimes giving a plain English attempt at describing what the bug itself is (e.g., `// The bug is that the method returns a Boolean, but it should return a Cardigan.`); these score very poorly compared to the smaller scales, which are able to comply to the stricter output format. Had we implemented a more lenient testing harness which would ignore comments like this, the model may have performed better at this scale for that prompt.

Chapter 5

Conclusion

We return to the research questions defined at the outset of our investigation.

RQ$_1$ *What effect does varying the scale of CodeGen1-multi models have upon its performance on tasks of code repair, code translation, and commit message generation?*

We find that varying the model scale of the CodeGen1-multi model from 350M parameters to 16.1B parameters generally tends to improve performance slightly in a linear fashion, albeit somewhat loosely; performance was not usually strictly increasing, nor was maximal performance always attained with maximal tested model scale.

Even though varying model scale loosely improves model performance, we find that prompting technique is a stronger determiner of model performance, especially as, with some prompts, increasing model scale negatively affects performance.

RQ$_2$ *Between which model scales, if any at all, do emergent capabilities appear?*

Emergent capabilities did not appear at any model scales on any of our tasks with any of our prompting methods.

This may be due to the fact Wei et al. did not encounter emergence on some tasks until upwards of 10B to 1T parameters ([27], Appendix D). This may also be explained as not finding the right prompts which would enable our models to experience emergent capabilities. Alternatively, it might well be the case that the CodeGen1-multi model is simply relatively stable and predictable at these relatively small scales.

5.1 The Future of Emergent Capabilities

Reasoning about emergent capabilities is a difficult task, fraught with doubts about the existence of the phenomenon itself, as well as uncertainties as to discovering the right prompts and the proper use of metrics. We hope that future research can utilize our pipeline for evaluating emergence to aid future investigations into the question of emergent capabilities.

Whether models can exhibit emergent capabilities is a double-edged sword. The prospect of untapped potential awaiting model designers, if only they provide the requisite increased scale, is both tantalizing and trepidating, depending on the exact nature of the emergent capabilities gained. The model may gain, say, superlative reasoning abilities emergently; it may also gain superlative discriminatory abilities emergently. Conversely, establishing a verdict of no emergent capabilities across a certain range of model scales would seem to bound the model's ability to perform, but also establish a certain degree of confidence in its stability.

Examining the question of emergence is an important step into assessing the interpretability, stability, and predictability of a model. While most models are not released to be available by discrete model scales, those training models can assess the intermediate models scaled by training FLOPs for signs of emergence using our pipeline.

34

Appendix A

Experimental Setup

A.1   Obtaining test cases

Generally, where we select a subset of test cases from a given reference source, we do so by extracting the first N test cases, as opposed to randomly sampling them. We observed that the distribution of unique features (such as variable names) was apparently randomly distributed throughout the documents, and inferred that there is no special ordering that would invalidate the approach of taking the first N lines.

A.1.1   Code refinement (aka Bugs2fix)

To form the basis for our automatic code repair experiment, we selected the first 100 lines from `Code-Code/code-refinement/data/small/test.buggy-fixed.buggy` as the input code, and the corresponding first 100 lines from `Code-Code/code-refinement/data/small/test` `.buggy-fixed.fixed` as reference solutions. Each line in each document represents a Java code snippet, formatted without newlines or excess spaces, and each token is separated by spaces with few exceptions. For example, `VAR_1 . METHOD_2 ( )` is presented with spaces between each of these tokens, but certain language primitives, such as `java.lang.String`, are presented without spaces.

A.1.1.1   Checklist

We apply the following bijective mapping on our base Bugs2fix test cases:

- `VAR_k` $\mapsto$ a or b or ... or z, excluding m and n given `VAR_k` ranged only up to `VAR_12`, making more than 24 variables superfluous.

- `TYPE_k` $\mapsto$ `Apple` or `Box` or `Cardigan` or `Doohickey` or `Egg` or `Gadget` or `Nicknack` or `Thingy` or `Widget` or `Yak`.

- `INT_k` $\mapsto sg$, where s is chosen uniformly from the multiset $\{1, 1, -1\}$ and $g \sim G_X(p = 0.1)$ is sampled from a Geometric distribution with success probability $p = 0.1$; that is, biased towards small, positive numbers, but potentially negative and potentially large.

- `FLOAT_k` $\mapsto \mathrm{round}\left(\mathrm{clip}\left(g, -1000, 1000\right), 4\right)$ where $g \sim \mathcal{N}(\mu = 0, \sigma = 150)$ is sampled from a Normal distribution with mean $\mu = 0$ and variance $\sigma = 150$; that is, biased towards numbers around zero, with 4 decimal places, and $|g| \leq 1000$.

- `CHAR_1` $\mapsto$ `'a'` or `'@'`; `CHAR_2` $\mapsto$ `'b'` or `'&'`; `CHAR_3` $\mapsto$ `'c'` or `'L'`; `CHAR_4` $\mapsto$ `'d'` or `'-'`.

- `STRING_1` $\mapsto$ `""` or `"truth"`; `STRING_2` $\mapsto$ `"results"` or `"\n"`; `STRING_3` $\mapsto$ `"Input: "` or `"orange"`; `STRING_4` $\mapsto$ `"abcdefghijklmnopqrstuvwxyz!.{}"` or `"00000000"`; `STRING_5` $\mapsto$ `"unchecked"` or `"Courier New"`; `STRING_6` $\mapsto$ `" "` or `"EOF"`; `STRING_7` $\mapsto$ `"Hello, World!"` or `"\t"`; `STRING_8` $\mapsto$ `"\r\n"` or `"_____"`; `STRING_9` $\mapsto$ `"asdjiopfapsdfjpoiajdfpoais"` or `"9999888666441"`.

Using this mapping, we project the dummy placeholders to random entries in the mapping. For example,

```
private TYPE_1 getType ( TYPE_2 VAR_1 ) { TYPE_3 VAR_2 = new TYPE_3 (
    ↪ STRING_1 ) ; return new TYPE_1 ( VAR_2 , VAR_2 ) ; }
```

became

```
private Cardigan getType ( Apple h ) { Yak k = new Yak ( "" ) ; return new
    ↪ Cardigan ( k , k ) ; }
```

abiding by the mapping

- `TYPE_1` $\mapsto$ `Cardigan`

- `TYPE_2` $\mapsto$ `Apple`

- `TYPE_3` $\mapsto$ `Yak`

- `VAR_1` $\mapsto$ `h`

- `VAR_2` $\mapsto$ `k`

In practice, although we defined transformations for all placeholders that appeared in the CodeXGLUE `small/test.buggy-fixed.buggy` dataset, we only encountered up to `VAR_6`, `TYPE_4`, `INT_2`, `FLOAT_1`, and `STRING_3` within the first 100 cases, and never encountered any instances of `CHAR_k`.

A.1.2   Code translation (aka CodeTrans)

To form the basis for our code translation experiment, we selected the first 100 lines from `Code-Code/code-to-code-trans/data/test.java-cs.txt.java` as the input code, and the corresponding first 100 lines from `Code-Code/code-to-code-trans/data/test.java-cs.txt.cs` as reference solutions. Each line in the `java-cs.txt.java` document represents a Java code snippet, whereas each line the in the `java-cs.txt.cs` document represents a $C^\sharp$ code snippet. Unlike the test cases for the code refinement task, tokens are not provided as tokens separated by whitespace, but rather with whitespace inserted somewhat arbitrarily.

For example, consider the Java snippet and its corresponding $C^\sharp$ snippet (comments ours):

```
// Java
public ObjectId getObjectId() {return objectId;}

// C#
public virtual ObjectId GetObjectId(){return objectId;}
```

A.1.3 Commit message generation

To form the basis for our commit message generation experiment, we selected the first 100 test cases from the CoDiSum dataset [28]. The input for each test case is the Linux command `diff` with parameters of the file paths of the changed files along with its corresponding output when invoked. The reference for each test case is the given English commit message associated with the file changes.

A.2 Querying the model

To interact with the CodeGen1-multi model, we use custom Python code and IPython Jupyter Notebooks running atop of PyTorch, transformers, Pandas, SciPy, checklist, and numpy libraries.

The main feature of our querying approach is the `generate_until` algorithm, which operates according to the following psuedocode:

```
Parameters inputs, stops, chunk_size, max_size.
current_size <- 0
current_inputs <- inputs
while(current_size < max_size) {
    output_tokens <- model.generate(
        current_inputs,
        max_new_tokens=chunk_size
    )
    // trim the input prompt from the output
    output_tokens_so_far <- slice output_tokens after token_count(inputs)
    output_string <- tokenizer.decode(output_tokens_so_far)

    foreach(stop in stops) {
        if(output_string contains stop) {
            truncate output_string to first index of stop
            return output_string
        }
    }
```

```
    }

    if(EOS_TOKEN in output_string) {
        return output_string
    }

    current_inputs <- concat(current_inputs, output_tokens_so_fars)
}
return output_string
```

In words, it requests tokens from the model in a pre-determined chunk size (50 tokens at a time in our experiments), and continues to do so until one of the following conditions is met:

1. The model generates an end of string token;

2. The model generates one of the requested stop characters (line breaks in our experiments); or

3. The model generates a total number of tokens exceeding the requested maximum token length (500 tokens in our experiments).

We prefer this algorithm to the default free-text generation method provided by the transformers library as it adheres to the input-output format for our test cases and it bypasses the tendency of the CodeGen1-multi model to ramble continuously; if we were to simply generate until the end of string token was encountered with no other stops, we would overload the GPU memory with too wide an input string, as the model is prone to repeating itself with and without variations.

Appendix B

Unused Methodology

B.1 Multiple-Choice Answer Extraction

While our final research product consists of grading and extracting the free text responses of LLMs, we developed and assessed a variety of methods to extract a multiple-choice answer from a model by observing the underlying logits, suited towards such datasets as found in BIG-Bench [22]. We include these methods here for discussion of future work. Of the three approaches we discuss (First branch evaluation, Aggregate normalized logit evaluation, and event-based evaluation), we recommend researchers investigating extracting multiple-choice answers from LLMs via logits investigate both normalized logit evaluation (Section B.1.1.2) and event-based evaluation (Section B.1.1.3). We hypothesize they may perform similarly (in which case normalized logit evaluation should prove more computationally efficient), but as we did not end up using any multiple-choice datasets to investigate emergence, we have not rigorously investigated either.

B.1.1 Evaluating Model Choices

Evaluating which choices a model is most likely to make is crucial to assessing how it thinks and performs. While it is trivial to assess a model's preference for a choice between many tokens by simply choosing the token with maximal logit likelihood, there is some latitude for

assessing the model's preference between multi-token choices. We investigated three different approaches to this problem, and selected 2 of them.

B.1.1.1 First branch evaluation

Intuitively expanding the trivial case of selecting the choice corresponding to the token with maximal logit likelihood, we may iteratively apply this determination to corresponding tokens in the choices in lockstep. That is, we can determine between K choices $\{A_0, A_1, \cdots, A_{K-1}\}$ by finding the minimal token index i where $\max_{0 \le k < K} \text{logit}(A_{k,i})$ is unique, and yielding the choice index k corresponding to that maximal determination.

For example, given two choices of tokenized strings "prints| values| from| 1| to| 10" and "prints| values| from| 0| to| 22" (choice examples from [22]), the model finds no unique maximum until token index i=3 (as the options are identical up to that point), and would make a final determination between the two choices by determining which of the tokens " 1" or " 0" had higher logit likelihood.

This approach is flawed, however, as it will almost certainly never consider the entirety of each option. Suppose the model prefers the choice "prints| values| from| 1| to| 10" based on the " 1" token; the model would choose that same choice regardless of what comes after it. Replacing the model's preferred choice with either of "prints| values| from| 1" and "prints| values| from| 1| to| MAX|_|INT| and| explodes" would not change the model's determination under the evaluation method as one might hope.

B.1.1.2 Aggregate normalized logit evaluation

To remedy the issues with the first branch evaluation approach above, it is clear the model must consider each token in each choice to make an informed determination. Palacio et al. [14] utilize an aggregate method of normalizing a model's logits to match predicted tokens with corresponding Abstract Syntax Tree nodes. We can leverage this approach to rank multiple-choice answers.

This method works simply by summing the logit likelihoods corresponding to each token in each choice, thereby considering each token, and dividing by the number of tokens in each choice, making sure the model does not unduly favor longer choices. Then, the choice with maximal average evaluation is chosen.

More precisely, we can determine between K choices $\{A_0, A_1, \cdots, A_{K-1}\}$ by calculating

$$\text{ANLE}(A_k) = \frac{1}{N} \sum_{i=0}^{N-1} \text{logit}(A_{k,i})$$

and finding the choice index k for $\max_{0 \leq k < K} \text{ANLE}(A_k)$.

Intuitively, adding together logits has an effect similar to adding directional vectors together; tokens with a large positive logit value act as strong positive directional vectors, tokens with smaller positive logit value act as weaker ones, and tokens with negative logit values act as directional vectors pointing in the opposite direction. This can be visualized by examining the sum as it moves along a reverse logistic curve.

B.1.1.3 Event-based evaluation

An alternative approach is to consider the model's outputs under the context of probability. This is motivated by leveraging the actual approach to free response generation in a constrained context. One can imagine a Monte Carlo simulation of many different free response inquiries to the model, and tallying how frequently the available choices occur. However, this is computationally expensive, and even infeasible for sufficiently long choices.

That being said, we need not actually employ such a simulation to calculate the odds of obtaining each choice. Using the softmax function, we can transform the logit space at a particular context point into a probability distribution, just as a transformer model would do to choose a token during free response generation. We can then chain these individual probabilities together to give an overall probability for the choice, rank our choices correspondingly, and choose the choice which is most likely to occur.

More precisely, when presented with K choices $\{A_0, A_1, \cdots, A_{K-1}\}$ and a question consisting of a string of H tokens, we can determine the most likely option the model would generate as $\arg\max_{0 \le j < K} P(A_j | H)$.

To arrive at $P(A|H)$, let $A = a_0 a_1 \cdots a_{N-1}$ be a string of tokens. Given a context string of tokens H, we can interpret A as a sequence of conditional events, and evaluate the probability of generating the string A given H as

$$P(A|H) = P(a_0|H) \times P(a_1|Ha_0) \times P(a_2|Ha_0a_1) \times \cdots \times P(a_{N-1}|Ha_0a_1 \cdots a_{N-2})$$
$$\equiv \prod_{j=0}^{N-1} P\left(a_j \middle| H \sum_{k=0}^{j-1} a_k\right)$$

(where $\sum$ here indicates concatenation). Each probability $P\left(a_j \middle| H \sum_{k=0}^{j-1} a_k\right)$ corresponds exactly to a_j in the softmax probability distribution of the model's logits after providing it the context $H \sum_{k=0}^{j-1} a_k$ of all tokens preceding it. Thus, each component probability is readily calculated, and consequently, so is $P(A|H)$.

Bibliography

[1] EMILY M. BENDER, TIMNIT GEBRU, ANGELINA MCMILLAN-MAJOR, AND SHMAR-GARET SHMITCHELL. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA, 2021. Association for Computing Machinery.

[2] TOM B. BROWN, BENJAMIN MANN, NICK RYDER, MELANIE SUBBIAH, JARED KAPLAN, PRAFULLA DHARIWAL, ARVIND NEELAKANTAN, PRANAV SHYAM, GIRISH SASTRY, AMANDA ASKELL, SANDHINI AGARWAL, ARIEL HERBERT-VOSS, GRETCHEN KRUEGER, TOM HENIGHAN, REWON CHILD, ADITYA RAMESH, DANIEL M. ZIEGLER, JEFFREY WU, CLEMENS WINTER, CHRISTOPHER HESSE, MARK CHEN, ERIC SIGLER, MATEUSZ LITWIN, SCOTT GRAY, BENJAMIN CHESS, JACK CLARK, CHRISTOPHER BERNER, SAM MCCANDLISH, ALEC RADFORD, ILYA SUTSKEVER, AND DARIO AMODEI. Language models are few-shot learners, 2020.

[3] KONSTANTIN CHERNYSHEV. Pypi/k4black/codebleu. https://pypi.org/project/codebleu/0.6.1/, 2024. Version 0.6.1.

[4] JACOB DEVLIN, MING-WEI CHANG, KENTON LEE, AND KRISTINA TOUTANOVA. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[5] ZHANGYIN FENG, DAYA GUO, DUYU TANG, NAN DUAN, XIAOCHENG FENG, MING GONG, LINJUN SHOU, BING QIN, TING LIU, DAXIN JIANG, AND MING ZHOU. Codebert: A pre-trained model for programming and natural languages, 2020.

[6] WEI GUO AND AYLIN CALISKAN. Detecting emergent intersectional biases: Contextualized word embeddings contain a distribution of human-like biases. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, AIES '21. ACM, July 2021.

[7] BEN HUTCHINSON, VINODKUMAR PRABHAKARAN, EMILY DENTON, KELLIE WEBSTER, YU ZHONG, AND STEPHEN DENUYL. Social biases in NLP models as barriers for persons with disabilities. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, pages 5491–5501, Online, July 2020. Association for Computational Linguistics.

[8] SIYUAN JIANG AND COLLIN MCMILLAN. Towards automatic generation of short summaries of commits, 2017.

[9] DANIEL MARTIN KATZ, MICHAEL JAMES BOMMARITO, SHANG GAO, AND PABLO ARREDONDO. Gpt-4 passes the bar exam. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 382(2270):20230254, 2024.

[10] CHIN-YEW LIN AND FRANZ JOSEF OCH. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland, aug 23–aug 27 2004. COLING.

[11] SHENG LU, IRINA BIGOULAEVA, RACHNEET SACHDEVA, HARISH TAYYAR MADABUSHI, AND IRYNA GUREVYCH. Are emergent abilities in large language models just in-context learning?, 2023.

[12] SHUAI LU, DAYA GUO, SHUO REN, JUNJIE HUANG, ALEXEY SVYATKOVSKIY, AMBROSIO BLANCO, COLIN B. CLEMENT, DAWN DRAIN, DAXIN JIANG, DUYU TANG,

GE LI, LIDONG ZHOU, LINJUN SHOU, LONG ZHOU, MICHELE TUFANO, MING GONG, MING ZHOU, NAN DUAN, NEEL SUNDARESAN, SHAO KUN DENG, SHENGYU FU, AND SHUJIE LIU. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[13] ERIK NIJKAMP, BO PANG, HIROAKI HAYASHI, LIFU TU, HUAN WANG, YINGBO ZHOU, SILVIO SAVARESE, AND CAIMING XIONG. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[14] DAVID N PALACIO, ALEJANDRO VELASCO, DANIEL RODRIGUEZ-CARDENAS, KEVIN MORAN, AND DENYS POSHYVANYK. Evaluating and explaining large language models for code using syntactic structures, 2023.

[15] KISHORE PAPINENI, SALIM ROUKOS, TODD WARD, AND WEI-JING ZHU. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[16] COLIN RAFFEL, NOAM SHAZEER, ADAM ROBERTS, KATHERINE LEE, SHARAN NARANG, MICHAEL MATENA, YANQI ZHOU, WEI LI, AND PETER J. LIU. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.

[17] EHUD REITER. A Structured Review of the Validity of BLEU. *Computational Linguistics*, 44(3):393–401, 09 2018.

[18] SHUO REN, DAYA GUO, SHUAI LU, LONG ZHOU, SHUJIE LIU, DUYU TANG, NEEL SUNDARESAN, MING ZHOU, AMBROSIO BLANCO, AND SHUAI MA. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[19] MARCO TULIO RIBEIRO, TONGSHUANG WU, CARLOS GUESTRIN, AND SAMEER SINGH. Beyond accuracy: Behavioral testing of nlp models with checklist, 2020.

[20] BAPTISTE ROZIÈRE, JONAS GEHRING, FABIAN GLOECKLE, STEN SOOTLA, ITAI GAT, XIAOQING ELLEN TAN, YOSSI ADI, JINGYU LIU, ROMAIN SAUVESTRE, TAL

REMEZ, JÉRÉMY RAPIN, ARTYOM KOZHEVNIKOV, IVAN EVTIMOV, JOANNA BITTON, MANISH BHATT, CRISTIAN CANTON FERRER, AARON GRATTAFIORI, WENHAN XIONG, ALEXANDRE DÉFOSSEZ, JADE COPET, FAISAL AZHAR, HUGO TOUVRON, LOUIS MARTIN, NICOLAS USUNIER, THOMAS SCIALOM, AND GABRIEL SYNNAEVE. Code llama: Open foundation models for code, 2024.

[21] RYLAN SCHAEFFER, BRANDO MIRANDA, AND SANMI KOYEJO. Are emergent abilities of large language models a mirage?, 2023.

[22] AAROHI SRIVASTAVA, ABHINAV RASTOGI, ABHISHEK RAO, ABU AWAL MD SHOEB, ABUBAKAR ABID, ADAM FISCH, ADAM R. BROWN, ADAM SANTORO, ADITYA GUPTA, ADRIÀ GARRIGA-ALONSO, AGNIESZKA KLUSKA, AITOR LEWKOWYCZ, AKSHAT AGARWAL, ALETHEA POWER, ALEX RAY, ALEX WARSTADT, ALEXANDER W. KOCUREK, ALI SAFAYA, ALI TAZARV, ALICE XIANG, ALICIA PARRISH, ALLEN NIE, AMAN HUSSAIN, AMANDA ASKELL, AMANDA DSOUZA, AMBROSE SLONE, AMEET RAHANE, ANANTHARAMAN S. IYER, ANDERS ANDREASSEN, ANDREA MADOTTO, ANDREA SANTILLI, ANDREAS STUHLMÜLLER, ANDREW DAI, ANDREW LA, ANDREW LAMPINEN, ANDY ZOU, ANGELA JIANG, ANGELICA CHEN, ANH VUONG, ANIMESH GUPTA, ANNA GOTTARDI, ANTONIO NORELLI, ANU VENKATESH, ARASH GHOLAMIDAVOODI, ARFA TABASSUM, ARUL MENEZES, ARUN KIRUBARAJAN, ASHER MULLOKANDOV, ASHISH SABHARWAL, AUSTIN HERRICK, AVIA EFRAT, AYKUT ERDEM, AYLA KARAKAŞ, B. RYAN ROBERTS, BAO SHENG LOE, BARRET ZOPH, BARTŁOMIEJ BOJANOWSKI, BATUHAN ÖZYURT, BEHNAM HEDAYATNIA, BEHNAM NEYSHABUR, BENJAMIN INDEN, BENNO STEIN, BERK EKMEKCI, BILL YUCHEN LIN, BLAKE HOWALD, BRYAN ORINION, CAMERON DIAO, CAMERON DOUR, CATHERINE STINSON, CEDRICK ARGUETA, CÉSAR FERRI RAMÍREZ, CHANDAN SINGH, CHARLES RATHKOPF, CHENLIN MENG, CHITTA BARAL, CHIYU WU, CHRIS CALLISON-BURCH, CHRIS WAITES, CHRISTIAN VOIGT, CHRISTOPHER D. MANNING, CHRISTOPHER POTTS, CINDY RAMIREZ,

CLARA E. RIVERA, CLEMENCIA SIRO, COLIN RAFFEL, COURTNEY ASHCRAFT, CRISTINA GARBACEA, DAMIEN SILEO, DAN GARRETTE, DAN HENDRYCKS, DAN KILMAN, DAN ROTH, DANIEL FREEMAN, DANIEL KHASHABI, DANIEL LEVY, DANIEL MOSEGUÍ GONZÁLEZ, DANIELLE PERSZYK, DANNY HERNANDEZ, DANQI CHEN, DAPHNE IPPOLITO, DAR GILBOA, DAVID DOHAN, DAVID DRAKARD, DAVID JURGENS, DEBAJYOTI DATTA, DEEP GANGULI, DENIS EMELIN, DENIS KLEYKO, DENIZ YURET, DEREK CHEN, DEREK TAM, DIEUWKE HUPKES, DIGANTA MISRA, DILYAR BUZAN, DIMITRI COELHO MOLLO, DIYI YANG, DONG-HO LEE, DYLAN SCHRADER, EKATERINA SHUTOVA, EKIN DOGUS CUBUK, ELAD SEGAL, ELEANOR HAGERMAN, ELIZABETH BARNES, ELIZABETH DONOWAY, ELLIE PAVLICK, EMANUELE RODOLA, EMMA LAM, ERIC CHU, ERIC TANG, ERKUT ERDEM, ERNIE CHANG, ETHAN A. CHI, ETHAN DYER, ETHAN JERZAK, ETHAN KIM, EUNICE ENGEFU MANYASI, EVGENII ZHELTONOZHSKII, FANYUE XIA, FATEMEH SIAR, FERNANDO MARTÍNEZ-PLUMED, FRANCESCA HAPPÉ, FRANCOIS CHOLLET, FRIEDA RONG, GAURAV MISHRA, GENTA INDRA WINATA, GERARD DE MELO, GERMÁN KRUSZEWSKI, GIAMBATTISTA PARASCANDOLO, GIORGIO MARIANI, GLORIA WANG, GONZALO JAIMOVITCH-LÓPEZ, GREGOR BETZ, GUY GUR-ARI, HANA GALIJASEVIC, HANNAH KIM, HANNAH RASHKIN, HANNANEH HAJISHIRZI, HARSH MEHTA, HAYDEN BOGAR, HENRY SHEVLIN, HINRICH SCHÜTZE, HIROMU YAKURA, HONGMING ZHANG, HUGH MEE WONG, IAN NG, ISAAC NOBLE, JAAP JUMELET, JACK GEISSINGER, JACKSON KERNION, JACOB HILTON, JAEHOON LEE, JAIME FERNÁNDEZ FISAC, JAMES B. SIMON, JAMES KOPPEL, JAMES ZHENG, JAMES ZOU, JAN KOCOŃ, JANA THOMPSON, JANELLE WINGFIELD, JARED KAPLAN, JAREMA RADOM, JASCHA SOHL-DICKSTEIN, JASON PHANG, JASON WEI, JASON YOSINSKI, JEKATERINA NOVIKOVA, JELLE BOSSCHER, JENNIFER MARSH, JEREMY KIM, JEROEN TAAL, JESSE ENGEL, JESUJOBA ALABI, JIACHENG XU, JIAMING SONG, JILLIAN TANG, JOAN WAWERU, JOHN BURDEN, JOHN MILLER, JOHN U. BALIS, JONATHAN BATCHELDER, JONATHAN BERANT, JÖRG FROHBERG,

Jos Rozen, Jose Hernandez-Orallo, Joseph Boudeman, Joseph Guerr, Joseph Jones, Joshua B. Tenenbaum, Joshua S. Rule, Joyce Chua, Kamil Kanclerz, Karen Livescu, Karl Krauth, Karthik Gopalakrishnan, Katerina Ignatyeva, Katja Markert, Kaustubh D. Dhole, Kevin Gimpel, Kevin Omondi, Kory Mathewson, Kristen Chiafullo, Ksenia Shkaruta, Kumar Shridhar, Kyle McDonell, Kyle Richardson, Laria Reynolds, Leo Gao, Li Zhang, Liam Dugan, Lianhui Qin, Lidia Contreras-Ochando, Louis-Philippe Morency, Luca Moschella, Lucas Lam, Lucy Noble, Ludwig Schmidt, Luheng He, Luis Oliveros Colón, Luke Metz, Lütfi Kerem Şenel, Maarten Bosma, Maarten Sap, Maartje ter Hoeve, Maheen Farooqi, Manaal Faruqui, Mantas Mazeika, Marco Baturan, Marco Marelli, Marco Maru, Maria Jose Ramírez Quintana, Marie Tolkiehn, Mario Giulianelli, Martha Lewis, Martin Potthast, Matthew L. Leavitt, Matthias Hagen, Mátyás Schubert, Medina Orduna Baitemirova, Melody Arnaud, Melvin McElrath, Michael A. Yee, Michael Cohen, Michael Gu, Michael Ivanitskiy, Michael Starritt, Michael Strube, Michał Swędrowski, Michele Bevilacqua, Michihiro Yasunaga, Mihir Kale, Mike Cain, Mimee Xu, Mirac Suzgun, Mitch Walker, Mo Tiwari, Mohit Bansal, Moin Aminnaseri, Mor Geva, Mozhdeh Gheini, Mukund Varma T, Nanyun Peng, Nathan A. Chi, Nayeon Lee, Neta Gur-Ari Krakover, Nicholas Cameron, Nicholas Roberts, Nick Doiron, Nicole Martinez, Nikita Nangia, Niklas Deckers, Niklas Muennighoff, Nitish Shirish Keskar, Niveditha S. Iyer, Noah Constant, Noah Fiedel, Nuan Wen, Oliver Zhang, Omar Agha, Omar Elbaghdadi, Omer Levy, Owain Evans, Pablo Antonio Moreno Casares, Parth Doshi, Pascale Fung, Paul Pu Liang, Paul Vicol, Pegah Alipoormolabashi, Peiyuan Liao, Percy Liang, Peter Chang, Peter Eckersley, Phu Mon Htut, Pinyu Hwang, Piotr Miłkowski, Piyush Patil, Pouya Pezeshkpour, Priti

OLI, QIAOZHU MEI, QING LYU, QINLANG CHEN, RABIN BANJADE, RACHEL ETTA RUDOLPH, RAEFER GABRIEL, RAHEL HABACKER, RAMON RISCO, RAPHAËL MILLIÈRE, RHYTHM GARG, RICHARD BARNES, RIF A. SAUROUS, RIKU ARAKAWA, ROBBE RAYMAEKERS, ROBERT FRANK, ROHAN SIKAND, ROMAN NOVAK, ROMAN SITELEW, RONAN LEBRAS, ROSANNE LIU, ROWAN JACOBS, RUI ZHANG, RUSLAN SALAKHUTDINOV, RYAN CHI, RYAN LEE, RYAN STOVALL, RYAN TEEHAN, RYLAN YANG, SAHIB SINGH, SAIF M. MOHAMMAD, SAJANT ANAND, SAM DILLAVOU, SAM SHLEIFER, SAM WISEMAN, SAMUEL GRUETTER, SAMUEL R. BOWMAN, SAMUEL S. SCHOENHOLZ, SANGHYUN HAN, SANJEEV KWATRA, SARAH A. ROUS, SARIK GHAZARIAN, SAYAN GHOSH, SEAN CASEY, SEBASTIAN BISCHOFF, SEBASTIAN GEHRMANN, SEBASTIAN SCHUSTER, SEPIDEH SADEGHI, SHADI HAMDAN, SHARON ZHOU, SHASHANK SRIVASTAVA, SHERRY SHI, SHIKHAR SINGH, SHIMA ASAADI, SHIXIANG SHANE GU, SHUBH PACHCHIGAR, SHUBHAM TOSHNIWAL, SHYAM UPADHYAY, SHYAMOLIMA, DEBNATH, SIAMAK SHAKERI, SIMON THORMEYER, SIMONE MELZI, SIVA REDDY, SNEHA PRISCILLA MAKINI, SOOHWAN LEE, SPENCER TORENE, SRIHARSHA HATWAR, STANISLAS DEHAENE, STEFAN DIVIC, STEFANO ERMON, STELLA BIDERMAN, STEPHANIE LIN, STEPHEN PRASAD, STEVEN T. PIANTADOSI, STUART M. SHIEBER, SUMMER MISHERGHI, SVETLANA KIRITCHENKO, SWAROOP MISHRA, TAL LINZEN, TAL SCHUSTER, TAO LI, TAO YU, TARIQ ALI, TATSU HASHIMOTO, TE-LIN WU, THÉO DESBORDES, THEODORE ROTHSCHILD, THOMAS PHAN, TIANLE WANG, TIBERIUS NKINYILI, TIMO SCHICK, TIMOFEI KORNEV, TITUS TUNDUNY, TOBIAS GERSTENBERG, TRENTON CHANG, TRISHALA NEERAJ, TUSHAR KHOT, TYLER SHULTZ, URI SHAHAM, VEDANT MISRA, VERA DEMBERG, VICTORIA NYAMAI, VIKAS RAUNAK, VINAY RAMASESH, VINAY UDAY PRABHU, VISHAKH PADMAKUMAR, VIVEK SRIKUMAR, WILLIAM FEDUS, WILLIAM SAUNDERS, WILLIAM ZHANG, WOUT VOSSEN, XIANG REN, XIAOYU TONG, XINRAN ZHAO, XINYI WU, XUDONG SHEN, YADOLLAH YAGHOOBZADEH, YAIR LAKRETZ, YANGQIU SONG, YASAMAN BAHRI, YEJIN

CHOI, YICHI YANG, YIDING HAO, YIFU CHEN, YONATAN BELINKOV, YU HOU, YUFANG HOU, YUNTAO BAI, ZACHARY SEID, ZHUOYE ZHAO, ZIJIAN WANG, ZI-JIE J. WANG, ZIRUI WANG, AND ZIYI WU. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2023.

[23] WEI TAO, YANLIN WANG, ENSHENG SHI, LUN DU, SHI HAN, HONGYU ZHANG, DONGMEI ZHANG, AND WENQIANG ZHANG. On the evaluation of commit message generation models: An experimental study, 2021.

[24] HUGO TOUVRON, THIBAUT LAVRIL, GAUTIER IZACARD, XAVIER MARTINET, MARIE-ANNE LACHAUX, TIMOTHÉE LACROIX, BAPTISTE ROZIÈRE, NAMAN GOYAL, ERIC HAMBRO, FAISAL AZHAR, AURELIEN RODRIGUEZ, ARMAND JOULIN, EDOUARD GRAVE, AND GUILLAUME LAMPLE. Llama: Open and efficient foundation language models, 2023.

[25] ASHISH VASWANI, NOAM SHAZEER, NIKI PARMAR, JAKOB USZKOREIT, LLION JONES, AIDAN N GOMEZ, Ł UKASZ KAISER, AND ILLIA POLOSUKHIN. Attention is all you need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, volume 30. Curran Associates, Inc., 2017.

[26] YUE WANG, WEISHI WANG, SHAFIQ JOTY, AND STEVEN C. H. HOI. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[27] JASON WEI, YI TAY, RISHI BOMMASANI, COLIN RAFFEL, BARRET ZOPH, SEBASTIAN BORGEAUD, DANI YOGATAMA, MAARTEN BOSMA, DENNY ZHOU, DONALD METZLER, ED H. CHI, TATSUNORI HASHIMOTO, ORIOL VINYALS, PERCY LIANG, JEFF DEAN, AND WILLIAM FEDUS. Emergent abilities of large language models, 2022.

[28] SHENGBIN XU, YUAN YAO, FENG XU, TIANXIAO GU, HANGHANG TONG, AND JIAN LU. Commit message generation for source code changes. In *Proceedings of the*

Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, pages 3975–3981. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[29] LINGHAO ZHANG, JINGSHU ZHAO, CHONG WANG, AND PENG LIANG. Using large language models for commit message generation: A preliminary study, 2024.