An Investigation into the Distillation of Code Intelligence from Large Language Models

James Doran

Master of Science
Cognitive Science
School of Informatics
University of Edinburgh
2023

Abstract

In the last few years, significant progress has been made in the field of code intelligence. AI models exist that are able to write working code for a solving a given task, automatically identify bugs and effectively document code. Unfortunately, these abilities are only exhibited by giant, closed-source large language models (LLMs) which is a barrier to the availability and development of these coding capabilities. Recent work has demonstrated that an effective way to transfer abilities from these LLMs is to train smaller models with synthetic data generated through the LLM's public API.

This project explores the extension of this synthetic data generation procedure to the coding domain, specifically the task of semantic code search. We implement and evaluate three types of synthetic data generating strategies; generating bimodal data from a unimodal source, generating hard examples to augment existing data and generating a fully synthetic dataset from scratch. We show that all of these methods produce synthetic datasets that result in a noticeable improvement in MRR on CodeXGLUE adversarial code search task when added to a smaller real dataset. We find that all synthetic data that is conditioned on real data results in better performance, with the greatest improvement coming from the dataset composed of real code with synthesised docstrings. We also find that the value of synthetic data is greatest when a large amount of synthetic data is combined with a small amount of real data.

Finally, we present a collection of synthetic data analysis techniques and show a connection to the empirical results. We make our analysis and synthetic data generation code available on GitHub so that it can be used for further research.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*James Doran*)

Acknowledgements

I would like to acknowledge my Amazon supervisors, Prarit and Camille, for their insightful guidance, my group members for their helpful discussions, and my girlfriend, Daria, for housing and supporting me through the process.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

As of 2023, there is a big gap in the capabilities of AI systems owned by private corporations and those that are available to the public and academia. Giant, successful technology companies have vast financial, computational and intellectual resources available to them which dwarf those available to individuals or smaller institutions. On top of this, the field of natural language processing has been moving towards the development of larger and larger models ever since its efficacy was demonstrated by Raffel et al. and Brown et al.

One of the most notable capabilities that remains exclusive to privately-owned large language models (LLMs) is the ability to generate and understand computer code. Figure 1.1 highlights the full extent of the disparity showing that most open-source models (all models right of Bard) have significantly worse performance than the five closed-source systems, most obviously GPT-4[1].

Although the model parameters are not public, most of the closed-source LLMs such as GPT-4 are released as APIs that can be used by the public for a modest price. Given this form of access, these LLMs can be used to generate high-quality "synthetic" training data that can be used to create smaller, specialised models. Recent work (*Stanford Alpaca*, Eldan and Y. Li, etc.) has shown that this "knowledge distillation" procedure is effective for transferring a variety of natural language capabilities. The goal of this work is to investigate how it can be applied to code intelligence tasks to bring the performance of open-source models closer to that of leading private LLMs.

[1] although GPT-4 is known to contain HumanEval in its pretraining dataset

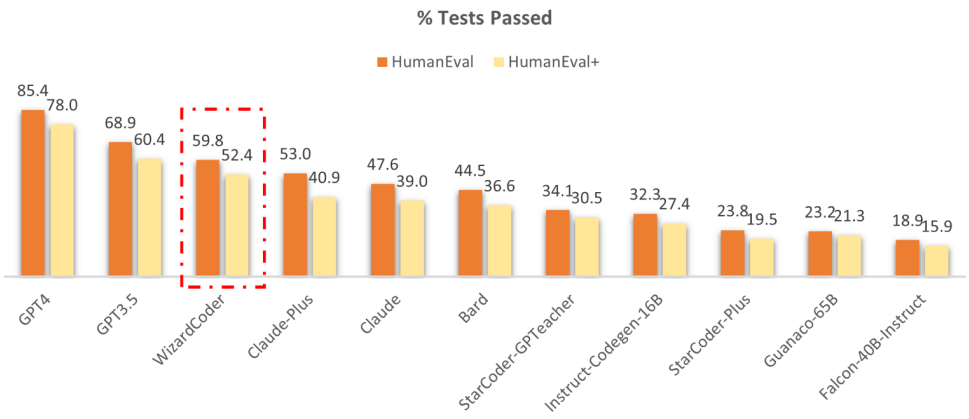


Figure 1.1: Performance of various LLMs on HumanEval, a program synthesis task where models are given a coding problem have to generate a program that passes test cases. The top performers (excluding WizardCoder) are all closed-source LLMs. Figure taken from Luo et al.

1.2 Wider impact

Apart from allowing more people to benefit from the advances in LLMs, research into deep learning with synthetic data has wider implications.

In general, machine learning models with more parameters require more training data to effectively generalise (Bishop). The size and scale of today's LLMs mean that they must be trained on trillions of words compiled from internet resources (Brown et al.). Villalobos et al. suggest that the rate at which LLMs are increasing in data requirements is exceeding the rate that the human race is producing new, especially high-quality, language data. Their model predicts that within the next decade the availability of high-quality language data for training larger LLMs will be limited by the speed at which humanity produces it. Given the impending arrival of this language data crisis, one answer could be to synthetically generate these language resources using LLMs, and use it to recursively improve upon their own abilities (Mukherjee et al.).

In contrast to this, Shumailov et al. believe that this form of recursive training can damage the performance and capabilities of generative models. They point out that the the public availability of generative AIs will inevitably lead to public data sources becoming inundated with synthetically generated data. This highlights another greater need for research in synthetic data – now that it may be largely indistinguishable from human-generated data it is likely to exist in all future data sources. This makes it even more important to understand its impact on deep learning systems so that future negative

effects can be mitigated.

1.3 Goals

The overarching goal of this project is to document the efficacy of learning from LLM-generated synthetic data for code intelligence tasks. This will be assessed by comparing the performance of systems trained on synthetic data relative to system using real data. We hypothesise that synthetic data will be able to produce systems of equivalent or superior capability capability to those using only real data. Additionally, we hope to describe analysis techniques that can be used to predict the efficacy of introducing synthetic data to deep learning training datasets.

1.4 Overview

This document describes an investigation into applying LLM-generated synthetic data to the code intelligence task of semantic code search. It is divided into the following chapters:

- Chapter 2 reviews some of the connected scientific literature serving as the inspiration and context for this work.
- Chapter 3 describes the structure of the experiments and synthetic data generation processes while discussing the experimental design choices.
- Chapter 4 reports some of the challenges that were overcome to set up the experiments.
- Chapter 5 reports and analyses the results of the experiments in three parts. Section 5.1 discusses empirical performance, Section 5.2 analyses features intrinsic to the synthetic datasets and Section 5.3 discusses some of the limitations of the project.
- Chapter 6 concludes and discusses directions for future work.

Chapter 2

Background

This chapter features an overview of the scientific literature that underpins this project. For further context, see the associated informatics project proposal document.

2.1 Large language models

A language model is statistical model of language that assigns probabilities to sequences, generally text. The most accurate and useful language models around today are artificial neural networks (ANNs) that are trained on a large corpus of textual data through deep learning. ANN language models have been the cornerstone of contemporary NLP techniques because they learn latent vector representations of language that have been shown to improve NLP tasks that they are used for (e.g. question answering, intent classification, etc.) (Mikolov et al.).

The ANN architecture used for language modelling can vary but almost all relevant ones (as of 2023) use the Transformer architecture of Vaswani et al. The success of Transformers can be largely attributed to their ability to take advantage of the massive parallel computation capabilities of modern GPUs[1]. Devlin et al. showed that this architecture could be used to train much larger language models on much larger corpora – and that initialising a network with this model (BERT) improved performance on almost every natural language benchmark. This pretrain-on-language-modelling/fine-tune-on-task paradigm opened the door for even bigger language models because they could be reused for many different tasks.

Today, technology companies create Transformer language models that have so many parameters that the training process costs millions of dollars in computer power

[1] as well as the ability to model long distance dependencies

alone. These systems are known in the field[2] as Large Language Models (LLMs) and feature impressive emergent abilities. Most notably, they are able to perform tasks based on a vague description where most machine learning systems require hundreds of examples (Brown et al.). Most companies do not not release the full systems to the public but make them accessible through a paid API (e.g Google's PaLM2 and OpenAI's GPT-3.5/4).

2.2 Language models for code intelligence

Transformers have also been effectively applied to the adjacent domain of understanding and generating programming languages (code intelligence). Inspired by the success of Devlin et al., Feng et al. apply the same principle to learning a language model for code: CodeBERT. They initialise the model from RoBERTa (a BERT-based language model from Liu et al.) and train it to predict masked and replaced tokens in docstring-code sequence pairs. In the same way as BERT, this develops rich internal representations that transfer to downstream code intelligence tasks. Extensions of this work predominantly involve additional pretraining tasks that incorporate special features of code. For example, GraphCodeBERT (Guo et al.) includes a pretraining task where the model must predict a data flow graph between the variables and UniXCoder (Guo et al.) is pretrained on code that has been parsed into a flattened abstract syntax tree.

Coding abilities of these models seem to be closely linked with model size (M. Chen et al.) and is considered one of the emergent abilities of scaling language models. As of 2023, the systems with the best performance on coding tasks are all closed-source LLMs[3], none of which have been trained explicitly for code intelligence (in contrast to previous efforts like CodeBERT).

2.3 Synthetic data generation

Synthetic data is data that has been created through some artificial process rather than collected/sampled from the real data distribution. Traditionally this has been used in the computer vision domain to create additional training examples featuring flipped, rotated and re-scaled versions of real data. T. Chen et al. that combining these "data augmentations" contributes significantly to the quality of learned representations.

[2], creatively,

[3]OpenAI's GPT-4 leads by a large margin

Analogous techniques exist for the natural language and coding domains but are less widespread. For example, words can be replaced with synonyms and code identifiers can be renamed without changing the overall meaning (Park et al.).

One of the more interesting ways to create synthetic data to use a generative model. For example, GANs (Goodfellow et al.) feature a discriminator network that is trained on the outputs of a data generator network. Another classic case is the process of back-translation from Sennrich et al. which uses a language-A-to-B machine translation model to generate language pairs from monolingual (language-A) data for training a language-B-to-A translation model.

LLMs are now capable of producing text that is almost indistinguishable from (and in some cases of a higher quality than) text created by humans (Cegin et al.). This has inspired some work using text from these models as a substitute for human-generated data and has shown promising results for several tasks. *Stanford Alpaca* is a language model that has been trained to follow instructions using a synthetic dataset of problems generated through the OpenAI GPT-3 API. Xu et al. also attempt to replicate a general purpose language model but use a special instruction data generation process that uses an LLM to produce progressively more complicated tasks to train the model to perform. At this point in time it is one of the most powerful open-source language models which demonstrates how effective training on LLM outputs can be.

2.3.1 Reasoning distillation

Reasoning (multi-step problem solving) is an ability of large language models that is closely connected to code intelligence. Several works highlight the connection between coding and reasoning abilities. For example Madaan et al. and Gao et al. show that complex reasoning tasks benefit from being expressed as coding tasks for language models that have been trained on code. Aside from this, strong performance on reasoning benchmarks is only achieved by the largest of LLMs, much like performance on coding tasks (OpenAI).

This has inspired research that attempts to replicate the reasoning abilities of LLMs in smaller models. Shridhar et al. and Magister et al. show that training models on chain-of-thought generations from large models somewhat successfully transfers reasoning abilities. Zelikman et al. use a language model's own chain-of-thought generations[4] as more training examples for itself.

[4]those that result in the correct final answer

2.4 Contrastive learning

This project makes use of a contrastive learning objective to learn a joint embedding space of natural language and code. The techniques used in this project are largely founded upon the in-batch contrastive loss used in works such as Karpukhin et al. and T. Chen et al. Essentially, the representations are learnt through a cross-entropy classification loss where each representation must have the maximum dot product with its respective pair while minimising its dot product with others in the batch. This is apparent from the standard formulation of the in-batch negative contrastive loss:

$$\mathcal{L} = \frac{1}{|B|} \sum_{x \in B} -log \frac{exp(f(x)^\top f(x^+))}{\sum_{x' \in B} exp(f(x)^\top f(x'))}, \tag{2.1}$$

where f is the encoder network, B is a batch of examples and x^+ is a positive example corresponding to x.

Quite a few papers use contrastive learning to learn code representations either for semantic code search or for code clone detection. In the original CodeBERT paper, Feng et al. learn representations using this contrastive learning configuration. Neelakantan et al. also do so and achieve state-of-the-art results by using a very large batch-size and proprietary base model. X. Li et al. uses the same loss function but combine it with a variety of strategies that insert more challenging examples into each batch during training and demonstrate very good performance on various semantic code search benchmarks.

Finally, Schick and Schütze provides a framework for generating and filtering synthetic data. They demonstrate results on a sentence representation learning task which shows that it is feasible to contrastively learn effective representations using synthetic data.

2.5 Similar concurrent work

Contemporaneously to this project (both uploaded in June 2023), other authors have attempted to distill code intelligence from LLMs in a similar manner.

Luo et al. use the techniques learned from WizardLM (briefly discussed in section 2.3) to generate a synthetic dataset with a technique that mutates existing coding tasks into more complicated ones using an LLM.

Gunasekar et al. observe that most code in model training datasets has limited educational value. They use GPT-4 to generate a high-quality synthetic dataset of code

(textbook style code and exercises) that is more suitable for learning from. This process yields a model that is not only superior to competing open-source models of code but is also a tenth of their size.

Both models are at the very top of the open-source leaderboard for the HumanEval code generation benchmark suggesting that this method of improving open-source models is a very effective one. They both share the underlying principle of this work; training on the outputs of superior models of code. However, this project focuses on applying it to the code search domain rather than program synthesis.

Chapter 3

Methodology

The primary goal of this work is to investigate the extent to which the code intelligence of large, black-box language models can be distilled through training on their generations. To do this we have to demonstrate that using synthetic data from these models confers a measurable performance on code intelligence tasks.

3.1 Semantic code search

The main code intelligence task that this project focuses on is the task of semantic code search (also called natural language code search). The goal of a semantic code search system is to allow users to find code that they are looking for given only the meaning of the code, described in natural language. Software developers frequently need to find a particular piece of code within a very large collection; this could range from searching the internet for an implementation of an algorithm to searching an unfamiliar codebase for a particular component

The standard strategy for textual informational retrieval over large datasets is to use substring and sparse vector comparison algorithms (listed as "traditional" baselines in Craswell et al.). However, in most situations, it is unreasonable to expect the users of these systems to know the exact sequence of characters that they are looking for. For example, when looking for an algorithm implementation they may only know the name and purpose of the algorithm rather than the exact variable and function names.

Systems that understand and operate over an extensive corpus of documents have been receiving greater attention recently. Despite the numerous achievements of modern LLMs, they have a finite[1] amount of knowledge available to them. The generations

[1] but vast

they produce are limited to text seen in their one-off pretraining process as well as what can be included in their fixed-length context window. Due to the quadratic scaling of the attention mechanism (Vaswani et al.) that these models use, training a model with a wide enough context to fit an entire internet-scale search database would be impossible[2]. Many systems today (e.g. Lu et al., Lewis et al.) combine the natural language abilities of LLMs and the scalability of semantic search systems under the paradigm of retrieval-augmented generation (RAG). Thus, semantic search serves as a valuable complement to the contemporary advancements in LLM capabilities.

The task benefits from a strong understanding of both code and natural language, yet Neelakantan et al. show that it does not require a prohibitively large base-model. This combination of semantic depth and convenient model-sizes is why we choose it for this synthetic data investigation.

3.1.1 Coding domain challenges

It is tempting to suggest that the task of semantic code search is subsumed and therefore solved by the natural language semantic search systems such as those submitted to TREC (Craswell et al.). Indeed, the architectures of high-performing semantic code search systems are largely the same as natural language ones, but the coding domain brings a number of special challenges.

Programming languages differ from natural language in some ways that make the search task harder. According to Husain et al., there is much less overlap between the natural language queries and the code documents which causes traditional text retrieval systems to struggle. X. Li et al. suggest that code tends to have a large imbalance between its high and low frequency tokens[3] which B. Li et al. show negatively impacts the representations learnt by a transformer model.

Finally, a large driver of the improvement of neural natural language search systems was the release of the vast datasets such as MS MARCO (Bajaj et al.). Until 2019, resources of this scale were not available for the task.

[2] Sun et al. show how many of the techniques that allow for larger context windows produce models that still struggle with long-context tasks

[3] i.e. it is dominated by keywords and characters like “if” and “;”

3.2 CodeSearchNet

CodeSearchNet (CSN) is a corpus and benchmark for semantic code search released by Husain et al. in 2019. It consists of 2.3 million function-docstring pairs and 4.1 million unlabeled functions across six coding languages[4]. The data was collected by GitHub and sourced from public, permissively-licensed repositories hosted on their site. The creators also provide a testing set which consists of 99 natural language queries from Bing searches and annotations for the 10 most relevant queries. It is a very popular source of data for pretraining language models for code intelligence, even in 2023 (Guo et al., Wang et al.). In April 2023, the CodeSearchNet challenge was officially concluded to encourage the use of newer benchmarks.

3.2.1 CodeXGLUE/CodeSearchNet-AdvTest

CodeXGLUE (Lu et al.) is a suite of code intelligence benchmarks that covers various tasks including code generation, clone detection, defect detection and natural language code search. CodeXGLUE features three different semantic code search tasks: CodeSearchNet-WebQueryTest, CoSQA and CodeSearchNet-AdvTest. The first two are focused on a binary task of identifying whether a code snippet is relevant to a natural language search query and are not used in this work. For this project, the experiments and evaluations will be conducted on the CodeSearchNet-AdvTest Python dataset (CSN-Adv). The training data they provide consists of 251,820 docstring-function examples which are filtered for quality[5] from the original CodeSearchNet corpus. The more significant difference from the CSN challenge is the testing set. In the CodeSearchNet testing task, systems must rank 1000 functions according to 99 real search queries. CSN-Adv instead tests on a task that is more similar to the training task: they use a dataset of 19,210 docstring-function pairs where each example must be ranked against all 19,210 other examples. In addition to this, all identifiers[6] in the functions have been replaced with generic labels. Conceptually, renaming these identifiers should not affect the semantics of the underlying function and helps to assess the generalisation performance of the model.

The result of both of these changes is that CSN-Adv is a significantly more challenging benchmark. Lu et al. report that the Python performance of CodeBERT (Feng

[4]Go, Java, JavaScript, PHP, Python, Ruby
[5]They remove non-English, long and ill-formed functions
[6]e.g. function, parameter and variable names

et al.) drops from an MRR (see section 3.3) of 0.8685 on CodeSearchNet to only 0.2719 on CSN-Adv. This makes it more suitable for the comparison of contemporary, high-performance systems.

Unfortunately, the CSN-Adv testing set no longer contains natural code search queries and is thus no longer an direct assessment of the performance on semantic code search. Instead assesses a more contrived task of searching for a function based on its docstring. The other CodeXGLUE search tasks, CoSQA and CodeSearchNet-WebQueryTest, are designed for this purpose but are significantly smaller because manually annotating natural retrieval data is very labour intensive. We do not consider this an issue because the focus of this project is on applying LLM synthetic data to code intelligence in general for which the CSN-Adv dataset is equivalent for, despite being a departure from the standard application of semantic code search.

3.3 Evaluation metric

Semantic code search is generally evaluated using standard information retrieval metrics. The one that is generally reported for the CSN-Adv test dataset is mean reciprocal rank (MRR). The equation to calculate this over ranking dataset $\mathcal{D}$ is

$$MRR_{\mathcal{D}} = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \frac{1}{rank_x}, \tag{3.1}$$

where $rank_x$ is the rank of the first relevant entry to query x. From the construction of the metric it is clear that it ranges from 1 when every query ranks a relevant entry first to $\frac{1}{|\mathcal{D}|}$ when the only relevant entry is always the last possible rank[7]. In practice, if $rank_x$ is greater than some number (100 for CSN-Adv) a score of zero is assigned to that query as if $rank_x = \infty$.

The main flaws with MRR are that it only takes into account the topmost relevant example and only supports a binary definition of relevance. The CSN-Adv dataset only features a single relevant entry for each query (the function that the docstring was taken from) so these restrictions are not an issue but the original CodeSearchNet task uses normalised discounted cumulative gain because it has graded relevance labels and multiple relevant entries.

The experiments on CSN-Adv in this project will be evaluated using MRR because it is the standard for the benchmark and will allow the results to be comparable with

[7] although inverting the score could produce a very effective ranker

other systems. It is also a simple, interpretable metric that corresponds well with the goal of the task.

3.4 Baseline

The standard strategy to create semantic code search systems is to perform a vector similarity comparison (dot product, cosine distance or euclidean distance) between an embedding of the query and the embeddings of every entry (dense retrieval). This transforms the task into learning a model that projects code and natural language input into joint embedding space. The most successful embedding models are large transformers that have been pretrained on modelling code and/or natural language (Devlin et al.) which are then adapted to produce vector representations through contrastive learning. Karpukhin et al. demonstrates the effectiveness of this architecture on a natural language question answering task.

In this work we use the CodeBERT model as a base for learning both the natural language and code embedding models. The model is then further trained on the CSN-Adv dataset using a contrastive loss function (see section 2.4) to minimise the dot product between the function and docstring vectors. This is the configuration tested with the original release of the CSN-Adv which has since been surpassed by more capable systems of a similar architecture (Guo et al., Guo et al., X. Li et al.). As the main focus of the project is on manipulating the data distribution we elect to conduct all experiments with this simple set-up.

3.5 Incorporating synthetic data

Given that some form of synthetic data is available, there is a number of ways that that it can be introduced to the baseline system described in section 3.4. Each of these will be the subject of experimentation using LLM-generated synthetic data.

3.5.1 Novel examples

The most obvious way to include synthetic data $\mathcal{D}_S$ is to use it to supplement your available data $\mathcal{D}$;

$$\mathcal{D}_{supp} = \mathcal{D} \cup \mathcal{D}_S. \tag{3.2}$$

Theoretically, $\mathcal{D}_{supp}$ can provide a lower variance estimate of $P(\mathcal{D})$, the true (future) data distribution, as long as the synthetic data is sampled from a similar distribution (relatively unbiased). For example, if the synthetic data is drawn from the same distribution as the training data it will essentially create a larger training set that will reduce the noise in the model estimates. Alternatively, if the synthetic data is systematically biased away from the distribution of the real data, the model learnt on the dataset will feature a systematic error in its predictions of real data. In practice it is hard to analytically quantify 1) the bias of synthetic data, 2) the variability of complicated models and 3) their exact trade-off so we must assess the value of the novel data empirically.

This technique is especially relevant if the original dataset is very small, where the synthetic data can act as a regulariser and provide examples in lower density, unsampled regions of the data distribution. In some situations it may be possible to generate synthetic data such that it better represents future data than $\mathcal{D}$, e.g. generating synthetic data that is free of biases present in historical data (Jordon et al.).

One special case of this strategy is when $\mathcal{D} = \emptyset$ and $\mathcal{D}_{supp} = \mathcal{D}_S$, i.e. when collecting real data is impossible or prohibitively expensive. In this situation, any ability to perform the desired task can be considered valuable.

3.5.2 Data augmentation (hard positives)

The field of computer vision has a long history of working with synthetic data through a procedure known as "data augmentation" (Lecun et al.). This involves creating a synthetic dataset $\mathcal{D}_{aug}$ using an augmentation procedure $f : X \rightarrow X^N$ that creates N new examples from each example which are then added to the dataset as in section 3.5.1.

In computer vision, this augmentation procedure usually involves image transformations that preserve the semantics of the image (e.g. $f(x) = \{x, greyscale(x), flip(x), rotate(x)\}$. These are used are positive pairs for contrastive learning and allows the models trained on $\mathcal{D}_{aug}$ to be more robust to those forms of non-semantic variance. T. Chen et al. show that including these augmentations confers benefits to the learned representations, especially if multiple augmentations are composed (i.e. $f_i \cdot f_j \in f$).

There is some work in the field of code representation learning that applies similar techniques to code. Bui et al., Jain et al. and Park et al. all generate positive pairs by performing semantic-preserving code transformations such as renaming identifiers, reordering statements and performing manually defined substitutions (some based on

compiler tricks). Each method is shown to improve downstream performance on code search and/or code clone detection task.

Not every domain features convenient semantics-preserving transformations like this. Natural language understanding, an integral part of semantic code search, has not seen significant benefits from augmentations such as synonym substitutions and sentence rewriting. Nikolenko suggest that this is due to a difference from computer vision where the field is able to benefit from the field of computer graphics. Despite this, Huang et al. show that performing simple augmentations on code search queries such as switching two words or deleting unimportant ones provides a marginal benefit.

The main experiments of this project include using the latest developments in the generative modelling of text and code to enhance this augmentation process. Proprietary LLMs have near-human zero-shot performance on many natural language and code-related tasks which opens up a wide space of possible augmentations (OpenAI).

3.5.3 Hard negatives

T. Chen et al. and Neelakantan et al. both highlight a well documented phenomenon; representations produced by contrastive learning improve with batch size. Although it is not fully understood, it is proposed that a greater number of negative samples in a batch increases the chance that in contains "hard-negative" examples. These are examples that have superficially similar encodings which force the model to learn a richer representation to distinguish them; the inverse challenge of the "hard-positive" examples produced by the data augmentation in section 3.5.2.

Scaling the batch size is limited by the memory available on one's hardware so it can be difficult to realise these benefits. An alternative approach by Xiong et al. suggests that constructing batches that contain these hard negatives improves text representations and X. Li et al. show that this transfers to the semantic code search task.

The final way that we employ synthetic data is with a novel approach that proposes to synthetically generate hard-negative examples. Theoretically the inclusion of these examples would create a more challenging contrastive task that could encourage richer representations.

3.5.4 Paired training

The standard contrastive loss described in section 2.4 assumes only one positive pair per batch. With the data augmentation strategies described in section 3.5.2 and section 3.5.3

it may be necessary to reformulate the task to support multiple positives per batch.

To allow for this we create a real-synthetic "paired" data loader which creates batches that feature both real examples and their respective augmentations. This alone is sufficient for the hard-negative condition[8] because the system will have to distinguish between all of the examples in the batch which is guaranteed to include the adversarially-generated hard-negative. For the hard-positive case, we pose the task as a linear mixture four contrastive objectives; where the original loss only contrasted real docstring-code ($\mathcal{L}_{d,c}$) representations, the new loss contrasts every combination between real examples (d,c) and their augmented equivalents (d',c'):

$$\mathcal{L}_{aug} = \frac{1}{4}(\mathcal{L}_{d,c} + \mathcal{L}_{d,c'} + \mathcal{L}_{d',c} + \mathcal{L}_{d',c'}). \tag{3.3}$$

3.6 Synthetic data generation

The LLM used for all the data generation procedures is OpenAI's `gpt-3.5-turbo`, the model that ChatGPT is built on. This model was chosen because it is available, affordable and known to have a reasonable coding and language skills. Ideally experiments would be repeated with data from GPT-4 as it is known to possess greatly improved coding abilities (OpenAI) but comes at a significantly greater cost. The data are generated by calling the OpenAI API through a Python script built with the LangChain package. For every synthetic data generation strategy we describe below we generate 12,800 examples which are all[9] conditioned on the same set of 12,800 examples from the real training data.

3.6.1 Prompting

All the data generation calls use a system prompt that begins with:

> "You are a helpful python programming assistant."

which serves to provide a framework to model the responses it creates. Each system also ends with:

> "Do not reply with any text other than this {data type}."

where {data type} is replaced with the desired output type (e.g. function/docstring). This is to simplify the task of parsing the response and to reduce costs incurred by superfluous explanations that are characteristic of ChatGPT.

[8] examples generated in this way can actually be included in the training set directly with no adjustment

[9] except for the taxonomically generated dataset which is independent of the original data

Each prompt used in this project has been created through informal experimentation with ChatGPT until the correct behaviour is exhibited. The science of prompting is still young and inexact and is not the focus of this project.

3.6.2 Semi-synthetic data

The first synthetic data generation strategy involves generating synthetic data based on examples from the original training data. To do this we take select one of the modalities (docstring or code) and generate the other modality using the LLM. The prompts to generate a docstring from code (C2D) and code from a docstring (D2C), respectively, are as follows:

> "I will provide a python function and you should write a succinct docstring for that python function."
>
> "I will provide a docstring and you should write a python function that has that docstring."

This procedure creates an idealised form of synthetic data that we term "semi-synthetic data" which side-steps some of the challenges associated with synthetic data generation. By conditioning on real data, we regularise the distribution of the synthetic data to be close to the real distribution. It is also a challenge to maintain the diversity of synthetic data (Eldan and Y. Li) and the real training data provides a necessary source of randomness.

Figure 3.1 features an example of a function generated by both strategies as well as the seed example (fig. 3.1a) that they were generated from. The examples show how the docstring is preserved in the D2C setting and the code is for the C2D setting.

3.6.3 Generative AI augmented data

Inspired by how data augmentation is performed in computer vision (see section 3.5.2), we propose an analogous strategy that produces the augmented examples with an LLM. This is very similar to the semi-synthetic procedure discussed in section 3.6.2 but instead of providing only one modality, the model has access to the full example. Theoretically data generated this way will produce example pairs that encode subtle semantically-invariant code and language transformations that have been learnt through the LLMs extensive training process.

The augmentation operations that we perform are centred around generating two kinds of valuable examples: hard-positives ($H+$) and hard-negatives ($H-$). Hard-

```
def rev_c(read):
    """return reverse completment of read"""
    rc = []
    rc_nucs = {'A':'T', 'T':'A', 'G':'C', 'C':'G', 'N':'N'}
    for base in read:
        rc.extend(rc_nucs[base.upper()])
    return rc[::-1]
```

(a) Seed example

```
def rev_c(read):
    """This function takes a DNA sequence string as input and returns the reverse
    complement of the sequence. The reverse complement is obtained by replacing each
    nucleotide with its complement (A with T, T with A, G with C, C with G) and
    reversing the resulting sequence. The function uses a dictionary to map each
    nucleotide to its complement."""
    rc = []
    rc_nucs = {'A':'T', 'T':'A', 'G':'C', 'C':'G', 'N':'N'}
    for base in read:
        rc.extend(rc_nucs[base.upper()])
    return rc[::-1]
```

(b) C2D semi-synthetic

```
def reverse_complement(read):
    """return reverse completment of read"""
    complement_dict = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    reverse_complement = ''
    for base in read:
        reverse_complement = complement_dict[base] + reverse_complement
    return reverse_complement
```

(c) D2C semi-synthetic

Figure 3.1: Example generations using the semi-synthetic data generation strategies

positive examples are examples that have different surface features but share underlying semantics and hard-negative examples are the inverse. The inclusion of $H+$ examples is similar to the classical image augmentation approach – it encourages the representations to be robust to semantically invariant transformations. As discussed in section 3.5.3, $H-$ examples encourage the representations to be sensitive to subtle semantic differences. The prompts used to generate $H+$ and $H-$ examples, respectively, are below:

> "I will provide a python function and you should heavily refactor it (renaming variables, reordering statements, etc.) into a very different looking but semantically identical python function with a new docstring."
>
> "I will provide a python function and you should write a superficially similar but semantically very different python function and docstring."

The hard-positive augmentation approach is closely related to the manually defined program transformations of Bui et al. but with certain advantages (e.g. realistic transformations and identifier renaming). Generally hard-negative examples are harder to systematically generate although Allamanis et al. and Ding et al. augment code with bug-inducing transformations as hard-negatives.

Figure 3.2 shows an example of the functions produced by these prompts. The hard-positive example in fig. 3.2c demonstrates the LLM's ability to intelligently rename variables and reword docstrings. In fig. 3.2b, the LLM is able to produce a similarly-themed function that serves a different purpose.

3.6.4 Fully-synthetic data

In this section we describe the techniques that we use to generate a fully-synthetic dataset that is independent of any existing data. It may not always be possible to acquire an effective seed dataset for semi-synthetic data generation. In these situations, being able to generate an entire dataset from scratch using only an LLM API would enable the, otherwise impractical, distillation of niche coding abilities.

3.6.4.1 TinyStories and Phi-1

Eldan and Y. Li documents the use of fully synthetic data to train a small, specialised language model. The main problem that they overcome is the lack of diversity in the generated LLM outputs. They do this by enforcing diversity on the generated outputs by prompting the model to generate a story that includes a certain word from a manually assembled list of words and well as including a particular narrative structure element

```
def rev_c(read):
    """return reverse completment of read"""
    rc = []
    rc_nucs = {'A':'T', 'T':'A', 'G':'C', 'C':'G', 'N':'N'}
    for base in read:
        rc.extend(rc_nucs[base.upper()])
    return rc[::-1]
```

(a) Original example

```
def get_complement(sequence):
    """Return the complement of a DNA sequence.
    Args:
        sequence (str): A DNA sequence.
    Returns:
        complement_sequence (str): The complement of the input DNA sequence.
    Raises:
        ValueError: If the input sequence contains invalid characters.
    Examples:
        >>> get_complement('ATCG')
        'TAGC'
        >>> get_complement('GCAT')
        'CGTA'"""
    complement_sequence = ""
    complement_dict = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G'}
    for base in sequence:
        if base in complement_dict:
            complement_sequence += complement_dict[base]
        else:
            raise ValueError(f"Invalid character: {base}")
    return complement_sequence
```

(b) Hard-negative augmented

```
def reverse_complement(read):
    """Returns the reverse complement of the given DNA sequence.
    Args:
        read (str): The DNA sequence to be reverse complemented.
    Returns:
        str: The reverse complement of the given DNA sequence."""
    complement = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G', 'N': 'N'}
    reverse_complement = []
    for base in read:
        reverse_complement.extend(complement[base.upper()])
    return reverse_complement[::-1]
```

(c) Hard-positive augmented

Figure 3.2: Example generations using the LLM data augmentation strategies

(e.g. a plot twist). Gunasekar et al. apply the same principle to generate a dataset resembling a coding textbook and coding exercises.

3.6.4.2 Taxonomical generation

The approach discussed in section 3.6.4.1 still requires a manually-assembled list of topics/vocabulary. We propose to use an LLM to generate this as well in an approach we call "taxonomical generation". The procedure is fairly straightforward; starting with a root concept, we exploit the LLMs understanding of a "sub-concept" to produce a list of topics that can be reasonable considered to be contained within that topic, this process is repeated to recursively generate finer and finer topics. The prompt template used at each stage is as follows:

> "{description}
> Generate a list of {k} subconcepts of '{concept}' and a short description of how each relates to {concept}. Format each list entry as: x. Subconcept: Description."

The description of the current concept (as it related to its parent) is included to disambiguate between topics with similar names and to encourage the generated subconcepts to also fall within the boundaries of the parent concept. A sample of the taxonomy generated in this way can be found in fig. C.1.

This approach could very easily be conducted with different hierarchical categorisation structures. For example one could generate a meronomy of industrial applications of coding (e.g. aviation software $>$ flight computers, flight recorders, in-flight entertainment systems).

Once a taxonomy is available, the concepts in its leaf nodes are used to generate functions that can be used as examples for the contrastive learning task using the following prompt:

```
Concept: {description}
Generate the code and docstring for {k} short python
functions that effectively demonstrate the concept of {concept}.
Each list item should be in the following format:
x. Function:
   Relation to the concept of {concept}:
   Docstring:
   Code:
```

In this setting we generate a list of k examples within the same query to theoretically allow the LLM to avoid generating duplicates. We use this technique to generate $10 \times 10 \times 16$ leaf concepts for which we generate 8 functions for a total 12,800 examples to match our other datasets.

Chapter 4

Implementation challenges

This chapter documents some of the technical challenges that were overcome in the process of experimentation.

4.1 GPU memory

As detailed in section 3.5.3, contrastive learning is sensitive to the batch size used during training. In the process of replicating the results from CodeXGLUE paper, it became apparent that the 12GB of memory on the GPUs[1] we have access to would only support a batch size of 8 – far lower than the recommended batch size of 32. To remedy this we attempt a series of optimisations.

4.1.1 Cross-GPU contrasting

The training script provided by the CodeXGLUE supported the use of multiple GPUs but treated each of these batches as independent, essentially performing multiple iterations of the same batch size rather than simulating a larger batch. To remedy this, we modify the script to calculate the contrastive loss after collecting embeddings across all GPUs, to more closely the behaviour of a large batch size.

T. Chen et al. warn against a pitfall of this type of training when batch normalisation is used. Positive pairs are calculated on the same device which creates a dependence between them and could provide an avenue for the model to cheat on the disambiguation task. For this reason and because we observe slow wall clock times in experiments, we do not use this training strategy for our main performance evaluation experiments.

[1] Nvidia GeForce RTX 2080 Ti

4.1.2 Gradient checkpointing

Gradient checkpointing is procedure proposed by T. Chen et al. that reduces the memory requirements of a machine learning model during training. When using stochastic gradient descent, the gradient of the parameters with respect to the output must be stored for each example in the batch. Instead, gradient checkpointing discards most of these gradient values generated during the forward pass, storing only a subset at key points which it used to recalculate the gradients for the backwards pass. The result is that the training procedure requires significantly less memory at the cost of extra forward computation. In the case of our experiments, it allowed a 4- to 8-fold increase in batch size and is used in all of them.

4.2 Synthetic data

4.2.1 Price

Generating 12,800 semi-synthetic examples using the strategy described in section 3.6.2 cost approximately $4.70 USD for both C2D and D2C. This is a fairly low price, especially compared to the cost of creating this amount of data by paying expert human coders. That said, it is not so low that it would be reasonable to generate 200,000 examples to match the original CSN-Adv training set ($> $70) so we restrict our investigation to this quantity of data. The cost of using GPT-4 for this process would also increase the cost per example 10-fold.

4.2.2 Time

Generating 12,800 examples with sequential calls to the API took a surprising amount of time – approximately 12 hours or 1000 examples per hour. This could have been sped up by maintaining a collection of simultaneous requests but was not deemed necessary for this project.

4.2.3 Retrying

Due to the stochastic nature of generative modelling, the responses produced by the LLM are not always well-formed. In addition, the request to the API would occasionally time out and fail to produce a result. This necessitated error catching infrastructure that identifies failed examples and attempts to regenerate them.

The main source of failure was the inclusion of very long docstrings that caused the model to exceed a specified token limit. When re-generating these examples, the docstrings are truncated to the first few lines which conventionally contains a summary of the function.

4.2.4 Parsing

The CSN-Adv training script expects the training data to feature a tokenised version of the code and docstring. It was a challenge finding the script that the original dataset had been tokenised with in order to process the synthetic data in a consistent way. The correct version and python grammar of Github's Tree-sitter are the versions specified in the original CodeSearchNet repository which we verify by re-tokenising the original dataset.

The data generation strategy described in section 3.6.4.2 encourages the LLM to return data as a list where each list item features several different components (description, relation to super concept, etc.). The textual response from the model is parsed with regular expressions to separate the list items and components from each other.

4.2.5 Semi-synthetic overlap

In the semi-synthetic regime, some of the examples are generated based on counterparts in the real training dataset. These examples are not mutually independent and including both in the same dataset introduces a confounding factor into the experiments that may have unexpected consequences. It also corresponds to using a smaller and less diverse corpus because one of the modalities in each example will be necessarily be repeated. In order to compare results against the size/diversity of equivalent number of real examples, we make modifications to the training script to ensure that seed examples for the semi-synthetic data do not overlap with any included real examples.

One criticism of this approach is that it means that external unimodal data is required to generate the synthetic data, reducing the applicability of the semi-synthetic strategy. This is a fair criticism but it is fairly realistic to have access to an additional unimodal corpus because, in general, it is much cheaper to collect than aligned data – for example, CodeSearchNet features one million bimodal docstring-function Python examples but an additional 4 million unimodal function examples. We ablate the necessity of this process in section 5.1.1.3

Chapter 5

Analysis

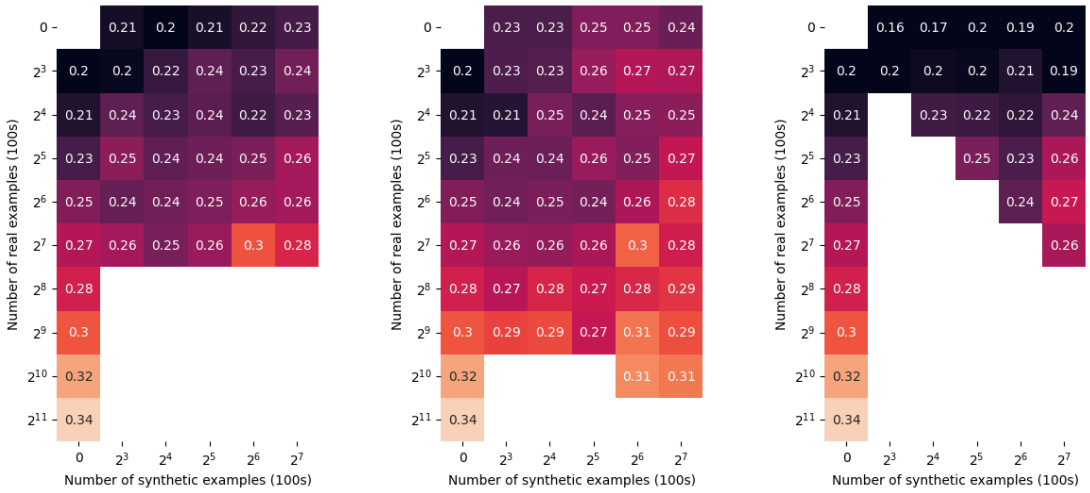
We assess the synthetic data we generate with respect to two attributes: utility and fidelity (Jordon et al.). These are closely related to the concepts of extrinsic and intrinsic evaluation, respectively.

5.1 Extrinsic evaluation: utility

The utility of a synthetic dataset is determined by the extent it can be used to solve a task. In our case, we evaluate its efficacy as training data for learning a joint embedding model of functions and their docstrings for the code search task.

5.1.1 Novel examples

We begin by treating the synthetic data as novel examples as described in section 3.5.1 and use it to supplement real data in various ratios. Figure 5.1 shows the results of experiments using data from three synthetic data generation strategies: D2C/C2D semi-synthetic data and fully synthetic data created with taxonomical generation. All of the heatmaps exhibit vertical gradation corresponding to the expected improvement in performance as the amount of real data increases. There also seems to be a general improvement in performance as the amount of synthetic data increases, most noticeably in fig. 5.1b, but the effect dwindles as the volume of real data overwhelms it. In almost every configuration, adding synthetic data strictly improves the performance of the system with a minimal risk of producing a model that is worse than the baseline, especially when the quantity of synthetic data exceeds the quantity of real data.



(a) D2C: Docstring to code (b) C2D: Code to docstring (c) Taxonomical generation

Figure 5.1: Heatmaps showing how MRR (see section 3.3) changes with respect to the number of real examples and synthetic examples for each data generation strategy. MRR is calculated on the CSN-Adv adversarial test set

5.1.1.1 Unusual effects of synthetic data in comparison to real data

We observe that adding synthetic data has different effects to adding an equivalent quantity of real data which suggests that the synthetic data generation process is not a perfect model of the true data distribution.

A one notable feature of the results is that the runs that do not include real training data (the top row of each heatmap) fail to improve much from additional synthetic data. Regardless of the generation process, the difference between using 800 and 12,800 synthetic examples is less than 0.04 MRR – significantly less than the 0.07 improvement observed from an equivalent increase in real training data. This suggests that the synthetic data is lacking in some aspect of real data which we explore in section 5.2.2.3.

Interestingly, the benefit of including additional synthetic data seems to be somewhat orthogonal to the benefit of including additional real data. Every strategy exhibits a roughly fixed benefit to MRR from including 12,800 additional synthetic examples until it is matched by the quantity of real data.

5.1.1.2 Comparison of novel data generation strategies

The performance of systems trained with semi-synthetic data (fig. 5.1a and fig. 5.1a) seems to be superior to that of systems trained with fully synthetic, taxonomically-

generated data (fig. 5.1c). This is an expected result given that it is regularised to follow a real data distribution instead of having to generate examples from scratch.

C2D semi-synthetic data also seems to have marginally better performance than D2D although this is largely due to the high performance in experiments with 800 real examples. We attribute the behaviour to an anomaly that would vanish given the scope for repeated experiments. That said, it may also be an indication that modelling exclusively C2D semi-synthetic data serves a conflicting objective to modelling both C2D data and real data combined. This strategy also displays unusually good performance when mixed 50/50 with real data, sometimes beating systems trained with the same amount of real data. When the amount of real data, this suggests that replacing half of it with LLM synthesised data may provide more value to the system than simply using the real data directly.

5.1.1.3 Dataset overlap ablation

In section 4.2.5, we discuss the choice to separate the semi-synthetic seed data from the real data used during training. We investigate the importance of this by performing ablation experiments where the real data used in training fully overlaps with the seed data used to generate the examples – the results are shown in fig. 5.2. C2D (left) exhibits the expected decline in performance from having access to fewer real code examples. The D2C case features anomalously good performance that we attribute to peculiarities of the D2C dataset that we discuss in section 5.2.1 (some of the examples erroneously feature modified docstrings). Overall it appears as though using overlapping data in a data augmentation approach still improves performance to a minor degree but we expect seeding with unimodal data to be a better strategy.

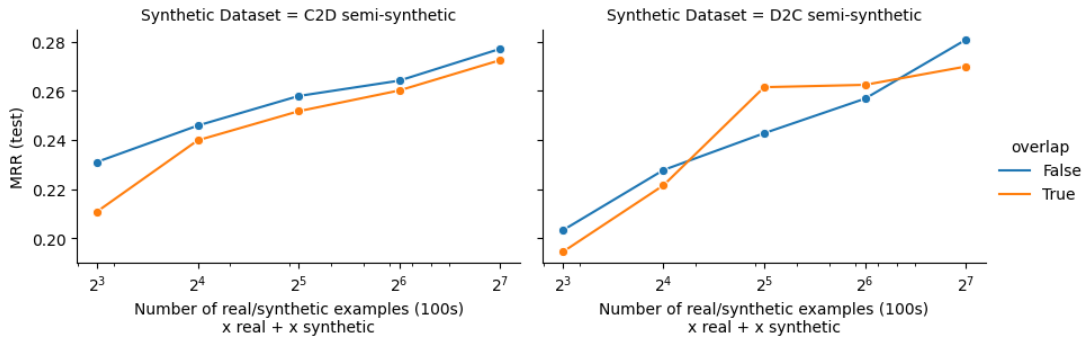
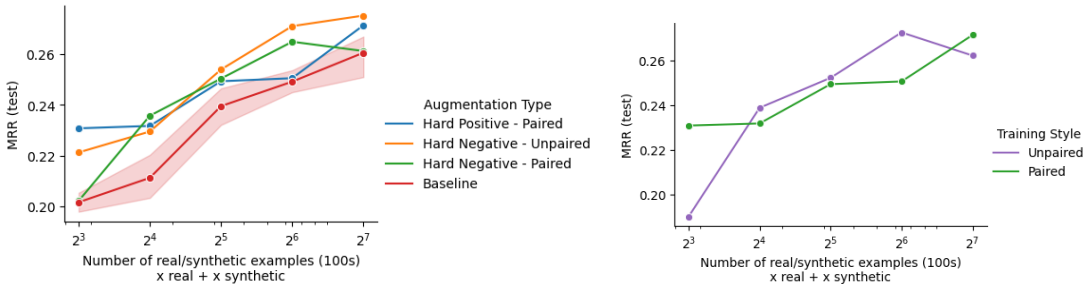


Figure 5.2: Graph showing the relationship between MRR on the CSN-Adv test set split by whether an overlapping portion of the dataset was used to generate examples.

5.1.2 Data augmentation

We also perform experiments to evaluate the effect of LLM-based data augmentation strategies (both hard-positives and hard-negatives). As each augmented example is generated from a corresponding real example, we restrict our analysis to datasets that feature an equal number of synthetic and real training examples. Figure 5.3a shows that all of the data augmentation procedures seem to improve upon a baseline without data augmentation, in most cases even outperforming real datasets of equivalent size. This is a strong indicator of the value of this procedure and reflects findings in the image domain literature on the value of data augmentation for contrastive learning (T. Chen et al.).



(a) Augmentations vs. baseline

(b) $H+$ paired training ablation

Figure 5.3: Graph showing the relationship between MRR on the CSN-Adv test set for each data augmentation strategy. Each x value indicates the number of real examples and synthetic examples for $2x$ overall (apart from the baseline strategy that only features x real examples)

5.1.2.1 Ablation of paired training

The orange and green lines in fig. 5.3a show that the paired training that includes hard-negative synthetic examples in each batch performs similarly or worse than simply adding augmented hard-negatives to the dataset (like the experiments in section 5.1.1). This could mean that the main performance improvement of including these hard-negative augmentations comes from them beinghigh quality additional examples rather than directly serving as a hard-negatives for specific examples.

Hard-positive examples must be treated differently. Including positive examples in the same way as novel examples would sometimes result in batches where positive pairs are erroneously sampled as in-batch negatives. However, this can also occasionally

occur in real datasets so we experiment with this way of including hard-positives anyway. Figure 5.3b shows the unexpected result that this form of augmentation is competitive with the paired training and also surpasses the baseline.

We propose several explanations for the good performance of this configuration. Firstly, the augmentations generated by `gpt-3.5-turbo` may not be hard-positives and instead may consist of examples that are semantically distinct from the data that they augment. Alternatively, the diversity from the augmentation might be outweighing the negative impact of occasionally sampling positive pairs. Finally, each component of loss function that we propose in section 3.5.4 must only distinguish between batches of examples that are half the size of those used in regular training. Due to the known sensitivity to batch size of contrastive learning (discussed in section 3.5.2), the paired training only having to distinguish between smaller batches may be hurting its performance.

5.2 Intrinsic evaluation: Fidelity

Another way that synthetic data can be evaluated is on its fidelity – how well it matches the distribution of real data. Like many intrinsic evaluation methods, it is only as valuable as its ability to predict the utility of the synthetic data. Indeed, if the synthetic data is of a higher quality than the real data is, it may improve the performance of the system while necessarily diverging from the distribution of the training data.

5.2.1 Token statistics

A basic way to examine evaluate differences in sequence data is to compare token statistics between the corpora. To begin we plot a histogram of the length of examples in each corpus, split by data type (fig. 5.4).

From this it is immediately clear that the distributions of the semi-synthetic data are very different from the baseline. The docstrings generated by the C2D strategy are notably longer than those in real data. From looking at examples in the data such as fig. 3.1b, `gpt-3.5-turbo` seems to produce very verbose descriptions of all the parts of the function, even describing the objects and the control flow of the code. While this makes for unrealistic docstrings, having this type of description could be beneficial for a model learning the relationship between natural language and code. The functions generated by the D2C and taxonomical strategies are unusually short which could make it hard to generalise to longer test examples, explaining some of the worse performance

of these two systems.

Finally, the fact that the distributions in the data augmentation cases match so effectively suggest that the model understands that it is not supposed to modify the examples much; this shows that it is usually capable of copying the form and structure of examples that it is provided. This suggests that our prompting generations might be improved if we provided some full examples in the generation prompt for the LLM to base its outputs on.

5.2.1.1 Common tokens

We also investigate the most common tokens of each dataset – the top ten code[1] tokens for each dataset are shown in table 5.1. Both augmented approaches show good agreement with the real data which suggests the process successfully maintains the structure of most examples. The D2C and Taxonomical strategies seem to generate many fewer class methods (missing 'self' keyword) than are present in real data which would also affect downstream performance on this common type of function.

Most worryingly, we investigate the prevalence of "pass" token generations in the D2C semi-synthetic data and find that many correspond to function stubs where the LLM has only produced a function signature. We discuss this in section 5.3.1.

Baseline	**C2D**	**D2C**	**H-**	**H+**	**Taxonomical**
(102487	(102487	(21245	) 84324	) 98464	(32371
) 102487	) 102487	) 21245	(84320	(98463	) 32369
. 94435	. 94435	: 19504	. 74135	. 91593	: 26879
, 74499	, 74499	def 12987	, 61241	, 72564	, 14814
= 65095	= 65095	, 12124	= 53001	= 66590	def 13195
: 57264	: 57264	= 7766	: 50140	: 56901	= 12579
self 35139	self 35139	pass 5763	self 29969	self 33908	return 11789
[27542	[27542	. 5591	[22293	[26381	[9798
] 27542	] 27542	return 4223	] 22293	] 26381	] 9798
if 19209	if 19209	] 2831	if 15789	if 18796	. 8654

Table 5.1: Top ten most common code tokens and counts for each dataset

[1] we have also looked at the most common docstring tokens but it is much less interpretable/interesting

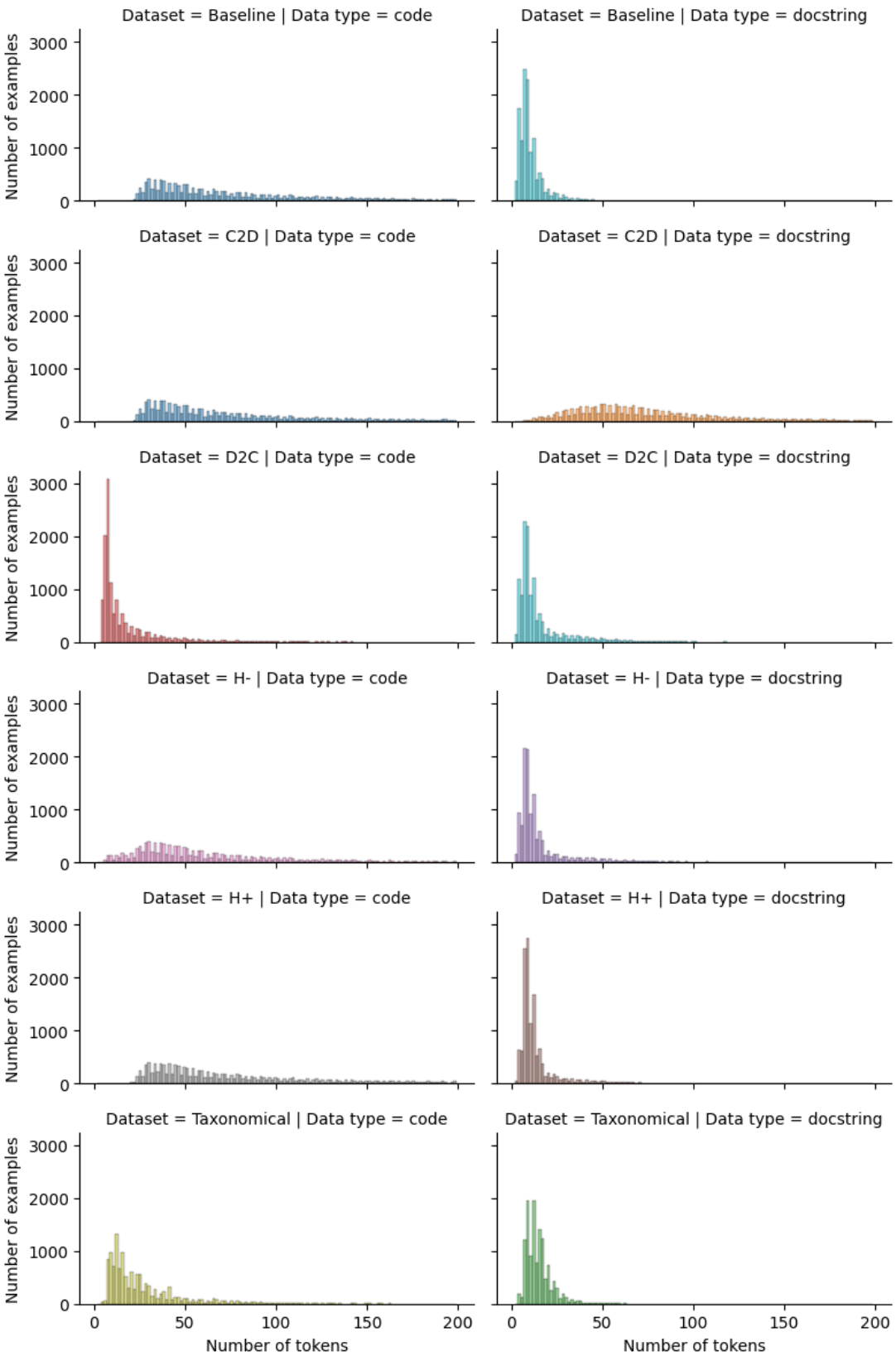


Figure 5.4: Histogram of code and docstring token lengths for each data generation strategy. Token lengths above 200 are hidden.

5.2.2 Embedding space analysis

Comparing basic statistics for the datasets only provides a crude estimate of close they are to the true distribution. In order to facilitate a more in-depth analysis, we take OpenAI's general purpose embedding model, `text-embedding-ada-002`, as ground truth representations for our data. This model is extremely affordable at $0.0001 per thousand tokens; embedding $8 \times 12,800$ examples cost less than $5 in total. One caveat to using an embedding model from OpenAI is that it is possible that the embedding model was initialised from a similar language model to `gpt-3.5-turbo`. This might cause it overrate the diversity of generations from `gpt-3.5-turbo` if they have a similar understanding of the breadth of language/code.

5.2.2.1 Quality check

To ensure that the `text-embedding-ada-002` produces high-quality representations for code search data, we evaluate its performance on the CSN-Adv semantic code search task. The model achieves an MRR of 0.445, significantly higher than our best CodeBERT model trained for this task.

5.2.2.2 Corpus distance metrics

While embedding spaces encode a notion of similarity between individual examples, it is not trivial to compare distances between collections of embeddings. Kour et al. evaluate a number of different corpus distance metrics and release a framework for easily computing them. The mathematical details of the different metrics are not the focus of this work so we merely provide a brief overview and direct the reader to the aforementioned evaluation paper. Classifier distance is the accuracy of a linear classifier trained to distinguish the corpora. Fréchet inception distance (FID) is originally from the field of computer vision but has been extended to mean the 2D Wasserstein distance between Gaussian distributions fit on the embedded corpora. Precision, recall, density and coverage estimate a manifold using the nearest neighbours of each example and then measure whether individual points lie inside it[2].

Figure 5.5 shows the ordering of the datasets that all the metrics largely agree upon. The distances from the training dataset are greatest for taxonomically generated data followed by semi-synthetic and then data augmentation approaches. This reflects our empirical observations in section 5.1 that the semi-synthetic data has greater utility. The

[2]sample level distances like this can be used for rejection sampling of synthetic data (Alaa et al.)

graph also highlights an unexpected discrepancy between the D2C docstring dataset and the baseline. This means that the docstrings featured in that dataset are not exactly identical to those in the baseline (see section 5.3.1 for discussion).

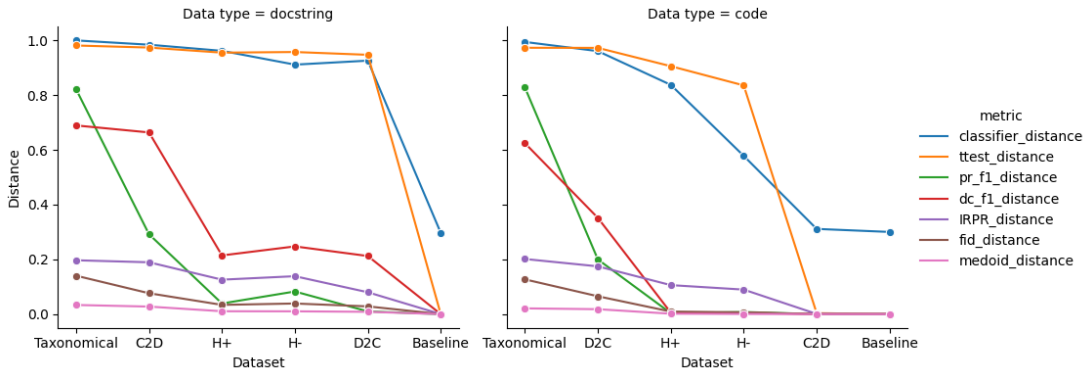


Figure 5.5: Plot of corpus distance metrics from the real training dataset showing most metrics agree upon an ordering that predicts the efficacy of that dataset generation strategy.

5.2.2.3 PCA Visualisation

To visualise the distribution of each synthetic dataset we perform principle component analysis (PCA) on the vectors of the original training set. PCA solves for a hyperplane that, when the data points are projected onto it, minimises a sum of squares reconstruction error. We learn this projection on the training set in the hopes that the dimensions will represent the principal components of the true data distribution. We choose a two dimensional projection for clear visualisation.

The results of this can be seen in fig. 5.6. This graph provides compelling evidence for the relative performances of each synthetic dataset. The fully synthetic data appears to occupy a tighter cluster than the semi-synthetic data does which itself occupies a tighter cluster than the original training dataset. A tighter clustering could imply that the dataset lacks diversity, translating to representations that neglect the full breadth of the domain. Interestingly, the natural language cluster for the C2D synthetic data appears relatively diffuse which provides an explanation for its superior performance on the code search task. It also appears to be closer to the embeddings of code, reflecting the tendency of `gpt-3.5-turbo` to over-articulate the exact behaviour of the function (such as describing that the code uses a dictionary in fig. 3.1b). Extensive graphs from this PCA process are included in Figure A.1.

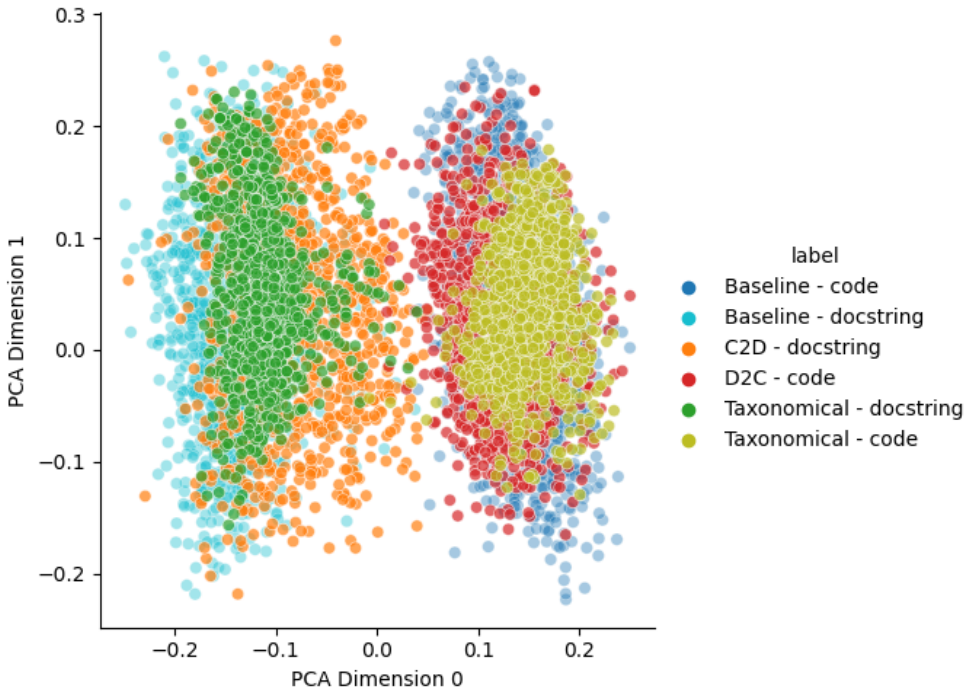


Figure 5.6: Different synthetic datasets embedded with `text-embedding-ada-002` and transformed using a PCA projection fit on the real training dataset

5.3 Limitations

In this section we review some of the limitations of the experiments conducted in this project.

5.3.1 D2C issues

Through our analysis, we uncover a couple of issues with our D2C generation process. Namely, there are a large number of functions generated that feature only a single "pass" token. This behaviour should be fairly easy to work around by detecting and re-sampling these examples however we observe that it occurs disproportionately often with complicated requests. This means that it may be triggered by the model having a high uncertainty about how to generate the requested function and may occur less with a more powerful LLM. Another issue is that we trust the LLM to perfectly copy the docstring it is provided into the function body. However, we notice that it frequently modifies the docstring to elaborate or correct spelling mistakes. This can also be resolved with a simple filtering step or using a regular expression to modify the function to include the original docstring.

These issues mean that our reported D2C results underestimate its potential. The proposed fixes are relatively simple but rerunning the generation and experiments is outside the temporal and financial scope of this project. That said, given all the complications and challenges of getting LLMs to generate diverse code, we believe that without a more powerful LLM it would be more effective to pursue C2D generation or data augmentation approaches.

5.3.2 Dataset size

Given the cost of generating extremely large synthetic datasets, we only conduct experiments with a small number of examples. In addition, we show that the benefit of including synthetic data is limited to when there is some real data available but significantly less than the amount of synthetic data available. We also expect synthetic datasets generated by LLMs to have diminishing value (relative to real datasets) as they include more and more examples but we leave this investigation to future, well-resourced research. These factors restrict the direct relevance of this work to low-resourced coding tasks.

5.3.3 Repeats

We do not perform a frequentist hypothesis testing analysis on our findings because resource constraints make it impractical to perform many repeats of the same experiment. Instead, we opt to explore a wider variety of strategies as well as perform ablation experiments. We perform repeats assessing the performance of the baseline which allow us to claim that the performance of each individual training run is significantly above the baseline but we cannot make a claim about the procedure helping on average. Given that almost every synthetically augmented run performs significantly better than the baseline we believe it is fair to claim that it is a helpful procedure.

5.3.4 Proxy task

We conduct experiments exclusively on the CSN-Adv in the hopes that the findings will translate to other domains. The most obvious domain to apply these techniques to is naturalistic code search tasks such as CoSQA (Huang et al.). Compared to the docstring-function pairs in CodeSearchNet, it is hard to collect real user code queries which makes the techniques for adapting monolingual code data even more relevant.

The techniques presented in this project do not target code search specifically so we would expect them to transfer to other code intelligence tasks. However this cannot be said with confidence before they are tested empirically.

5.3.5 Python

Another limitation on the scope of this project is that we only work with Python code. Python is one of the most popular programming languages in the world and `gpt-3.5-turbo` is likely to have encountered a disproportionate amount of Python code in its pretraining process. Zhou et al. suggest that the pretraining process is where these models learn almost all of their knowledge; having a greater exposure during pretraining is very likely to translate to better understanding and generation capabilities. In addition, the CodeBERT model that is used for contrastive learning has been explicitly pretrained on six languages, including over one million Python functions.

From these two factors, it is clear that the procedures described in this project have an advantage when using Python data. As such, it is not clear that the findings will transfer to less popular languages that teacher and student models are less familiar with. As seen in section 5.3.2, the value of this technique is greatest when bimodal training data is limited – a situation heavily associated with less popular languages. If the success of this process is dependent on the popularity of the language during pretraining, this significantly reduces its applicability.

5.3.6 Hyperparameter tuning

In the interest of practicality, we largely neglect the tuning of hyperparameters. Training is conducted with the baseline hyperparameters selected by the CodeXGLUE team for the task. A consequence of this is that the results reported on the downstream task may be specific to this set of hyperparameters. Alternatively, the benefits of including synthetic data may not hold on fully tuned systems.

5.3.6.1 Early stopping with learning rate decay

Since their invention by Vaswani et al., it is standard practice to train transformers with a decaying learning rate schedule and a linear warm up. The issue with this is that it requires an estimate of the total length of the training – something that is unavailable

when unpredictable experiments are being done for the first time and terminated with early stopping. As such we leave these parameters untouched for our experiments but it is likely that it would be more optimal to reduce the warm up and decay period. This is a special case of the hyperparameter issue described above that would be solved with proper tuning.

5.3.7 ChatGPT vs. true generative models

Although `gpt-3.5-turbo` was originally trained with a language modelling objective, it is then fine-tuned to follow instructions as well as to be aligned to human preferences with RLHF. Both of these procedures manipulate its underlying model of language to make it easier to use but in the process the model becomes less representative of the full diversity of the distribution (i.e. it disproportionately produces high quality outputs when the real world contains many low quality ones) (Bai et al.). For coding tasks where the model needs to produce a piece of code that solves a problem, being restricted to producing a single (high quality) style of code can be an advantage (as seen in Gunasekar et al.). However, with a task such as representation learning for retrieving real code examples, the examples generated by the model will tend to lie in a limited region of the future data distribution. Sampling synthetic data from a purely generative model might be a way to improve the diversity of the data.

5.3.8 LLM pretraining dataset

The LLMs trained by OpenAI are so large that training them requires vast amounts of data. Gathering that scale of data must be done with automated processes that cannot be fully supervised. Models like `gpt-3.5-turbo` and GPT-4 are trained on so much data that it is fairly likely that they have seen the train and test set of CSN-Adv during their pre-training process. If the training set was seen, the semi-synthetic generation strategies could be unrealistically close to the true data. If the test set was seen, some of the synthetic data that was generated could include features of the test set that would give it an unfair advantage. Realistically, this is likely to be a minor effect but we perform a basic check through all the generated datasets and find no exact[3] copies of any of the test set examples.

Finally, the pretraining dataset is likely to feature undesirable instances of bias and harmful behaviour. While the RLHF tuning process aims to remove most of this, there

[3] we do not check for noisy copies that could still confound the results

is no guarantee that it can be totally eradicated. Training on the outputs of an LLM is liable to reproduce any biases that it has developed which can propagate them further into the world.

Chapter 6

Conclusions

Through this investigation we find that using LLMs to generate synthetic data for training smaller models of code has significant potential. We propose and evaluate several synthetic data generation strategies and find that our synthetic data augmentation approaches produce data that can be more effective for learning representations than real data.

These data generation strategies correspond to situations with varying availability of existing data. In the case where unimodal data is available we demonstrate the value of semi-synthetic data, especially generating docstrings from code. When only a small bimodal dataset is available we demonstrate data augmentation approaches that maximise its value. For situations where no source of data is available we demonstrate how to use an LLM to generate a dataset entirely from scratch to create a system with modest performance.

Finally, we show a variety of ways to analyse the data generated through these processes which we hope can be used to further refine the prompts, hyperparameters and LLMs that affect the quality of the synthetic data.

6.1 Future work

One of the most direct extensions to this work would be to experiment with few-shot approaches. Including a few examples sampled from the real dataset might allow the LLM to more accurately reflect the style and distribution of real data (as it does in the data augmentation strategies).

A valuable direction for future work is exploring different ways to filter the generated dataset for quality. This is usually an important part of a synthetic data generation

process that can provide a training signal capable of improving a model beyond the abilities of the data generating model (Schick and Schütze). For example, Zelikman et al. filter synthetically generated reasoning chains-of-thought by whether they produce a correct answer. The coding domain is especially suitable for these filtering procedures because code can be run and there is a large literature on analysing abstract syntax trees from the compiler field. Haluptzok et al. explore an idea like this where they recursively train a program synthesis model to generate programming puzzles and solutions that can be verified by running the code.

Another direction that we believe is promising is synthetically generating dataset during the training process. Humans do not learn everything in a single extended training process with no guidance – generally, teachers attempt to help students by identifying weaknesses and flaws in the student's understanding throughout the learning process. Having powerful general purpose AI systems that are available to provide feedback at any time is a significant advantage compared to supervision from humans. This means that it may be feasible to design training examples for a student model that are specialised for its current failings. This is connected to both active learning and human-in-the-loop training systems and we believe that today's LLMs make these ideas possible.

Works Cited

Alaa, Ahmed M, et al. “How Faithful Is Your Synthetic Data? Sample-level Metrics for Evaluating and Auditing Generative Models”.

Allamanis, Miltiadis, et al. “Self-Supervised Bug Detection and Repair”. *Advances in Neural Information Processing Systems*. Curran Associates, 2021, pp. 27865–76, proceedings.neurips.cc/paper/2021/hash/ea96efc03b9a050d895110db8c4af057-Abstract.html. Accessed 14 Aug. 2023.

Bai, Yuntao, et al. “Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback”, 12 Apr. 2022, arxiv.org/abs/2204.05862v1. Accessed 15 Aug. 2023.

Bajaj, Payal, et al. “MS MARCO: A Human Generated MAchine Reading COmprehension Dataset”. *arXiv*, 31 Oct. 2018, arxiv.org/abs/1611.09268. Accessed 11 Aug. 2023.

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2006. Information Science and Statistics.

Brown, Tom B., et al. “Language Models Are Few-Shot Learners”. *arXiv*, arxiv.org/abs/2005.14165, 22 July 2020, https://doi.org/10.48550/arXiv.2005.14165.

Bui, Nghi D. Q., et al. “Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations”. *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’21, Association for Computing Machinery, 11 July 2021, pp. 511–21, https://doi.org/10.1145/3404835.3462840.

Cegin, Jan, et al. “ChatGPT to Replace Crowdsourcing of Paraphrases for Intent Classification: Higher Diversity and Comparable Model Robustness”. *arXiv*, 22 May 2023, arxiv.org/abs/2305.12947. Accessed 22 Aug. 2023.

Chen, Mark, et al. “Evaluating Large Language Models Trained on Code”. *arXiv*, arxiv.org/abs/2107.03374, 14 July 2021, https://doi.org/10.48550/arXiv.2107.03374.

Chen, Tianqi, et al. “Training Deep Nets with Sublinear Memory Cost”. *arXiv*, 22 Apr. 2016, arxiv.org/abs/1604.06174. Accessed 14 Aug. 2023.

Chen, Ting, et al. “A Simple Framework for Contrastive Learning of Visual Representations”. *Proceedings of the 37th International Conference on Machine Learning*. Proc. of International Conference on Machine Learning, PMLR, 21 Nov. 2020, pp. 1597–607, proceedings.mlr.press/v119/chen20j.html. Accessed 13 Aug. 2023.

Craswell, Nick, et al. “OVERVIEW OF THE TREC 2019 DEEP LEARNING TRACK”.

Craswell, Nick, et al. “OVERVIEW OF THE TREC 2022 DEEP LEARNING TRACK”.

Devlin, Jacob, et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. *arXiv*, arxiv . org / abs / 1810 . 04805, 24 May 2019, https://doi.org/10.48550/arXiv.1810.04805.

Ding, Yangruibo, et al. “Contrastive Learning for Source Code with Structural and Functional Properties”. 6 Oct. 2021. openreview.net/forum?id=7KgeqhkbZab. Accessed 14 Aug. 2023.

Eldan, Ronen, and Yuanzhi Li. “TinyStories: How Small Can Language Models Be and Still Speak Coherent English?” *arXiv*, 24 May 2023, arxiv.org/abs/2305.07759. Accessed 28 May 2023.

Feng, Zhangyin, et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. *arXiv*, 18 Sept. 2020, arxiv.org/abs/2002.08155. Accessed 22 Mar. 2023.

Gao, Luyu, et al. “PAL: Program-aided Language Models”. *arXiv*, arxiv.org/abs/2211.10435, 27 Jan. 2023, https://doi.org/10.48550/arXiv.2211.10435.

Goodfellow, Ian, et al. “Generative Adversarial Nets”. *Advances in Neural Information Processing Systems*. Curran Associates, 2014, proceedings.neurips.cc/paper_files/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html. Accessed 14 Aug. 2023.

Gunasekar, Suriya, et al. “Textbooks Are All You Need”. *arXiv*, 20 June 2023, arxiv.org/abs/2306.11644. Accessed 26 June 2023.

Guo, Daya, et al. “GraphCodeBERT: Pre-training Code Representations with Data Flow”. *arXiv*, arxiv.org/abs/2009.08366, 13 Sept. 2021, https://doi.org/10.48550/arXiv.2009.08366.

Guo, Daya, et al. “UniXcoder: Unified Cross-Modal Pre-training for Code Representation”. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Proc. of ACL 2022, Association for Computa-

tional Linguistics, May 2022, pp. 7212–25, https://doi.org/10.18653/v1/2022.acl-long.499.

Haluptzok, Patrick, et al. "Language Models Can Teach Themselves to Program Better". *arXiv*, arxiv.org/abs/2207.14502, 12 Apr. 2023, https://doi.org/10.48550/arXiv.2207.14502.

Huang, Junjie, et al. "CoSQA: 20,000+ Web Queries for Code Search and Question Answering". *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Proc. of ACL-IJCNLP 2021, Association for Computational Linguistics, Aug. 2021, pp. 5690–700, https://doi.org/10.18653/v1/2021.acl-long.442.

Husain, Hamel, et al. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". *arXiv*, arxiv.org/abs/1909.09436, 8 June 2020, https://doi.org/10.48550/arXiv.1909.09436.

Jain, Paras, et al. "Contrastive Code Representation Learning". *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Proc. of EMNLP 2021, Association for Computational Linguistics, Nov. 2021, pp. 5954–71, https://doi.org/10.18653/v1/2021.emnlp-main.482.

Jordon, James, et al. "Synthetic Data – What, Why and How?" *arXiv*, arxiv.org/abs/2205.03257, 6 May 2022, https://doi.org/10.48550/arXiv.2205.03257.

Karpukhin, Vladimir, et al. "Dense Passage Retrieval for Open-Domain Question Answering". *arXiv*, arxiv.org/abs/2004.04906, 30 Sept. 2020, https://doi.org/10.48550/arXiv.2004.04906.

Kour, George, et al. "Measuring the Measuring Tools: An Automatic Evaluation of Semantic Metrics for Text Corpora". *Proceedings of the 2nd Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*. Proc. of GEM 2022, Association for Computational Linguistics, Dec. 2022, pp. 405–16, https://doi.org/10.18653/v1/2022.gem-1.35.

Lecun, Y., et al. "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE*, vol. 86, no. 11, Nov. 1998, pp. 2278–324. https://doi.org/10.1109/5.726791.

Lewis, Patrick, et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". *Advances in Neural Information Processing Systems*. Curran Associates, 2020, pp. 9459–74, proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html. Accessed 12 Aug. 2023.

Li, Bohan, et al. “On the Sentence Embeddings from Pre-trained Language Models”. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Proc. of EMNLP 2020, Association for Computational Linguistics, Nov. 2020, pp. 9119–30, https://doi.org/10.18653/v1/2020.emnlp-main.733.

Li, Xiaonan, et al. “CodeRetriever: A Large Scale Contrastive Pre-Training Method for Code Search”. *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Proc. of EMNLP 2022, Association for Computational Linguistics, Dec. 2022, pp. 2898–910, aclanthology.org/2022.emnlp-main.187. Accessed 23 June 2023.

Liu, Yinhan, et al. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. *arXiv*, 26 July 2019, arxiv.org/abs/1907.11692. Accessed 6 May 2023.

Lu, Shuai, et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation”. *arXiv*, arxiv.org/abs/2102.04664, 16 Mar. 2021, https://doi.org/10.48550/arXiv.2102.04664.

Lu, Shuai, et al. “ReACC: A Retrieval-Augmented Code Completion Framework”. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Proc. of ACL 2022, Association for Computational Linguistics, May 2022, pp. 6227–40, https://doi.org/10.18653/v1/2022.acl-long.431.

Luo, Ziyang, et al. “WizardCoder: Empowering Code Large Language Models with Evol-Instruct”. *arXiv*, 14 June 2023, arxiv.org/abs/2306.08568. Accessed 14 July 2023.

Madaan, Aman, et al. “Language Models of Code Are Few-Shot Commonsense Learners”. *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Proc. of EMNLP 2022, Association for Computational Linguistics, Dec. 2022, pp. 1384–403, https://doi.org/10.18653/v1/2022.emnlp-main.90.

Magister, Lucie Charlotte, et al. “Teaching Small Language Models to Reason”. *arXiv*, 19 Dec. 2022, arxiv.org/abs/2212.08410. Accessed 16 Apr. 2023.

Mikolov, Tomas, et al. “Efficient Estimation of Word Representations in Vector Space”. *arXiv*, arxiv.org/abs/1301.3781, 6 Sept. 2013, https://doi.org/10.48550/arXiv.1301.3781.

Mukherjee, Subhabrata, et al. “Orca: Progressive Learning from Complex Explanation Traces of GPT-4”. *arXiv*, arxiv.org/abs/2306.02707, 5 June 2023, https://doi.org/10.48550/arXiv.2306.02707.

Neelakantan, Arvind, et al. “Text and Code Embeddings by Contrastive Pre-Training”. *arXiv*, 24 Jan. 2022, arxiv.org/abs/2201.10005. Accessed 13 Apr. 2023.

Nikolenko, Sergey I. *Synthetic Data for Deep Learning*. Vol. 174, Springer International Publishing, 2021, https://doi.org/10.1007/978-3-030-75178-4. Springer Optimization and Its Applications.

OpenAI. “GPT-4 Technical Report”. *arXiv*, arxiv.org/abs/2303.08774, 27 Mar. 2023, https://doi.org/10.48550/arXiv.2303.08774.

Park, Shinwoo, et al. “Contrastive Learning with Keyword-based Data Augmentation for Code Search and Code Question Answering”. *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. Proc. of EACL 2023, Association for Computational Linguistics, May 2023, pp. 3609–19, aclanthology.org/2023.eacl-main.262. Accessed 10 June 2023.

Raffel, Colin, et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. *arXiv*, arxiv.org/abs/1910.10683, 28 July 2020, https://doi.org/10.48550/arXiv.1910.10683.

Schick, Timo, and Hinrich Schütze. “Generating Datasets with Pretrained Language Models”. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Proc. of EMNLP 2021, Association for Computational Linguistics, Nov. 2021, pp. 6943–51, https://doi.org/10.18653/v1/2021.emnlp-main.555.

Sennrich, Rico, et al. “Improving Neural Machine Translation Models with Monolingual Data”. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Proc. of ACL 2016, Association for Computational Linguistics, Aug. 2016, pp. 86–96, https://doi.org/10.18653/v1/P16-1009.

Shridhar, Kumar, et al. “Distilling Multi-Step Reasoning Capabilities of Large Language Models into Smaller Models via Semantic Decompositions”. *arXiv*, 30 Nov. 2022, arxiv.org/abs/2212.00193. Accessed 17 Apr. 2023.

Shumailov, Ilia, et al. “The Curse of Recursion: Training on Generated Data Makes Models Forget”. *arXiv*, arxiv.org/abs/2305.17493, 31 May 2023, https://doi.org/10.48550/arXiv.2305.17493.

Stanford Alpaca: An Instruction-following LLaMA Model. 10 Mar. 2023, 12 Apr. 2023, github.com/tatsu-lab/stanford_alpaca. Accessed 12 Apr. 2023.

Sun, Simeng, et al. “Do Long-Range Language Models Actually Use Long-Range Context?” *Proceedings of the 2021 Conference on Empirical Methods in Natural*

Language Processing. Proc. of EMNLP 2021, Association for Computational Linguistics, Nov. 2021, pp. 807–22, https://doi.org/10.18653/v1/2021.emnlp-main.62.

Vaswani, Ashish, et al. "Attention Is All You Need". 12 June 2017. https://doi.org/10.48550/arXiv.1706.03762.

Villalobos, Pablo, et al. "Will We Run out of Data? An Analysis of the Limits of Scaling Datasets in Machine Learning". *arXiv*, 25 Oct. 2022, arxiv.org/abs/2211.04325. Accessed 13 Aug. 2023.

Wang, Yue, et al. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". *arXiv*, 13 May 2023, arxiv.org/abs/2305.07922. Accessed 18 May 2023.

Xiong, Lee, et al. "Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval". *arXiv*, 20 Oct. 2020, arxiv.org/abs/2007.00808. Accessed 13 July 2023.

Xu, Can, et al. "WizardLM: Empowering Large Language Models to Follow Complex Instructions". *arXiv*, arxiv.org/abs/2304.12244, 10 June 2023, https://doi.org/10.48550/arXiv.2304.12244.

Zelikman, Eric, et al. "STaR: Bootstrapping Reasoning With Reasoning". *arXiv*, 20 May 2022, arxiv.org/abs/2203.14465. Accessed 16 Apr. 2023.

Zhou, Chunting, et al. "LIMA: Less Is More for Alignment". *arXiv*, arxiv.org/abs/2305.11206, 18 May 2023, https://doi.org/10.48550/arXiv.2305.11206.

Appendix A

Figures

A.1 PCA

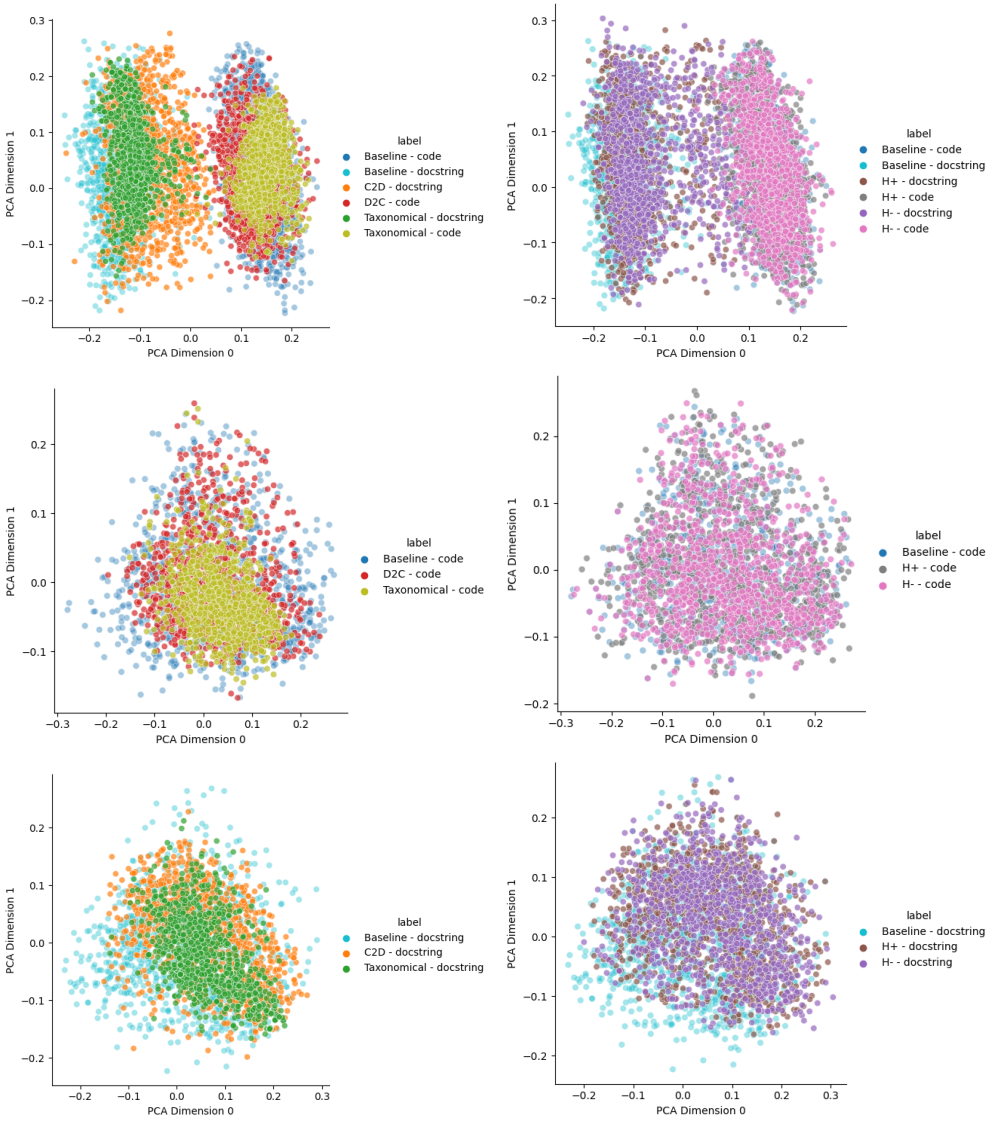


Figure A.1: PCA projections of augmentations as well as datasets in code/docstring exclusive principal components

Appendix B

Examples

B.1 Model disagreement examples

```
    ======== 182
def Func(arg_0: arg_1, arg_2: arg_1, **arg_3) -> Callable[[arg_6[arg_1], arg_8, arg_8], pd.Data
    """Build a function that handles downloading tabular data and parsing it into a pandas Data

    :param data_url: The URL of the data
    :param data_path: The path where the data should get stored
    :param kwargs: Any other arguments to pass to :func:`pandas.read_csv`
    """
    arg_4 = make_downloader(arg_0, arg_2)

    def get_df(arg_5: arg_6[arg_1] = None, arg_7: arg_8 = True, arg_9: arg_8 = False) -> pd.Dat
        """Get the data as a pandas DataFrame.

        :param url: The URL (or file path) to download.
        :param cache: If true, the data is downloaded to the file system, else it is loaded fro
        :param force_download: If true, overwrites a previously cached file
        """
        if arg_5 is None and arg_7:
            arg_5 = arg_4(arg_9=arg_9)

        return pd.read_csv(
            arg_5 or arg_0,
            **arg_3
        )

    return get_df
```

```
======== 260
def Func(arg_0):
        """Represent this dependency as a dict. For json compatibility."""
        return dict(
            dependencies=list(arg_0),
            all=arg_0.all,
            success=arg_0.success,
            failure=arg_0.failure
        )
======== 8785
def Func(arg_0, arg_1=False, arg_2=True,
                            arg_3=False, arg_4=None, arg_5=1, arg_6='./',
                            arg_7=arg_8,
                            arg_9=2**19, arg_10='times', arg_11=True):
        """Simulate Brownian motion trajectories and emission rates.

        This method performs the Brownian motion simulation using the current
        set of parameters. Before running this method you can check the
        disk-space requirements using :method:`print_sizes`.

        Results are stored to disk in HDF5 format and are accessible in
        in `self.emission`, `self.emission_tot` and `self.position` as
        pytables arrays.

        Arguments:
            save_pos (bool): if True, save the particles 3D trajectories
            total_emission (bool): if True, store only the total emission array
                containing the sum of emission of all the particles.
            rs (RandomState object): random state object used as random number
                generator. If None, use a random state initialized from seed.
            seed (uint): when `rs` is None, `seed` is used to initialize the
                random state, otherwise is ignored.
            wrap_func (function): the function used to apply the boundary
                condition (use :func:`wrap_periodic` or :func:`wrap_mirror`).
            path (string): a folder where simulation data is saved.
            verbose (bool): if False, prints no output.
        """
        if arg_4 is None:
            arg_4 = np.random.RandomState(arg_5=arg_5)
        arg_0.open_store_traj(arg_9=arg_9, arg_10=arg_10,
                              arg_3=arg_3, arg_6=arg_6)
        # Save current random state for reproducibility
```

```
        arg_0.traj_group._v_attrs['init_random_state'] = arg_4.get_state()

        arg_14 = arg_0.emission_tot if arg_2 else arg_0.emission

        print('- Start trajectories simulation - %s' % ctime(), flush=True)
        if arg_11:
            print('[PID %d] Diffusion time:' % os.getpid(), end='')
        arg_15 = 0
        arg_16 = arg_0.emission.chunkshape[1]
        arg_17 = arg_16 * arg_0.t_step

        arg_18 = arg_0.particles.positions
        arg_19 = 0
        for arg_20 in iter_chunksize(arg_0.n_samples, arg_16):
            if arg_11:
                arg_21 = int(arg_17 * (arg_15 + 1))
                if arg_21 > arg_19:
                    print(' %ds' % arg_21, end='', flush=True)
                    arg_19 = arg_21

            arg_22, arg_23 = arg_0._sim_trajectories(arg_20, arg_18, arg_4,
                                            arg_2=arg_2,
                                            arg_1=arg_1, arg_3=arg_3,
                                            arg_7=arg_7)

            ## Append em to the permanent storage
            # if total_emission, data is just a linear array
            # otherwise is a 2-D array (self.num_particles, c_size)
            arg_14.append(arg_23)
            if arg_1:
                arg_0.position.append(np.vstack(arg_22).astype('float32'))
            arg_15 += 1
            arg_0.store.h5file.flush()

        # Save current random state
        arg_0.traj_group._v_attrs['last_random_state'] = arg_4.get_state()
        arg_0.store.h5file.flush()
        print('\n- End trajectories simulation - %s' % ctime(), flush=True)
======== 3300
def Func(arg_0):
    """Properly format arXiv IDs."""
    if arg_0 and "/" not in arg_0 and "arXiv" not in arg_0:
```

```
        return "arXiv:%s" % (arg_0,)
    elif arg_0 and '.' not in arg_0 and arg_0.lower().startswith('arxiv:'):
        return arg_0[6:]  # strip away arxiv: for old identifiers
    else:
        return arg_0
======== 8100
def Func(arg_0, arg_1):
        """
        Retrieve various metadata associated with a URL, as seen by Skype.

        Args:
            url (str): address to ping for info

        Returns:
            dict: metadata for the website queried
        """
        return arg_0.conn("GET", SkypeConnection.API_URL, params={"url": arg_1},
                          auth=SkypeConnection.Auth.Authorize).json()
```

B.2 Universally easy examples

```
def Func(arg_0, arg_1=True, arg_2=False):
        """
        Remove all bounding boxes that are fully or partially outside of the image.

        Parameters
        ----------
        fully : bool, optional
            Whether to remove bounding boxes that are fully outside of the image.

        partly : bool, optional
            Whether to remove bounding boxes that are partially outside of the image.

        Returns
        -------
        imgaug.BoundingBoxesOnImage
            Reduced set of bounding boxes, with those that were fully/partially outside of
            the image removed.

        """
        arg_3 = [bb for bb in arg_0.bounding_boxes
```

```
                    if not bb.is_out_of_image(arg_0.shape, arg_1=arg_1, arg_2=arg_2)]
        return BoundingBoxesOnImage(arg_3, shape=arg_0.shape)
========
def Func(arg_0):
'''
Func - Copy this model, and return that copy.

  The copied model will have all the same data, but will have a fresh instance of the FIELDS ar
    and the INDEXED_FIELDS array.

  This is useful for converting, like changing field types or whatever, where you can load from

@return <IndexedRedisModel> - A copy class of this model class with a unique name.
'''

arg_1 = _modelCopyMap[arg_0]
_modelCopyMap[arg_0] += 1
arg_2 = type(arg_0.__name__ + '_Copy' + str(arg_1), arg_0.__bases__, copy.deepcopy(dict(arg_0._

arg_2.FIELDS = [field.copy() for field in arg_0.FIELDS]

arg_2.INDEXED_FIELDS = [str(idxField) for idxField in arg_0.INDEXED_FIELDS] # Make sure they di
    #  so do a comprehension of str on these to make sure we only get names

arg_2.validateModel()

return arg_2
========
def Func(arg_0):
    """
    Uses the ``msgconvert`` Perl utility to convert an Outlook MS file to
    standard RFC 822 format

    Args:
        msg_bytes (bytes): the content of the .msg file

    Returns:
        A RFC 822 string
    """
    if not is_outlook_msg(arg_0):
        raise ValueError("The supplied bytes are not an Outlook MSG file")
    arg_1 = os.getcwd()
```

```
    arg_2 = tempfile.mkdtemp()
    os.chdir(arg_2)
    with open("sample.msg", "wb") as msg_file:
        msg_file.write(arg_0)
    try:
        subprocess.check_call(["msgconvert", "sample.msg"],
                              stdout=null_file, stderr=null_file)
        arg_3 = "sample.eml"
        with open(arg_3, "rb") as eml_file:
            arg_4 = eml_file.read()
    except FileNotFoundError:
        raise EmailParserError(
            "Failed to convert Outlook MSG: msgconvert utility not found")
    finally:
        os.chdir(arg_1)
        shutil.rmtree(arg_2)

    return arg_4
========
def Func(arg_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6):
    """
    Convert reduce_sum layer.

    Args:
        params: dictionary with layer parameters
        w_name: name prefix in state_dict
        scope_name: pytorch scope name
        inputs: pytorch node inputs
        layers: dictionary with keras tensors
        weights: pytorch state_dict
        names: use short names for keras layers
    """
    print('Converting reduce_sum ...')

    arg_7 = arg_0['keepdims'] > 0
    arg_8 = arg_0['axes']

    def target_layer(arg_9, arg_7=arg_7, arg_8=arg_8):
        import keras.backend as K
        return K.sum(arg_9, arg_7=arg_7, arg_8=arg_8)

    arg_10 = keras.layers.Lambda(target_layer)
```

```
    arg_4[arg_2] = arg_10(arg_4[arg_3[0]])
========
def Func(arg_0):
        """Generator for the LIST OVERVIEW.FMT

        See list_overview_fmt() for more information.

        Yields:
            An element in the list returned by list_overview_fmt().
        """
        arg_1, arg_2 = arg_0.command("LIST OVERVIEW.FMT")
        if arg_1 != 215:
            raise NNTPReplyError(arg_1, arg_2)

        for arg_3 in arg_0.info_gen(arg_1, arg_2):
            try:
                arg_4, arg_5 = arg_3.rstrip().split(":")
            except ValueError:
                raise NNTPDataError("Invalid LIST OVERVIEW.FMT")
            if arg_5 and not arg_4:
                arg_4, arg_5 = arg_5, arg_4
            if arg_5 and arg_5 != "full":
                raise NNTPDataError("Invalid LIST OVERVIEW.FMT")
            yield (arg_4, arg_5 == "full")
```

B.3 Universally difficult examples

```
========
def Func(arg_0, arg_1, arg_2, arg_3=35, arg_4=35, arg_5=0, arg_6=0):
        """
        Rectangle Funces for all the colors in the list.
        """
        for arg_7 in arg_0:
            arg_7.Func(arg_1, arg_2, arg_3, arg_4, arg_6)
            arg_2 += arg_4 + arg_5
========
def Func(arg_0):
        """return the connection Func for this object's sockets."""
        return (arg_0.identity, arg_0.url, arg_0.pub_url, arg_0.location)
========
def Func(arg_0, arg_1, arg_2):
```

```
    """
    Interpolate passage times for shape points.

    Parameters
    ----------
    shape_distances: list
        list of cumulative distances along the shape
    shape_breaks: list
        list of shape_breaks
    stop_times: list
        list of stop_times

    Returns
    -------
    shape_times: list of ints (seconds) / numpy array
        interpolated shape passage times

    The values of stop times before the first shape-break are given the first
    stopping time, and the any shape points after the last break point are
    given the value of the last shape point.
    """
    arg_3 = np.zeros(len(arg_0))
    arg_3[:arg_1[0]] = arg_2[0]
    for arg_4 in range(len(arg_1)-1):
        arg_5 = arg_1[arg_4]
        arg_6 = arg_2[arg_4]
        arg_7 = arg_1[arg_4+1]
        arg_8 = arg_2[arg_4+1]
        if arg_5 == arg_7:
            arg_3[arg_5] = arg_2[arg_4]
        else:
            arg_9 = arg_0[arg_5:arg_7+1]
            arg_10 = ((np.array(arg_9)-float(arg_9[0])) /
                                float(arg_9[-1] - arg_9[0]))
            arg_11 = (1.-arg_10)*arg_6+arg_10*arg_8
            arg_3[arg_5:arg_7] = arg_11[:-1]
    # deal final ones separately:
    arg_3[arg_1[-1]:] = arg_2[-1]
    return list(arg_3)
========
def Func(arg_0, arg_1, arg_2):
        """Extracts the selected time frame as a new object.
```

```
        :param int start: Start time.
        :param int end: End time.
        :returns: class:`pympi.Elan.Eaf` object containing the Funced frame.
        """
        from copy import deepcopy
        arg_3 = deepcopy(arg_0)
        for arg_4 in arg_3.get_tier_names():
            for arg_5, arg_6, arg_7 in arg_3.get_annotation_data_for_tier(arg_4):
                if arg_5 > arg_2 or arg_6 < arg_1:
                    arg_3.remove_annotation(arg_4, (arg_1-arg_2)//2, False)
        arg_3.clean_time_slots()
        return arg_3
========
def Func(arg_0, arg_1, arg_2=-1):
    """Pack rdd with a specific collection constructor."""
    arg_3 = 0
    arg_4 = []
    for arg_5 in arg_0:
        if (arg_2 > 0) and (arg_3 >= arg_2):
            yield _pack_accumulated(arg_4, arg_1)
            arg_4 = []
            arg_3 = 0
        arg_4.append(arg_5)
        arg_3 += 1
    if arg_3 > 0:
        yield _pack_accumulated(arg_4, arg_1)
```

Appendix C

Taxonomy Sample

- Coding
  - Variables
    - Data types
      - Integer
      - String
      - Boolean
      - Floating-point
      - Array
      - Character
      - Double
      - Long
      - Byte
      - Short
      - Object
      - Enum
      - Null
      - Struct
      - Pointer
      - Void
    - Variable declaration
      - Name assignment
      - Data type specification
      - Scope definition
      - Memory allocation
      - Initialization
      - Visibility
      - Lifetime
      - Constant declaration
      - Declaration order
      - Variable naming conventions
      - Type inference
      - Declaration modifiers
      - Multiple declarations
      - Forward declaration
      - External declaration
      - Declaration visibility modifiers
    - Variable scope
      - Global Scope
      - Local Scope
      - Function Scope
      - Block Scope
      - Nested Scope
      - Lexical Scope
      - Enclosing Scope
      - Shadowing
      - Scope Chain
      - Global Variable
      - Local Variable
      - Parameter Scope
      - Module Scope
      - Closure Scope
      - Dynamic Scope
      - Block-Level Function Scope
    - Variable naming conventions
      - Camel case
      - Pascal case
      - Snake case
      - Hungarian notation
      - Avoiding reserved keywords
      - Using meaningful names
      - Avoiding ambiguous names
      - Avoiding excessive abbreviation
      - Avoiding numbers at the beginning of names
      - Using plural for collections
      - Using singular for single items
      - Avoiding excessive length
      - Consistency within a project
      - Following language-specific conventions
      - Avoiding unnecessary abbreviations
      - Avoiding reserved characters
  - Control Flow
    - If-else statements
      - Conditional statement
      - Boolean expression
      - If statement
      - Else statement
      - Else if statement
      - Nesting
      - Short-circuit evaluation
      - Ternary operator
      - Multiple if statements
      - Default case
      - Equality operator
      - Logical operators
      - Comparison operators
      - Negation operator
      - Scope
      - Flow control
    - Loops
      - Iteration
      - Control flow
      - Conditional statements
      - Loop variable
      - Infinite loops
      - For loop
      - While loop
      - Do-while loop
      - Loop control statements
      - Nested loops
      - Loop initialization
      - Loop increment/decrement
      - Loop termination condition
      - Loop efficiency
      - Loop indexing
      - Loop patterns
    - Switch statements
      - Case
      - Default
      - Break
      - Fall-through
      - Expression
      - Nested switch
      - Multiple cases
      - Value comparison
      - Enumerations
      - Strings
      - Flow control
      - Efficiency
      - Error handling
      - Constant expressions
      - Limited data types
      - Code organization
    - Nested control flow
      - Nested if-else statements
      - Nested for loops
      - Nested while loops
      - Nested do-while loops
      - Nested switch-case statements
      - Nested try-catch blocks
      - Nested if-else if-else statements
      - Nested for-each loops
      - Nested repeat-until loops
      - Nested break statements
      - Nested continue statements
      - Nested control flow within functions
      - Nested control flow within classes or objects
      - Nested control flow within recursion
      - Nested control flow with parallel processing
      - Nested control flow in event-driven programming
  - Functions
    - Abstraction
      - Encapsulation
      - Modularity
      - Polymorphism
      - Inheritance
      - Interface
      - Abstract Class
      - Generic Programming
      - Dependency Injection
      - Data Abstraction
      - Procedural Abstraction
      - Opaque Data Types
      - Data Hiding
      - Parametric Polymorphism
      - Design Patterns
      - Abstraction Layers
      - Delegation
    - Modularity
      - Abstraction
      - Encapsulation
      - Separation of Concerns
      - Loose Coupling
      - High Cohesion
      - Reusability
      - Scalability
      - Testability
      - Maintainability
      - Extensibility
      - Readability
      - Collaboration
      - Version Control
      - Debugging
      - Portability
      - Documentation
    - Parameters
      - Default parameters
      - Named parameters
      - Positional parameters
      - Variable-length parameters
      - Keyword parameters
      - Multiple parameters
      - Type annotations
      - Parameter passing by value
      - Parameter passing by reference
      - Named parameter unpacking
      - Tuple unpacking
      - Parameter validation
      - Parameter scoping
      - Parameter overloading
      - Parameter order
      - Parameter passing between functions
    - Recursion
      - Base case
      - Recursive case
      - Divide and conquer
      - Tree recursion
      - Mutual recursion
      - Tail recursion
      - Indirect recursion
      - Backtracking
      - Memoization
      - Fractal generation
      - Permutations and combinations
      - Tower of Hanoi
      - Fibonacci sequence
      - Depth-first search
      - Quick sort
      - Maze solving

Figure C.1: Sample of taxonomy generated by recursive LLM calls