Un-reinventing the Wheel: Large Scale Open Source Code Search

ETHAN HELLMAN and BENJAMIN SPECTOR

In this work, we investigate searching large-scale public code repositories using large language models. Our system, CodeCrawlr (Crawlr.), aims to increase developer productivity and accessibility by enabling simple translation between natural language and code snippets through a Google-like search application. Rather than take the time to re-implement commonly used code, CodeCrawlr will take a brief description of the desired functionality and instantly return multiple viable candidate functions. While there exist services to search code based off of keywords, to the best of our knowledge, there does not currently exist a service for performing code search based on functionality/semantics. CodeCrawlr, contains four main components. First, a data downloading and parsing pipeline, which checks the license of open-source GitHub repositories, downloads and organizes them, and parses the code. Second, an embeddings generation and storage pipeline. Third, a query back-end which produces relevant code given a programmer's in-progress work or natural-language query. Fourth, a front-end which allows users to submit queries and receive back code snippets (and links to their sources). We find CodeCrawlr to be surprisingly effective and useful given its small scale and believe it demonstrates the power of the techniques which underpin it.

Additional Key Words and Phrases: neural networks, large language models, code similarity, search, open-source

ACM Reference Format:
Ethan Hellman and Benjamin Spector. 2022. Un-reinventing the Wheel: Large Scale Open Source Code Search. 1, 1 (December 2022), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Within the field of machine programming, machine learning has brought new perspective to automatic code generation by treating it as an autoregressive natural language modeling problem. The burgeoning field, despite its nascence, has brought with it many opportunities and efficiencies. However, most work in this area has focused on generating new code with these models as opposed to leveraging their understanding to reuse existing work. We feel this is a missed opportunity, because – on a local scale – almost all code is generally similar to other code which has been written before.

We believe that this leads to a new potential paradigm of machine programming: search and adaptation as invention. In other words, we feel that because new ideas are almost always formed out of the rearrangement and improvement of previous work, new code should rarely need to be written from scratch. Rather, when we wish to invent new code, we should first aim to find previously written code which approximates what we want to do, and then adapt it to better fit our needs. While there may remain some kinds of programs that are too different from any code ever written before, we hypothesize that these programs should be very rare due to the scale of the modern open-source ecosystem. Furthermore, we posit that this factorization of invention – as search and adaptation – leads to easier, more reliable, and less harmful invention than current learned methods of program synthesis.

Regarding easier: the tools to download source code from public repositories are well-developed. Additionally, modern search engines can search through vast amounts of data in milliseconds. Together, this enables one to rapidly find a good starting point.

Regarding reliability: when code is written from scratch without formal verification, bugs can easily emerge. While existing open-source code is by no-means bug-free, one can use the semi-trust of repositories to decrease the probability of bugs by relying more heavily on trustworthy repositories and flagging code from untrustworthy repositories for review. Furthermore, if bugs or security vulnerabilities are later found in source code, it could potentially be automatically traced to both upstream and derivative code, which would allow these vulnerabilities to be more quickly resolved.

Regarding harm: modern learned program synthesis tools (to be discussed in more detail in the following section) are trained on vast amounts of publicly available code, and appear to sometimes reproduce code almost verbatim which is not freely licensed for that purpose. While legal questions of fair use are pending litigation, we feel that the laundering of code through these models is unethical. A search and adapt approach alleviates these concerns because one can simply index only freely available code. That way, the system cannot accidentally steal code.

In this work, we introduce one key component of our proposed search-and-adapt framework: the search engine. (Humans can do the adaptation for the time being.) The search engine takes in queries of intention – what the user wants to accomplish – and produces relevant code from a small subset of MIT-licensed GitHub which we have indexed.

Our search engine has four main components: the data pipeline, the embedding pipeline, the query engine, and the front-end.

One key limitation of our work which we wish to address up-front is its scale. As students, we set our budget for the project to $50, which includes both the expenses for indexing and searching for queries with the OpenAI API (the dominant expense) as well as compute to serve our queries. There are two main effects of this limitation on our work. First, due to their expense, we were not able to use the highest-quality embeddings which decreases the quality of our search results. Second, we are not able to index very many repositories, and thus our engine does not achieve the density of indexing that it would otherwise. Consequently, we believe that this project should be treated as a scaffolding and a proof-of-concept for the search engine in our search-and-adapt framework for program invention. With the injection of additional capital, we believe the performance of our approach would and the utility of our service increase dramatically.

Authors' address: Ethan Hellman, hellman1@stanford.edu; Benjamin Spector, bfs@stanford.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
XXXX-XXXX/2022/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

We divide this section into two parts. First, we examine the available datasets for code search and automatic code generation. Second, we consider the evolution of natural language processing (NLP) based methods.

2.1 Data

We organize this survey chronologically, with earliest work first.

We also note that we actually ended up using *none* of these datasets for our project, for two reasons. First, we wanted to be able to source the origin of code – we feel this is an important part of our contribution. Second, we realized that if we wanted to construct a representative example of what large-scale open-source code search would look like, we should sample it from large-scale open-source repositories. Nonetheless, we summarize the relevant datasets here because they were important to our thought process (and therefore must be cited) and would likely still be useful in future work.

Source code is rarely written in isolation, it depends on the context of the surrounding functions. One might imagine wanting to generate class member functions given English documentation and the programmatic context provided by the rest of the class. CONCODE (2018) is a dataset with over 100,000 examples of Java classes from online code repositories. [Iyer et al. 2018]

The Neural Code Search Evaluation Dataset (2019) uses the most popular Android repositories on GitHub (ranked by the number of stars) to create a search corpus with 24,549 repositories. The search corpus is indexed using all method bodies parsed from the repositories, leading to a total of 4,716,814 methods in this corpus. [Li et al. 2019]

CodeSearchNet (2020) is a large dataset of functions with associated documentation written in Go, java, JS, PHP, Python, and Ruby for open source projects on GitHub. It includes 6 million methods overall, 2 million of which have associated documentation (docstrings, JavaDoc, and more). [Husain et al. 2019]

Pseudo-code to Code (known as SPoC, though we do not understand the acronym very well) is a 2019 program synthesis dataset containing both code and pseudo-code for 18,356 programs. [Kulal et al. 2019]

We also investigated the 2021 Code Search and Question Answering dataset (CoSQA), which consists of 20,064 pairs of queries and code. These real-world user queries are collected from Bing query logs and the code for queries are taken from CodeSearchNet. [Huang et al. 2021]

Search4Code is a 2021 dataset which consists of 6596 Java queries and 4974 C# queries. The main purpose of this dataset is actually simply to determine whether queries are code queries or not, rather than in finding the appropriate code for the query. Like CoSQA, the dataset is also collected from real Bing query logs. [Rao et al. 2021]

CoDesc is a large datset of code and code descriptions which aggregates existing datasets such as CodeSearchNet and CONCODE (among others). It also uses Python-to-Java transpilation to increase the size of the dataset. Altogether, it consts of approximately 4.2M Java functions and their descriptions. [Hasan et al. 2021]

The CodeSyntax dataset (2022) consists of code annotated with syntax derived from the abstract syntax trees of the code. Its purpose is primarily to evaluate the quality of generated code's syntax structure. [Shen et al. 2022]

Cross-Lingual Code SnippeT (XLCoST) is a large-scale 2022 dataset of parallel code written in seven different languages. It contains just over 1M different snippets in these different languages and is designed to aid in research into cross-lingual code intelligence. [Zhu et al. 2022]

2.2 Methods

In the Natural Language Processing (NLP) space, word2vec's distributed representation of words played a key role in enabling deeper understanding of language semantics. [Mikolov et al. 2013] Similarly, code2vec enabled learning distributed representations of code in vectors or "code-embeddings," for different downstream tasks. Using a combination of Abstract Syntax Trees (AST) and a "novel" soft-attention network architecture, this technique achieved up to 17% relative improvement, 5x faster training, and greater generalizability over the previous state of the art. [Alon et al. 2019]

Since [Alon et al. 2019], significant gains have been realized with the advent of the Transformer [Vaswani et al. 2017] - a simple network architecture solely based on attention mechanisms. Advances in transformers reduced the number of trainable parameters, as well as increased model performance and parallelizability. Fast-forwarding to 2020, Large-Language Models (LLM's) like GPT-3 [Brown et al. 2020] can now be trained with up to 175 billion parameters - 10x more than any previous language model - to perform a general array of NLP tasks. More recently, versions of the model, such as GPT-J [Wang and Komatsuzaki 2021], GPT-Neo, [Black et al. 2022] and CodeBERT [Feng et al. 2020], amongst others have fueled progress in program synthesis. Following GTP-3, OpenAI released Codex, a GPT model with 12 billion parameters fine-tuned on 159 GB of code from Github to test LLM code-writing capabilities. [Chen et al. 2021] The authors found that Codex is able to generate at least one correct function output out of 100 samples for 77.5% of the coding prompts it is fed from "HumanEval" and significantly outperforms its competitors. While this does represent an impressive advancement in machine programming, one of its major drawbacks is its lack of accessibility. Codex is not open-source, and clarity on what code was used to train the model is sparse. As such, [Xu et al. 2022] created PolyCoder, an open-source LLM trained on 249 GB of code across 12 different languages.

Key here are effective embedding representations of both code and text to enable better performance on down-stream tasks - including search. [Neelakantan et al. 2022] demonstrated how a contrastive learning objective using GPT and Codex produce text and code embeddings with state-of-the-art results over previous best supervised and unsupervised models. The model consists of a transformer encoder network where the training objective is to assess the cosine similarity between samples. More recently, [Gao et al. 2022] introduced a similarly-inspired unsupervised contrastive learning technique for sentence embeddings which leverages BERT and RoBERTa. Their approach elegantly stresses the power of using standard dropout as noise. They build on this with a slightly-augmented

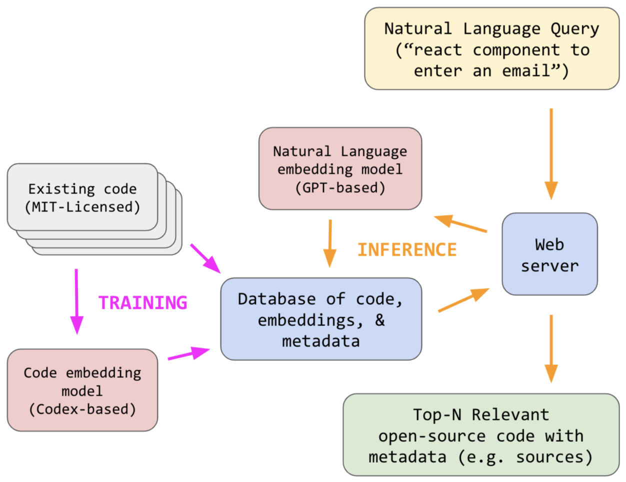

Natural Language Query ("react component to enter an email")
Natural Language embedding model (GPT-based)
Existing code (MIT-Licensed)
INFERENCE
Web server
TRAINING
Database of code, embeddings, & metadata
Code embedding model (Codex-based)
Top-N Relevant open-source code with metadata (e.g. sources)

Fig. 1. A diagram of the overall architecture of Crawlr.

supervised approach which outperforms [Neelakantan et al. 2022]. as the state-of-the-art text embedding technique.

3 SYSTEM DESIGN

The overall design of our system consists of four main components: the data pipeline, embedding pipeline, query back-end, and front-end user interface. While none of these components are particularly original in nature, we feel that the engineering to synthesize them together into a coherent product is what makes the project interesting. A diagram of the project is presented in figure 1.

First, we constructed a database of code, metadata, and embeddings taken from permissively (MIT) licensed GitHub repositories. While we originally tried automating the selection of trustworthy repositories, we found that this approach dramatically over-represented certain kinds of repositories which were unlikely to contain much interesting code. (For example, there are several high-starred open-source repositories which consist of just a README file of other useful resources, which is not particularly useful for our purposes.) We then calculated how many repositories we would actually be able to index on our limited budget and realized it would be both easier and probably higher-quality to select our repositories by hand. A list of the repositories we indexed can be found in Appendix A. We then augmented the repositories with some additional metadata so that code can be traced back directly to the source. The code we indexed ended up consisting of 13,286,894 bytes of code distributed over 22,335 functions.

Second, we built a pipeline to parse, embed, and store code. First, a Python script crawls through directories of files, parses functions from .py files, assembles the code with its relevant metadata, and stores it in json. Then, a second script batches these functions, determines which ones will (due to length) be unable to be embedded, calls the relevant OpenAI endpoint (which contains a modified version of Codex [Chen et al. 2021]) to embed them, and stores them in batched json files for easy loading. In the end, it cost approximately $33 to index at around 400 kB/$.

Third, we built a query engine using the library FAISS which embeds an input query and returns the nearest-neighbor results using an inner product similarity with its database. This is deployed on an e2-standard-2 server on Google Cloud. One benefit of the inner product method is that it allows us to tune the scores for different levels of trust and usefulness. When normalized, it produces the naive cosine similarity, but we can also penalize code which comes from less reliable sources or which contains features which we believe negatively signal its usefulness by scaling down their corresponding embedding vectors, and increase code containing useful features by upscaling their vectors proportionately, too.

Finally, we built a lightweight user interface to allow users on the web to query the system and receive code back. The site is modeled (somewhat minimalistically) on the original Google website and is served in flask. The reason for expending this effort is that, if all goes well, we hope to be able to conduct a live demonstration with the class during our final presentation.

As we stated in the introduction, we wrote our infrastructure to be scalable: simply input more money and it can be run on an increasing fraction of all of the permissively licensed open-source code on GitHub. (Some additional effort would be required to build parsers for other languages but this is not too difficult.)

Regarding the ethics of this project, we had a conversation with OpenAI before embarking on putting up our front-end to ensure no harm would come from the project. We concluded that because we only query embedding APIs rather than completion APIs, we could not foresee any risks from exposing the model to the internet. Additionally, we ensured that we only indexed permissively-licensed code, so that even if someone finally stumbled on our project they would not accidentally end up using code that is unlicensed for those purposes. Finally, we emphasized that the project is a class project only expected to be visible to us and, potentially, the class. OpenAI agreed that the risks were minimal and advised us to proceed. Of course, if the project were to be expanded, additional review would be necessary.

3.1 CodeCrawlr Application Design

A strong motivating factor for this project was the ability to utilize the underlying nearest-neighbor search to build a usable tool in the form of a web-application. While Google ushered in the development of web search, Crawlr. similarly aims to service initial forms of online code search. As such, our application pays homage to the early implementations of google.

3.1.1 CodeCrawlr Back-End Design. The back-end for the Crawlr. web application is hosted on a remote Flask server. We decided to use Flask for our back-end because all of our functionality was initially implemented in Python. Most importantly, our embedding generation and our nearest-neighbor search - the core of our service - were written in Python. Flask is a light-weight micro web framework written in Python. It does not require additional tools or libraries. Additionally, Flask does not have a database abstraction layer or perform form validation. This allowed us to quickly develop and deploy a server for Crawlr.

Crawlr. requires the embeddings and the code snippet itself in order to perform its search functionality. Given the amount of data we have to "code crawl" (199MB), we did not take the time to optimize our search by creating a custom database. Rather, our data is stored in JSON format locally on the server. These files contain the embeddings, actual snippets, as well as other metadata such as function names and filepaths. This allow for simple integration with our existing embedding generation and nearest-neighbor search pipeline. As such, our Flask server simply communicates with our existing functions to perform queries.

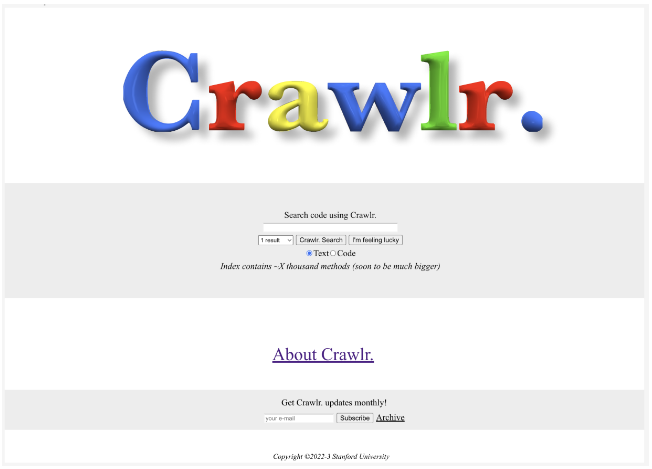
3.1.2 CodeCrawlr Front-End Design. The front-end of our web application is built using HTML, CSS, JavaScript, and React. Axios is used as our promise-based HTTP client for the browser. Upon accessing codecrawlr.com, the user is given a clear prompt in the middle of the screen - a query input field (figure 2). There are multiple ways Crawlr. can be used for search. The two search methods implemented in our web application are text-to-code and code-to-code. Crawlr.'s default is text to code search; however, this can be changed with the click a button. Additionally, the user is able to specify the desired number of results they wish to see. Given our choice of a nearest-neighbor search, this is trivial to implement.

Illustrated in figure 3 is a page of top-n (user-specified) closest results from performing nearest-neighbor search. Results are ranked by effective "distance" to the search query. The code is displayed in a React component capable of rendering properly-formatted code in a variety of languages. This allows the user a quick glance at the code. Clicking on the function name redirects the user to a page dedicated to that function for a more comprehensive view. As seen in figure 4, this page also contains a button labeled "Code Crawl."

This button automatically performs a code-to-code nearest-neighbor search using the code snippet of the current page. figure 6 shows the results of clicking the "Code Crawl" button. As with text-to-code, the crawl results display the same information, are ranked by effective "distance" to the code snippet query, and can be clicked to move to the next code snippet page. While it remains to be thoroughly user-tested, the code crawl functionality is meant to assist in both search and general code exploration. While an immediate candidate code snippet may not adequately fulfill the intention of the user's query, in our exploration (ref. Section 4.) we find that it may be reasonably close to the desired result. This can result from differences in the search terms different individuals use as well as the under-performance of Crawlr. search. Hence, the crawl functionality enables the user to search the adjacent space of results for potentially more appropriate code snippets.

4 EXPERIMENTAL RESULTS

The main limitation of our approach of using data directly from GitHub, which we realized only after we had implemented the project, is that even though we didn't to train with any of the queries from any of the datasets listed in the background section, it would have made our lives easier in evaluation. As a result, we have constructed several of our own benchmarks which can be bootstrapped from our data. While they are imperfect, we feel they at least give a basic understanding of how our system performs. We begin with the quantitative analyses we performed and then turn later to our qualitative assessments of the system.



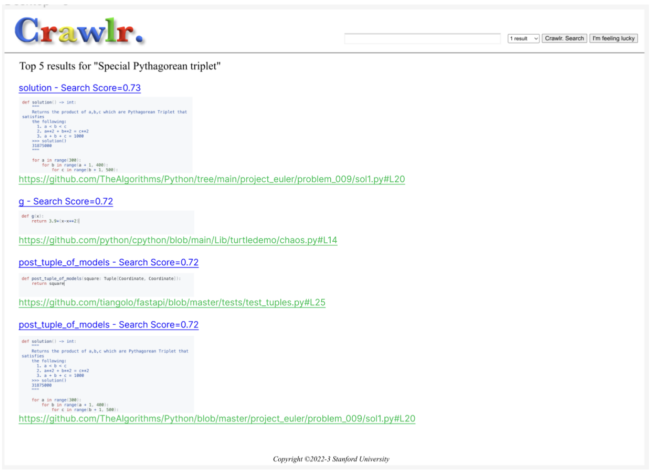
Search code using Crawlr.

Index contains ~X thousand methods (soon to be much bigger)

Get Crawlr. updates monthly!

Copyright ©2022-3 Stanford University

Fig. 2. Crawlr. Homepage



Top 5 results for "Special Pythagorean triplet"

solution - Search Score=0.73

```
def solution() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet that
satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution()
    31875000
    """

    for a in range(300):
        for b in range(a + 1, 400):
            for c in range(b + 1, 500):
```

https://github.com/TheAlgorithms/Python/tree/main/project_euler/problem_009/sol1.py#L20

g - Search Score=0.72

```
def g(x):
    return 3.9*(x-x**2)
```

https://github.com/python/cpython/blob/main/Lib/turtledemo/chaos.py#L14

post_tuple_of_models - Search Score=0.72

```
def post_tuple_of_models(square: Tuple[Coordinate, Coordinate]):
    return square
```

https://github.com/tiangolo/fastapi/blob/master/tests/test_tuples.py#L25

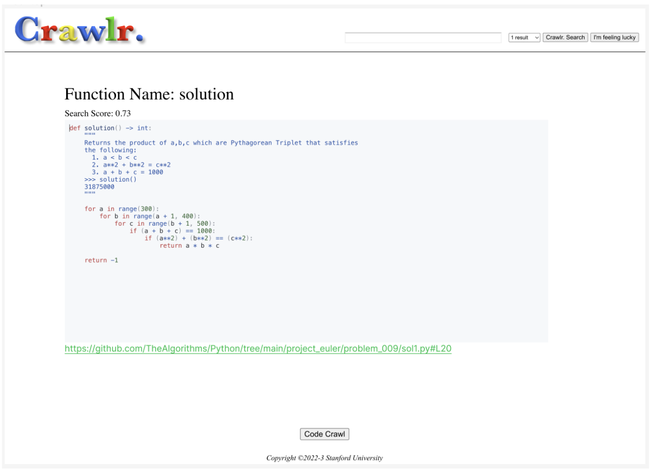
post_tuple_of_models - Search Score=0.72

```
def solution() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet that
satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution()
    31875000
    """

    for a in range(300):
        for b in range(a + 1, 400):
            for c in range(b + 1, 500):
```

https://github.com/TheAlgorithms/Python/blob/master/project_euler/problem_009/sol1.py#L20

Copyright ©2022-3 Stanford University

Fig. 3. Crawlr. Search Query Results Page



Function Name: solution

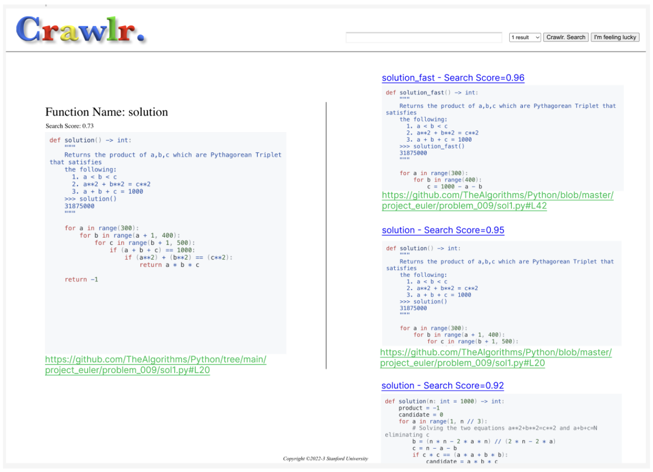
Search Score: 0.73

```
def solution() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet that satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution()
    31875000
    """

    for a in range(300):
        for b in range(a + 1, 400):
            for c in range(b + 1, 500):
                if (a + b + c) == 1000:
                    if (a**2) + (b**2) == (c**2):
                        return a * b * c

    return -1
```

https://github.com/TheAlgorithms/Python/tree/main/project_euler/problem_009/sol1.py#L20

Copyright ©2022-3 Stanford University

Fig. 4. Crawlr. Code Snippet Page



Function Name: solution

Search Score: 0.73

```python
def solution() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet
that satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution()
    31875000
    """

    for a in range(300):
        for b in range(a + 1, 400):
            for c in range(b + 1, 500):
                if (a + b + c) == 1000:
                    if (a**2) + (b**2) == (c**2):
                        return a * b * c

    return -1
```

https://github.com/TheAlgorithms/Python/tree/main/project_euler/problem_009/sol1.py#L20

solution_fast - Search Score=0.96

```python
def solution_fast() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet that
satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution_fast()
    31875000
    """

    for a in range(300):
        for b in range(400):
            c = 1000 - a - b
```

https://github.com/TheAlgorithms/Python/blob/master/project_euler/problem_009/sol1.py#L42

solution - Search Score=0.95

```python
def solution() -> int:
    """
    Returns the product of a,b,c which are Pythagorean Triplet that
satisfies
    the following:
      1. a < b < c
      2. a**2 + b**2 = c**2
      3. a + b + c = 1000
    >>> solution()
    31875000
    """

    for a in range(300):
        for b in range(a + 1, 400):
            for c in range(b + 1, 500):
```

https://github.com/TheAlgorithms/Python/blob/master/project_euler/problem_009/sol1.py#L20

solution - Search Score=0.92

```python
def solution(n: int = 1000) -> int:
    product = -1
    candidate = 0
    for a in range(1, n // 3):
        # Solving the two equations a**2+b**2=c**2 and a+b+c=N
eliminating c
        b = (n * n - 2 * a * n) // (2 * n - 2 * a)
        c = n - a - b
        if c * c == (a * a + b * b):
            candidate = a * b * c
```

Copyright ©2022-3 Stanford University

Fig. 5. Crawlr. Crawl Page

4.1 Quantitative evaluations

The first benchmark we used was based on Project Euler. Project Euler is a set of online programming challenges, and within one of the repositories of our dataset happen to be solutions to many of these problems. So, we evaluated Crawlr by querying it with the title of the problem (e.g. "Special Pythagorean triplet") and seeing with what probability it produced code from the actual solutions to these problems.

We evaluated this by hand on the first 50 problems of Project Euler and found it got 14 of them right. While this performance may not sound impressive, we believe that more careful examination reveals that it is actually surprisingly good.

First, many of the queries from this dataset are highly ambiguous. Take, as examples, problem 13 ("Large Sum"), problem 22 ("Names scores"), or problem 48 ("Self powers"). All of these could actually mean many things, and it is unlikely that even a human oracle, given the prompt "Large Sum" would immediately return the solution to the problem (which involves returning only the first 10 digits of a large sum). We additionally found that for many of the problems which it could not solve, prompting with a larger substring from the problem description ("Work out the first ten digits of the sum") yielded the correct response. We actually think it is rather remarkable that Crawlr can return the correct solution to problems using a query like "Sum square difference" almost 30% of the time.

Additionally, regarding the ambiguity of these queries, another 10% of the time Crawlr would fail to return the complete solution to the problem, but would produce code that is clearly relevant and useful for the solution. For example, with problem 25 ("1000-digit Fibonacci number") although it failed to produce the complete solution, it did return a fast algorithm for Fibonacci numbers using matrix exponentiation, which could be easily used to construct the full solution. We reiterate that Crawlr currently uses neither search data nor fine-tuning in its responses.

A second test we did was to see how well Crawlr is able to identify full functions given random substrings from their body. We tried this with three different lengths of substrings: 50 characters, 100 characters, and 200 characters. (In all situations we ensured the length of true function was significantly longer to prevent cases from being too easy.) The results of this investigation are presented in table 1.

Substring length (chars)	Accuracy
50	49.7%
100	71.1%
200	84.2%

Table 1. Probability of Crawlr successfully determining the origin function given a random substring of varying lengths.

We suspect that this measurement is in some ways more of a measurement of the data than it is of Crawlr. Consider, for example, the following 50 character query:

"cls, v, field, **kwargs):\n   calls.append"

We suspect this query would be hard to resolve under most conditions. On the flip side, a query which happened to include function arguments which were distinctive might be more likely to return the right response.

The main benefit of this metric of Crawlr, however, is that it measures to what degree the embedding models used have actually compressed the code in addition to understanding it at a high level. What we mean by this is that if Crawlr were unable to complete this task, it would indicate that the embeddings don't capture well the contents of the code itself in a consistent way, even if they captured the meaning of the code. But because Crawlr is able to complete this task reasonably well even with often small substrings of code, it indicates that the code's contents is being compressed. We then used this knowledge to guide the construction of the traversal search we built into the front-end.

4.2 Qualitative evaluations

We began by investigating the code that we indexed and the embeddings we produced. First, we were interested in the distribution of lengths of functions. We hypothesized that certain functions within the dataset would be too short, and others too long, to likely be useful. (One result of this is that in our final model which is deployed we penalize the scores of function lengths which are either shorter than 100 characters or longer than 3000.) We show a histogram of this distribution in figure 6

A second visualization of the code we constructed is a UMAP projection of the embeddings. UMAP is a method for reducing the dimensionality of datasets which assumes a uniform distribution of data on a locally connected manifold with a Riemannian metric and then constructs a representation which minimizes the distortion of the topological structure. [McInnes et al. 2018] In figure 7 we show these embeddings, reduced from 2,048 dimensions to 2 dimensions, and colored according to their source repositories. First, we note that we suspect these graphs are highly interesting in their own right. For example, where these repositories overlap, it suggests that they have considerable amounts of code in common which could potentially be refactored to improve both code conciseness and also code quality. On the flip side, when the clusters are well-separated, it suggests that the code contained is very different. For example,

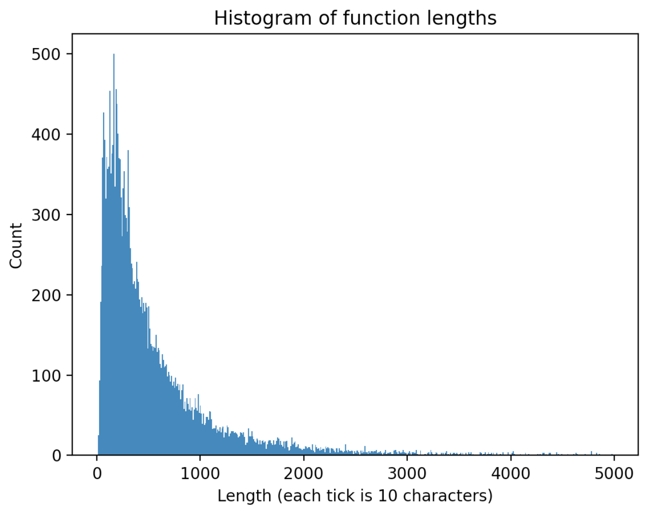


Fig. 6. A histogram of the lengths of the 22,335 functions we indexed. Each tick is 10 characters, and the total function length includes its declaration.

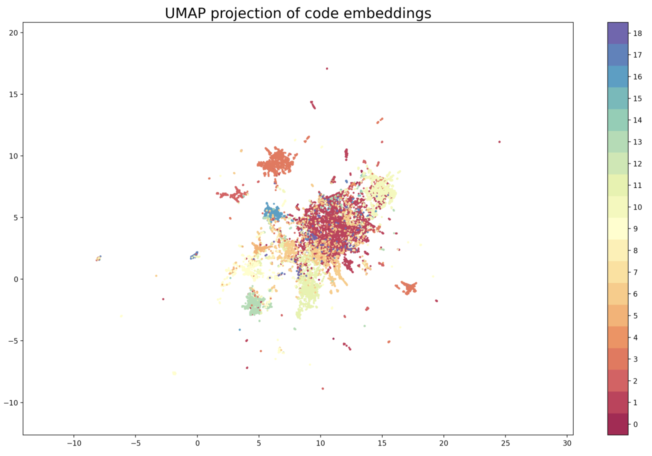


Fig. 7. An index of the repositories in this visualization is as follows: 0: thumbor/thumbor, 1: python/cpython, 2: ManimCommunity/manim, 3: explosion/spaCy, 4: kitao/pyxel, 5: pallets/flask, 6: ansible/ansible, 7: zappa/Zappa, 8: psf/requests, 9: tiangolo/fastapi, 10: TheAlgorithms/Python, 11: localstack/localstack, 12: vipstone/faceai, 13: pydantic/pydantic, 14: faif/python-patterns, 15: ageitgey/face_recognition, 16: Textualize/rich, 17: soimort/you-get, 18: scrapy/scrapy. A higher resolution version is available **here**.

the orange clusters which are found on the top left and bottom right of the image correspond to the repository explosion/spaCy, a natural language processing library, which is indeed meaningfully distinct from all the other repositories we indexed. By contrast, python/cpython is quite spread out over the embedding space, which makes sense since one expects (and hopes) for its functionality to cover a wide range of programming and use-cases.

We wish to emphasize that the purpose of this visualization is – unlike other uses of UMAP in categorical classification settings – not to demonstrate that the representations partition the input. In fact, this property would be largely undesirable, because real codebases do have real overlap, and in fact this property is necessary for our project to have any value whatsoever. One expects – and observes – that some repositories should have more overlap with others, and some should be more distinct. Nonetheless, we feel that this illustration provides intuition about the structure of and relationships between these popular codebases.

We now turn to a qualitative assessment of the engine itself. This too has two components. The first is the performance of the initial engine; the second is the usability of the crawling component. In both cases these results are derived from considerable time spent querying it and testing its limits.

In general, we have found that CodeCrawlr performs very well when it is given queries that are fairly close matches with the documentation of a function – that is, it does not need to understand the body of the function too well to produce a reasonable result. For example, if one asks it for a "Breadth-First Search" one gets back pretty good answers: the top four responses include two different breadth-first search implementations, one depth-first search implementation, and one A* implementation. On the flip side, when the query involves a completely different phrasing or interpretation of the contents of a function, Code Crawlr does not perform particularly well.

One major limitation of the work is that we did not generally index class methods. The reason for this is that class methods rarely can stand on their own in isolation, so they're not very useful to return to users. However, we have since come to realize that often times pure functions are written within classes for organizational reasons, and that our current approach of ignoring classes also misses out on lots of the most important code from these repositories.

5 FUTURE WORK

There is quite a lot of low-hanging fruit to improve CodeCrawlr. The simplest would be to increase its scale. As of now, we constructed it on a subset of 19 repositories containing 13.2MB of code at a cost of $50. By contrast, GitHub hosts 78M repositories containing 10s of terabytes of codes, at least a reasonable fraction of which is open-sourced. Thus, one can conservatively estimate that one could simply index 100,000 times more code in order to produce significantly more specific and high-quality results.

A second source of future optimization is using query data to improve the search engine. Very few search engines work well without being optimized for their workload; indeed, commercial search engines like Google have the benefit of trillions of queries by which they can improve their recommendations. In the future, one might imagine up or down-weighting certain code snippets based on how often they are accepted when suggested, and also iteratively refining the embeddings themselves according to user queries.

There is also considerable fine-tuning one could imagine doing in other ways. Right now we use very little semi-trust data to improve our results; the main one (discussed earlier) is to prefer returning functions which are neither too short nor too long. But, there are lots of other features one could imagine using. For example, if we indexed more low-quality data, it would be natural to adjust the scores of this low-quality data to really ensure it is a good match before we would consider returning the code snippet.

One could also improve CodeCrawlr by allowing queries to include richer metadata. For example, one could let the user choose the levels of trust of the source they are willing to accept, or the approximate length of the code they are looking for.

Regarding indexing class methods, one useful change which we might make in the future is to automatically detect whether class methods are pure or not (one could actually check this fairly easily in Python by seeing if the function ever references "self") and at least include those functions. In the long-term, we also hope it would be possible to handle object-oriented paradigms better within a code search framework, as we feel this is a significant difficulty.

6 CONCLUSIONS

In this work, we investigated using large language models based on Codex [Chen et al. 2021] to search permissively licensed public code repositories. Our system, CodeCrawlr consists of four parts: a data pipeline, an embeddings pipeline, a query engine, and a front-end. We conducted both quantitative and qualitative analyses of CodeCrawlr and found it to be moderately effective at code search despite its small scale, though it has serious limitations imposed both by budgetary constraints and the naivety of our approach. Finally, we proposed modifications which we feel could significantly improve its performance in the future based on the methods which power existing search engines. We hope our work illustrates and inspires a broader class of "search-and-adapt" approaches for inventing out of intent.

7 ACKNOWLEDGMENTS

We'd like to thank the course staff for a great semester!

REFERENCES

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745* (2022).

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

Lei Gao, Lijuan Zhang, Lei Zhang, and Jie Huang. 2022. RSVN: A RoBERTa Sentence Vector Normalization Scheme for Short Texts to Extract Semantic Information. *Applied Sciences* 12, 21 (2022), 11278.

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md Haque, Mahim Anjum, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. CoDesc: A Large Code-Description Parallel Dataset. *arXiv preprint arXiv:2105.14220* (2021).

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239* (2021).

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural code search evaluation dataset. *arXiv preprint arXiv:1908.09804* (2019).

Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

Nikitha Rao, Chetan Bansal, and Joe Guan. 2021. Search4Code: Code search intent classification using weak supervision. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 575–579.

Da Shen, Xinyun Chen, Chenguang Wang, Koushik Sen, and Dawn Song. 2022. Benchmarking Language Models for Code Syntax Understanding. *arXiv preprint arXiv:2210.14473* (2022).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. *arXiv preprint arXiv:2206.08474* (2022).

A REPOSITORIES INDEXED

https://github.com/thumbor/thumbor,
https://github.com/python/cpython,
https://github.com/ManimCommunity/manim,
https://github.com/explosion/spaCy,
https://github.com/kitao/pyxel,
https://github.com/pallets/flask,
https://github.com/ansible/ansible,
https://github.com/zappa/Zappa,
https://github.com/psf/requests,
https://github.com/tiangolo/fastapi,
https://github.com/TheAlgorithms/Python,
https://github.com/localstack/localstack,
https://github.com/vipstone/faceai,
https://github.com/pydantic/pydantic,
https://github.com/faif/python-patterns,
https://github.com/ageitgey/face_recognition,
https://github.com/Textualize/rich,
https://github.com/soimort/you-get,
https://github.com/scrapy/scrapy