Lecture 17: Code Generation

Giới thiệu

Hôm nay, chúng ta sẽ tìm hiểu về nhiệm vụ tạo mã (code generation), một lĩnh vực nghiên cứu mà hiện nay đang trở nên rất hữu ích. Tôi muốn chia sẻ về những kiến thức cơ bản và những tiến bộ mới nhất trong lĩnh vực này.

Trước đây, tôi đã nói nhiều về các khái niệm chung, không liên quan đến nhiệm vụ cụ thể nào. Tôi hiểu rằng không phải ai cũng quan tâm đến các nhiệm vụ mà tôi sẽ đề cập. Tuy nhiên, tôi hy vọng rằng các bạn có thể áp dụng những câu hỏi tương tự vào bất kỳ nhiệm vụ nào mà bạn quan tâm.

Cụ thể, tôi sẽ trình bày về mục tiêu của nhiệm vụ: tại sao chúng ta thực hiện nhiệm vụ đó và tầm quan trọng của nó. Tôi cũng sẽ nói về các tập dữ liệu có thể sử dụng để huấn luyện hoặc kiểm tra mô hình, các tiêu chí đánh giá và cách chúng ta đánh giá hiệu quả của mô hình cả thủ công lẫn tự động. Cuối cùng, tôi sẽ trình bày về các mô hình và phương pháp giải quyết vấn đề.

Code Generation - Overview and Objectives

Code Generation

Tạo mã tự động là quá trình tạo ra mã thực thi, đóng vai trò như một giao diện cho chương trình hoặc máy tính. Có nhiều phương pháp khác nhau để thực hiện điều này, và lý do chính để chúng ta quan tâm đến nó là vì tầm quan trọng của kỹ thuật phần mềm. Khả năng tạo mã tự động giúp tăng tốc quá trình phát triển phần mềm. Đây là một công cụ thực tiễn mà tôi khuyến khích mọi người sử dụng để cải thiện quy trình làm việc của mình.

Ngoài ra, tạo mã tự động còn cho phép các mô hình truy cập vào các công cụ khác nhau. Ngay cả khi bạn không làm việc trực tiếp với các tác vụ liên quan đến phần mềm, điều này vẫn có thể hữu ích. Tuy nhiên, tôi sẽ trình bày chi tiết hơn về vấn đề này trong một bài học về LLM Agents.

Một điểm khác mà tôi muốn đề cập là, ngay cả khi bạn không sử dụng mã, việc huấn luyện trên mã đã được chứng minh là mang lại lợi ích cho các mô hình học tập, đặc biệt là trong việc giải quyết các nhiệm vụ suy luận đa bước phức tạp. Đây là một lý do khác để bạn quan tâm đến mã.

Inputs and Outputs

Đầu vào của chúng ta là một mô tả về những gì cần thực hiện, và đầu ra sẽ là mã nguồn. Khi viết chương trình, làm thế nào để bạn mô tả điều bạn muốn triển khai trước khi thực hiện nó? Dưới đây là một số cách mà người ta có thể đưa ra mô tả:

- Kiểu dữ liệu của đầu vào và đầu ra: Ví dụ, trong Python, bạn có thể chỉ định kiểu dữ liệu cho các hàm.
- Yêu cầu về độ phức tạp và ràng buộc: Các ràng buộc có thể bao gồm yêu cầu về tốc độ hoặc việc sử dụng các thư viện cụ thể.

- Mã giả (Pseudo code): Mã giả thường được viết bằng ngôn ngữ tự nhiên. Ví dụ, bạn có thể yêu cầu một chương trình tạo giao diện web để đặt pizza.
- Mô tả đa phương tiện: Bạn có thể mô tả ứng dụng bằng cách chỉ ra vị trí các thành phần trên giao diện, như tên người dùng, menu, và giỏ hàng.
- Đầu vào và đầu ra dưới dạng unit test: Bạn có thể cung cấp các unit test để chỉ ra đầu vào và đầu ra mong đợi.

Những yếu tố này không chỉ hữu ích cho lập trình viên mà còn cho các mô hình sinh mã tự động. Việc sử dụng kiểu dữ liệu và ràng buộc là những điểm quan trọng có thể được tích hợp vào mô hình sinh mã.

Example: Github Copilot

GitHub Copilot là một công cụ mạnh mẽ giúp tự động hoàn thành mã dựa trên các comments mà bạn viết. Ví dụ, bạn chỉ cần viết các comments và Copilot sẽ tự động điền mã lệnh tương ứng. Tuy nhiên, cần lưu ý rằng mã được tạo ra có thể không hoàn toàn chính xác, vì vậy việc kiểm tra lại là cần thiết.

Ngoài ra, GPT-4 và Claude 3 cũng là những công cụ hữu ích. Chẳng hạn, bạn có thể yêu cầu Claude tạo một ứng dụng React dựa trên ảnh chụp màn hình. Tuy nhiên, do các mô hình ngôn ngữ lớn thường bị giới hạn về số lượng token, chúng có thể không hoàn thành toàn bộ dự án mà chỉ cung cấp một phần mã. Điều này có nghĩa là bạn cần phải điều chỉnh và bổ sung thêm để hoàn thiện dự án của mình.

Example: Claude 3

Gần đây, tôi đã tham gia phát triển một ứng dụng mã nguồn mở hỗ trợ lập trình. Tôi chỉ cần mô tả yêu cầu của mình, như việc tạo một ứng dụng với cửa sổ chat bên trái và ba khung bên phải (terminal, planner, và code editor), và Claude đã tạo ra một bản mẫu cho tôi. Mặc dù giao diện ban đầu không đẹp, nhưng tôi có thể yêu cầu điều chỉnh như thay đổi màu nền và thêm biểu tượng người dùng.

GitHub Copilot và Claude phục vụ các mục đích khác nhau. Copilot chủ yếu hỗ trợ hoàn thành mã ngắn, phù hợp khi bạn đã quen thuộc với ngôn ngữ lập trình và cần kiểm soát chi tiết. Ngược lại, Claude thích hợp cho các tác vụ dài hơn, như xây dựng một lớp hoàn chỉnh, đặc biệt hữu ích khi bạn không quen thuộc với ngôn ngữ lập trình đó.

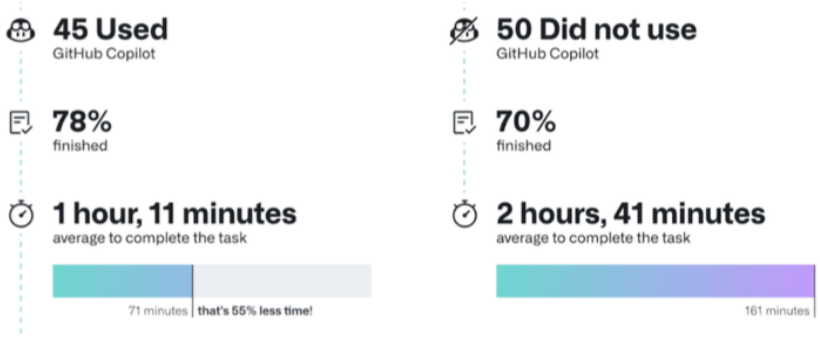
Một điểm khác biệt quan trọng là tốc độ phản hồi. Copilot cần hoạt động nhanh chóng để theo kịp tốc độ suy nghĩ của lập trình viên, trong khi Claude có thể chậm hơn, cho phép bạn yêu cầu một ứng dụng web và quay lại sau bữa tối để nhận kết quả.

Why do we do it?

Trong vai trò là một nhà phát triển phần mềm, việc sử dụng các công cụ hỗ trợ như GitHub Copilot có thể mang lại nhiều lợi ích đáng kể. Một nghiên cứu từ GitHub, được thực hiện ngay

sau khi Copilot ra mắt, đã chỉ ra rằng việc tạo mã tự động không chỉ giúp tăng năng suất mà còn mang lại giá trị kinh tế lớn. Theo thống kê, tổng thu nhập hàng năm của các nhà phát triển phần mềm lên tới 175 tỷ USD, cho thấy đây là một ngành nghề có giá trị cao. Nếu có thể tăng tốc độ phát triển phần mềm, giá trị này sẽ còn tăng thêm.

Nghiên cứu của GitHub đã phân chia ngẫu nhiên các nhà phát triển thành hai nhóm: một nhóm sử dụng Copilot và một nhóm không sử dụng. Kết quả cho thấy nhóm sử dụng Copilot hoàn thành nhiệm vụ nhanh hơn 55% và tỷ lệ hoàn thành công việc tăng 8% so với nhóm không sử dụng. Điều này không quá ngạc nhiên, vì các công cụ hỗ trợ mã hóa giúp tăng tốc độ viết mã và cải thiện hiệu quả công việc. Ngoài ra, Copilot còn hỗ trợ viết docstring, giúp tài liệu hóa mã nguồn dễ dàng hơn.



Difference Between code and Natural Language

Có những khác biệt rõ rệt giữa mã nguồn và ngôn ngữ tự nhiên, ảnh hưởng đến cách chúng ta xây dựng các mô hình xử lý. Đầu tiên, mã nguồn có ngữ pháp rất nghiêm ngặt; chỉ cần một lỗi nhỏ cũng có thể khiến chương trình không hoạt động, đòi hỏi sự cẩn thận cao. Ngược lại, ngữ pháp trong ngôn ngữ tự nhiên có thể chấp nhận một số lỗi nhỏ mà không ảnh hưởng nhiều đến ý nghĩa.

Thứ hai, trong mã nguồn, chúng ta có thể theo dõi luồng ngữ nghĩa, biết được các biến liên kết với nhau và cách chúng di chuyển qua chương trình. Điều này khác với ngôn ngữ tự nhiên, nơi mà luồng ý nghĩa không rõ ràng như vậy.

Thứ ba, mã nguồn có thể thực thi được, cho phép chúng ta chạy và quan sát kết quả, điều mà ngôn ngữ tự nhiên không thể làm được. Cuối cùng, mã nguồn được tạo ra một cách tuần tự và thường xuyên được chỉnh sửa trong suốt quá trình phát triển phần mềm, trong khi văn bản ngôn ngữ tự nhiên thường chỉ được viết và chỉnh sửa một vài lần trước khi hoàn thiện.

Code Generation - Subtasks and Datasets

HumanEval (Chen et al. 2021)

Một trong những bộ dữ liệu nổi tiếng nhất hiện nay là HumanEval. Bộ dữ liệu này được đánh giá cao vì nhiều lý do. HumanEval chứa các ví dụ về việc sử dụng thư viện chuẩn Python, với

độ khó khác nhau. Ví dụ, một bài toán yêu cầu tính tổng các phần tử lẻ nằm ở vị trí chẵn trong một danh sách số nguyên không rỗng. Đây là dạng bài toán tương tự như trên LeetCode, có thể giải quyết bằng một dòng mã nhưng đòi hỏi sự suy nghĩ.

Tuy nhiên, HumanEval chỉ có 164 ví dụ, một con số khá nhỏ, và chỉ tập trung vào thư viện chuẩn Python, không kiểm tra việc sử dụng các thư viện khác. Điều này khiến nó không phải là thước đo thực tế nhất cho kỹ năng lập trình, tương tự như LeetCode. Dù vậy, nhiều công ty vẫn sử dụng nó, có thể vì tính hợp lý của nó trong một số trường hợp.

Bộ dữ liệu này thường bao gồm docstring, các ví dụ về đầu vào và đầu ra, cùng với các bài kiểm tra để xác minh độ chính xác của kết quả.

Metric: Pass@K (Chen et al. 2021)

Trong lĩnh vực đánh giá các hệ thống tạo mã, một chỉ số quan trọng được sử dụng là "pass@K". Ý tưởng cơ bản là tạo ra K ví dụ và kiểm tra xem ít nhất một trong số đó có vượt qua được các unit tests hay không. Ví dụ, với "pass@1", chúng ta đo lường khả năng một chương trình duy nhất vượt qua unit test. Với "pass@5", có thể hiển thị năm chương trình cho người dùng chọn hoặc kiểm tra xem chương trình nào vượt qua unit test để sử dụng.

Để giảm thiểu sự biến động khi chỉ tạo một ví dụ, người ta thường tạo ra nhiều đầu ra, chẳng hạn 10 hoặc 200, rồi tính toán số lượng kỳ vọng của các trường hợp vượt qua unit test. Ký hiệu K là giá trị "pass@K", n là tổng số đầu ra được tạo, và C là số câu trả lời đúng.

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

Các mô hình ngôn ngữ lớn (LLMs) thường được đánh giá bằng chỉ số "pass@K" vì chúng được huấn luyện với nhiều dữ liệu mã nguồn. Điều này không chỉ giúp LLMs có khả năng tạo mã tốt hơn mà còn cải thiện khả năng suy luận của chúng. Một ví dụ điển hình là tập dữ liệu "The Pile", trong đó gần một nửa là mã nguồn.

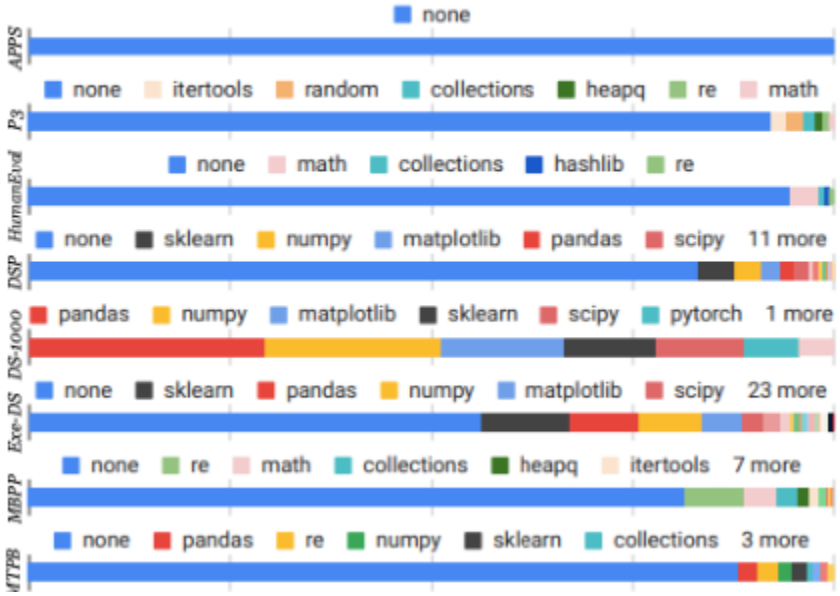
Broader Domains: CoNaLa/ODEX (Yin et al. 2018, Wang et al. 2022)

Một tập dữ liệu mở rộng hơn có tên là CoNaLa, được tạo ra từ dữ liệu thu thập từ Stack Overflow. Tập dữ liệu này bao gồm các câu hỏi và câu trả lời về cách giải quyết các vấn đề lập trình, không bị giới hạn trong thư viện chuẩn của Python. Điều này cho phép CoNaLa bao phủ một loạt các thư viện phổ biến, phản ánh đúng thực tế sử dụng của lập trình viên.

Một điểm nổi bật của CoNaLa là phương pháp đánh giá dựa trên thực thi, được gọi là ODEX. Trước đây, chúng tôi chỉ đánh giá dựa trên đoạn mã có thể giải quyết vấn đề, nhưng giờ đây, chúng tôi đã bổ sung khả năng thực thi các unit tests. Điều này giúp đánh giá chính xác hơn khả năng của mô hình trong việc sử dụng các thư viện khác nhau.

Biểu đồ phân phối thư viện của ODEX cho thấy khoảng một nửa số câu hỏi sử dụng các thư viện như Pandas, NumPy, và nhiều thư viện khác. Ngược lại, HumanEval - một tập dữ liệu khác - chủ yếu chỉ yêu cầu các thư viện chuẩn của Python. Điều này cho thấy CoNaLa và ODEX cung cấp một cái nhìn thực tế hơn về cách các truy vấn lập trình thực sự diễn ra, không giống như các bài tập trên LeetCode mà thường không sử dụng nhiều thư viện bên ngoài.

Figure 3: ODEX library distribution.



Metrics: BLUE, CodeBLUE

Trong quá trình phát triển CoNaLa, ban đầu các tác giả không sử dụng phương pháp đánh giá dựa trên thực thi do việc tạo ra các unit test cho nhiều bài viết trên Stack Overflow là rất khó khăn. Có hai vấn đề chính:

- Mã nguồn cần phải dễ dàng thực thi. Ví dụ, với thư viện Matplotlib, việc tạo một bài kiểm tra đơn vị để xác nhận rằng nó đã tạo ra biểu đồ thanh thành công là khá phức tạp. Bạn cần phải lấy hình ảnh và xác nhận rằng đó là biểu đồ thanh. Tương tự, với các framework như Django, việc xác nhận máy chủ hoạt động đúng cách cũng không đơn giản.
- Đánh giá dựa trên thực thi bỏ qua các yếu tố về phong cách viết mã. Mã có thể rất lộn xộn nhưng nếu thực thi đúng, nó vẫn được coi là chính xác, điều này đôi khi gây ra vấn đề.

Một số phương pháp thay thế bao gồm BLEU score, tính toán sự trùng lặp n-gram giữa mã chuẩn và mã do hệ thống tạo ra. Ngoài ra, có các phương pháp đánh giá mã như CodeBLEU, xem xét cả cú pháp và luồng ngữ nghĩa của mã. CodeBLEU đo lường sự trùng lặp giữa các chuỗi trong mã gốc và cả giữa các cây cú pháp và đồ thị luồng thông tin ngữ nghĩa.

Để có được cây cú pháp, ví dụ với Python, có thể sử dụng thư viện Abstract Syntax Tree (AST) của Python. Tuy nhiên, một nhược điểm của BLEU và CodeBLEU là chúng có thể đánh giá thấp các chương trình khác nhau về hình thức nhưng đều đúng. Do đó, kết hợp cả hai phương pháp có thể là lựa chọn phù hợp nếu có thể viết unit test.

Metric: CodeBERTScore (Zhou et al. 2023)

Tiếp theo, chúng ta sẽ tìm hiểu về CodeBERTScore, một chỉ số đánh giá mã nguồn dựa trên embedding được phát triển tại CMU. Trước đây, chúng ta đã biết đến BERTScore, một chỉ số tính toán độ tương đồng cosine giữa các token của văn bản sinh ra và văn bản tham chiếu. CodeBERTScore áp dụng nguyên tắc tương tự cho mã nguồn, tính toán độ tương đồng cosine giữa các token của mã nguồn tham chiếu và mã nguồn sinh ra.

Các tác giả đã phát triển một mô hình có tên là CoDIR, dựa trên BERT nhưng được tiếp tục huấn luyện trên một lượng lớn mã nguồn. Kết quả cho thấy, CodeBERTScore có sự tương quan tốt hơn với độ chính xác thực thi cuối cùng và đánh giá của con người về tính đúng đắn của mã nguồn. Điều này đặc biệt hữu ích trong các trường hợp không thể tạo ra unit test hoặc khi việc tạo ra chúng quá tốn kém.

Một điểm đáng chú ý là CodeBERTScore không xem xét cấu trúc mã nguồn, nhưng nó ít bị ảnh hưởng bởi sự khác biệt về tên biến so với các chỉ số khác. Ví dụ, CoDIR có thể cho ra các biểu diễn tương tự cho các biến "i" và "j" vì chúng thường được sử dụng làm biến lặp, trong khi mô hình BERT thông thường có thể cho ra các biểu diễn rất khác nhau do "i" là đại từ nhân xưng và "j" thì không. Điều này giải thích tại sao việc tiếp tục huấn luyện trên mã nguồn lại có ích.

Data Science Notebooks: ARCADE (Yin et al. 2022)

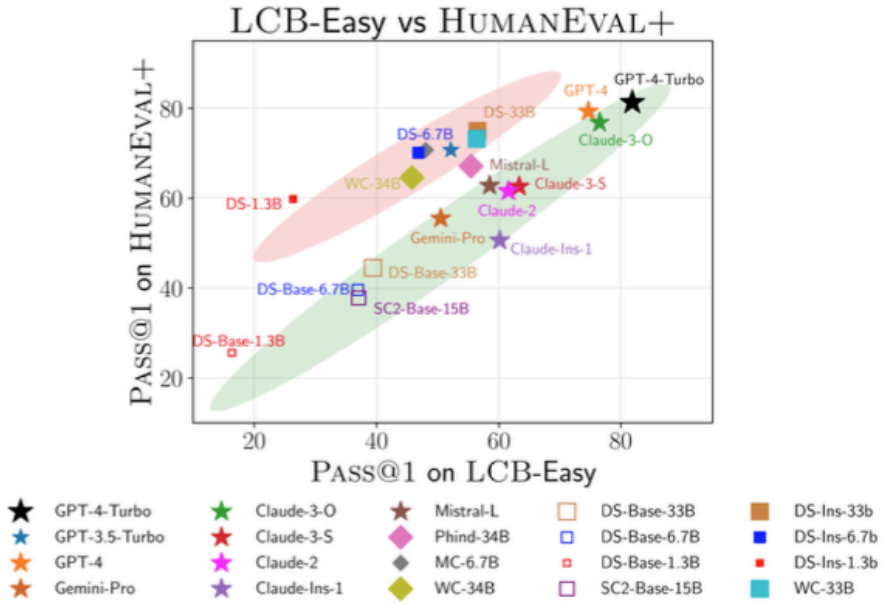
Một ứng dụng hữu ích của việc sinh mã là trong các notebooks dành cho khoa học dữ liệu, chẳng hạn như Google Colab. Các notebook này cho phép triển khai mã một cách từng bước, điều mà hầu hết mọi người đều sử dụng. Một điểm thú vị khác là khả năng đánh giá việc sinh mã trong ngữ cảnh hoặc sinh mã từng bước. Bạn có thể bắt đầu với một notebook, sử dụng ngôn ngữ AAL (Affective Action Language) để tạo ra các lệnh và sau đó sinh ra kết quả tương ứng. Đây là một ví dụ từ bộ dữ liệu STA. Bài báo này của Google rất đáng chú ý vì nó cung cấp một bộ dữ liệu phong phú và hữu ích.

An Aside: Dataset Leakage

Trong một nghiên cứu gần đây, vấn đề rò rỉ dữ liệu đã được nhấn mạnh khi đánh giá các mô hình ngôn ngữ, đặc biệt là trong lĩnh vực sinh mã. Nghiên cứu này đã chỉ ra sự khác biệt rõ rệt giữa các notebook mã hóa được lấy từ web và những notebook do nhóm nghiên cứu tự tạo ra. Các mô hình sinh mã, ngoại trừ PaLM - mô hình tốt nhất trong bộ dữ liệu này, thường hoạt động kém trên dữ liệu mới so với dữ liệu có sẵn, cho thấy khả năng dữ liệu đã bị rò rỉ vào tập huấn luyện.

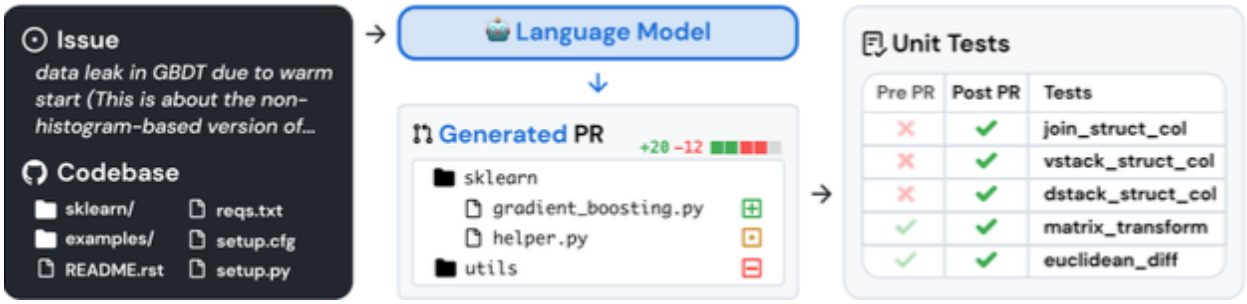
Một nghiên cứu khác gần đây, có thể xuất bản vào năm 2024, đã thực hiện đánh giá tương tự trên các bài toán từ LeetCode và các trang web khác. Kết quả cho thấy độ chính xác cao trên các bài toán cũ trước thời điểm cắt huấn luyện, nhưng giảm mạnh sau đó. Điều này cho thấy sự rò rỉ dữ liệu có thể đã xảy ra.

Biểu đồ trong nghiên cứu này minh họa mối tương quan giữa việc vượt qua các bài kiểm tra trên LiveCodeBench và HumanEval. Một nhóm mô hình có điểm số cao hơn trên HumanEval so với dự kiến, cho thấy khả năng dữ liệu từ HumanEval đã bị rò rỉ vào tập huấn luyện. Tương tự, vấn đề này cũng xuất hiện trong các bài toán suy luận toán học.



Dataset: SWEBench (Jiminez et al. 2023)

Tiếp theo, tôi muốn giới thiệu về một bộ dữ liệu thú vị có tên là "SWEBench". Gần đây, nó đã thu hút sự chú ý khi được sử dụng để đánh giá một assistant code mới có tên là Devon. SWEBench lấy các vấn đề từ GitHub và các mã nguồn làm đầu vào, sau đó tạo ra một pull request để giải quyết các vấn đề này. Ví dụ, đầu vào có thể là "data leak in GBDT due to warm start", và hệ thống sẽ tạo ra một PR để giải quyết vấn đề đó. PR này sau đó được kiểm tra qua các unit test để đảm bảo nó vượt qua tất cả các unit test sau khi PR được tạo.



Quá trình này tương tự như những gì bạn thường làm trong một dự án phần mềm lớn: mở một vấn đề và sau đó mở một PR để sửa vấn đề đó. Điều này đòi hỏi khả năng hiểu ngữ cảnh dài và thực hiện các triển khai chính xác dựa trên các dự án phần mềm lớn. Hiện tại, phương pháp tiên tiến nhất cho bài toán này chỉ đạt khoảng 14%, cho thấy đây vẫn là một thách thức lớn. Trong bài báo gốc, phương pháp tiên tiến nhất chỉ đạt khoảng 6%.

Tôi không kỳ vọng chúng ta sẽ đạt được 90% trong thời gian sớm, vì có lẽ hệ thống chỉ đang giải quyết những vấn đề dễ hơn, trong khi những vấn đề khó hơn vẫn vượt quá khả năng của bất kỳ mô hình ngôn ngữ nào hiện nay. Tuy nhiên, tôi rất thích benchmark này. Một lưu ý nhỏ là việc chạy benchmark này khá nặng, vì bạn cần tải về toàn bộ các kho mã nguồn để thực hiện, do đó cần cẩn thận khi sử dụng.

Dataset: Design2Code (Si et al. 2024)

Một bộ dữ liệu đáng chú ý là "Design2Code", một bộ dữ liệu rất mới. Ý tưởng chính là tạo mã từ các trang web, tức là đầu vào là một trang web và đầu ra là mã JavaScript, CSS hoặc HTML để triển khai trang web đó. Tôi thực sự thích bộ dữ liệu này vì nó là một nền tảng thử nghiệm tốt cho các mô hình đa phương tiện. Hiện tại, chưa có nhiều mô hình mã nguồn mở mạnh mẽ có thể giải quyết tốt vấn đề này. Họ cũng đã đề xuất một mô hình "Design2Code" đạt kết quả tốt nhất trên bộ dữ liệu này so với các mô hình mã nguồn mở khác, nhưng vẫn còn nhiều thách thức cần vượt qua.

Metric: Visual Similarity of Website

Câu hỏi đặt ra là làm thế nào để đánh giá điều này ngay từ đầu. Ý tưởng cơ bản là họ sử dụng sự tương đồng hình ảnh cao và tính toán các embedding hình ảnh của các trang web được tạo ra. Họ cũng xem xét sự tương đồng của các yếu tố, cố gắng xác định tất cả các thành phần trong trang web đã tạo và đảm bảo rằng tất cả các yếu tố đó đều được ghi nhận.

Một điều đáng chú ý là ngay cả những mô hình tiên tiến như Claude 3 hay GPT-4 cũng gặp khó khăn trong việc này. Chúng có thể tạo ra thứ gì đó trông có vẻ tương tự, nhưng thường thiếu các yếu tố cần thiết và thiết kế không chính xác.

Vậy tại sao đây lại là một vấn đề khó khăn cho các mô hình? Tôi không có câu trả lời chắc chắn, nhưng có thể nói rằng chúng có khả năng cải thiện. Chúng có thể tạo ra một sản phẩm và nếu bạn phản hồi rằng nó không tốt, chúng có thể làm tốt hơn trong lần tạo tiếp theo, đặc biệt khi bạn đưa ra những yêu cầu cụ thể.

Nếu bạn nghĩ về một lập trình viên con người có kỹ năng, liệu họ có thể viết mã cho một trang web và chỉ xem một lần mà đã đúng không? Có lẽ là không. Chúng ta đang yêu cầu các mô hình làm điều tương tự, nhưng chúng còn kém hơn cả chúng ta trong việc theo dõi tất cả các yếu tố hình ảnh. Có lẽ vấn đề này cần phải được cải thiện qua nhiều lần lặp lại, nếu không thì yêu cầu đặt ra cho mô hình là quá lớn.

Code Generation - Methods

Basic method: Code generating LM

Trong lĩnh vực tạo mã tự động, các mô hình ngôn ngữ đóng vai trò quan trọng. Bạn có thể cung cấp mã trước đó hoặc ngữ cảnh liên quan vào mô hình để tạo ra mã mới. Hiện nay, hầu hết các mô hình ngôn ngữ lớn đều được huấn luyện với dữ liệu mã nguồn. Một điểm quan trọng khi tạo mã là điều chỉnh cài đặt temperature của mô hình. Nên đặt temperature ở mức thấp để tránh việc tạo ra mã không hợp lý và đảm bảo mã được tạo ra có chất lượng tốt hơn.

Code Infilling (Fried et al. 2022)

Một khả năng cốt lõi của các mô hình ngôn ngữ mã hóa (code language models), đặc biệt là những mô hình được sử dụng trong môi trường phát triển tích hợp (IDE) như Co-pilot, là khả năng điền vào chỗ trống (infilling). Theo một nghiên cứu của Daniel Freed tại LTI, quá trình này thường bắt đầu bằng việc sử dụng mô hình ngôn ngữ từ trái sang phải (left-to-right LM) tiêu chuẩn để điền vào đoạn mã bị thiếu. Cụ thể, khi bạn muốn thêm một dòng mã như một câu lệnh "if" hoặc một sự sửa đổi nào đó, bạn sẽ đặt một "mask" tại vị trí cần điền, thường là nơi con trỏ của bạn đang đứng trong IDE. Sau đó, bạn di chuyển "mask" này đến cuối chuỗi và sử dụng mô hình ngôn ngữ tự hồi quy (auto-regressive language model) để tạo ra đoạn mã cần thiết. Kỹ thuật này rất quan trọng nếu bạn muốn xây dựng một công cụ giống như Co-pilot, và hầu hết các mô hình ngôn ngữ mã hóa hiện nay đều áp dụng phương pháp này.

Lots of Available Information for Coding!

Trong lĩnh vực học lập trình và giải quyết các tác vụ mã hóa, có rất nhiều thông tin sẵn có để hỗ trợ. Nó bao gồm ngữ cảnh mã hiện tại, mô tả vấn đề cần giải quyết, ngữ cảnh từ các tệp khác trong repo, và cả những tab bạn đang mở.

Khi GitHub Copilot ra mắt, họ không tiết lộ chi tiết về cách hoạt động của nó. Tuy nhiên, một sinh viên (có thể là từ Georgia Tech) đã giải mã JavaScript của Copilot và viết một bài blog rất thú vị (Thakkar 2023) về cách thức hoạt động của nó.

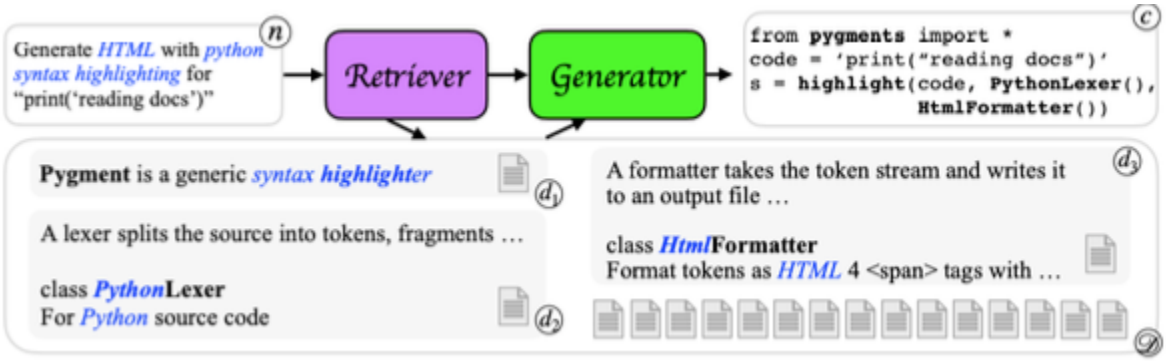
Copilot thực hiện việc trích xuất thông tin cho prompt dựa trên tài liệu hiện tại và vị trí con trỏ. Nó xác định đường dẫn tương đối của tệp và ngôn ngữ lập trình đang sử dụng, như Python hay JavaScript. Copilot theo dõi 20 tệp gần nhất được truy cập trong cùng ngôn ngữ, và tạo prompt bao gồm văn bản trước và sau vị trí con trỏ, các tệp tương tự trong số 20 tệp đã mở gần đây, thông tin từ các tệp được nhập và metadata về ngôn ngữ và đường dẫn.

Tất cả thông tin này được gửi đến mô hình, thể hiện một kỹ thuật "prompt engineering" rất tốt, nhằm tối ưu hóa thông tin cần thiết để mô hình hoạt động hiệu quả. Bài blog này cung cấp nhiều thông tin chi tiết hơn và rất đáng để tham khảo nếu bạn quan tâm.

Retrieval-based Code Generation

Một phương pháp hữu ích là sử dụng "tạo mã dựa trên truy xuất" (retrieval-based code generation). Phương pháp này tương tự như RAG nhưng áp dụng cho mã nguồn. Nó cho phép truy xuất mã tương tự từ các nguồn trực tuyến và sử dụng mã này để hỗ trợ mô hình ngôn ngữ trong việc tạo mã mới. Điều này đặc biệt hữu ích khi bạn có một mô hình không mạnh về mã nguồn hoặc khi bạn có một cơ sở mã lớn mà mô hình chưa được huấn luyện trên đó, chẳng hạn như trong các công ty lớn với phong cách viết mã riêng.

Ngoài ra, tài liệu cũng có thể được truy xuất để hỗ trợ quá trình này. Một vấn đề phổ biến khi sử dụng các công cụ như ChatGPT hoặc Claude là chúng có thể sử dụng các phiên bản thư viện cũ không còn tương thích. Để giải quyết vấn đề này, một phương pháp gọi là "DOC prompting" đã được đề xuất. Phương pháp này tìm kiếm tài liệu tương tự và thêm vào prompt để tạo ra mã phù hợp. Điều này đặc biệt hữu ích khi làm việc với các thư viện mới hoặc đã được cập nhật.



Execution Feedback (Shi et al. 2022)

Trong lĩnh vực lập trình, một trong những lợi ích của việc sử dụng mã so với ngôn ngữ tự nhiên là khả năng nhận phản hồi từ việc thực thi mã. Một nghiên cứu đã chỉ ra rằng có thể tạo ra nhiều loại mã khác nhau bằng cách lấy mẫu từ các biến thể mã. Cụ thể, nếu bạn có các unit tests, bạn có thể xác định mã nào là chính xác bằng cách chạy các unit test này. Tuy nhiên, trong trường hợp không thể thực hiện điều đó, một phương pháp khả thi là thực thi tất cả các đoạn mã mà mô hình đã tạo ra và kiểm tra xem các đầu ra có trùng lặp hay không.

Nếu có nhiều chương trình (ví dụ: 30 chương trình) tạo ra các đầu ra tương tự nhau, điều đó cho thấy chương trình đó có khả năng đúng. Trong nghiên cứu này, các nhà nghiên cứu đã chọn chương trình có rủi ro cơ bản thấp nhất, dựa trên khái niệm “minimum bayes risk”. Họ thực hiện nhiều lần và tính toán rủi ro cơ bản của các đoạn mã.

Khái niệm “self-consistency” là một dạng của rủi ro cơ bản, vì đầu ra có thể không hoàn toàn giống nhau nhưng gần nhau thì có thể được coi là tốt hơn. Việc so sánh mã có thể được thực hiện bằng cách sử dụng các công cụ như “code BT score”, mặc dù trong nghiên cứu này, các nhà nghiên cứu không sử dụng phương pháp đó.

Fixing Based on Error Messages

Một khía cạnh thú vị là việc sửa lỗi dựa trên thông điệp lỗi. Ý tưởng cơ bản là bạn tạo ra mã, chạy thử và nhận thông điệp lỗi, sau đó cung cấp thông tin này cho mô hình ngôn ngữ lớn để sửa chữa lỗi. Các LLMs có khả năng nhất định trong việc này, đặc biệt là khi chúng được cải tiến qua thời gian.

Một nghiên cứu gần đây, được đề cập trong bài báo "Intercode" (Yang et al. 2023), đã mở rộng khái niệm này. Nghiên cứu cho thấy rằng không chỉ có thể thực hiện việc tạo mã trong một lần tương tác, mà còn có thể yêu cầu mô hình thử lại hoặc thực hiện lập kế hoạch và giải quyết vấn đề. Điều này tạo ra một môi trường tốt cho việc phát triển các trợ lý lập trình tương tác, giúp người dùng có trải nghiệm lập trình hiệu quả hơn.

Code Synthesize from Input/ Output Examples

Tổng hợp mã (code synthesis) từ các cặp input/output là một chủ đề quan trọng trong nghiên cứu, có lịch sử phát triển lâu dài hơn so với tổng hợp mã từ ngôn ngữ tự nhiên. Phương pháp này không cần đến mô tả bằng ngôn ngữ tự nhiên mà chỉ dựa vào các ví dụ về đầu vào và đầu ra để suy luận ra chương trình.

Một ví dụ điển hình là tính năng Flash Fill trong Microsoft Excel. Ví dụ, khi có một cột chứa các chuỗi như "R new big" và cần chuyển đổi thành "Rig", hệ thống sẽ tự động suy luận ra quy tắc: "lấy ký tự đầu tiên của từ thứ nhất và hai ký tự của từ cuối cùng, sau đó ghép lại". Tính năng này đặc biệt hữu ích khi xử lý các bảng tính lớn.

Hiện nay, hầu hết các nghiên cứu về tổng hợp mã tập trung vào các ngôn ngữ chuyên biệt (domain-specific languages) như RegEx hay SQL, thay vì các ngôn ngữ lập trình đa năng. Điều này là do khi không có mô tả bằng ngôn ngữ tự nhiên, việc tìm kiếm giải pháp trở nên phức tạp hơn, đòi hỏi phải thu hẹp không gian tìm kiếm. Những người quan tâm có thể tham khảo thêm từ các công trình nghiên cứu từ nhóm của Joshua Tenenbaum tại MIT về chủ đề này.

Code Generation - Representative Code LMs

Codex (Chen et al. 2022)

Trong phần cuối cùng của bài viết, tôi muốn đề cập đến các mô hình ngôn ngữ chuyên về mã nguồn, đặc biệt là Codex. Codex là mô hình đầu tiên và có ảnh hưởng lớn trong lĩnh vực này, được phát triển bởi OpenAI. Ban đầu, Codex được huấn luyện bằng cách tiếp tục đào tạo từ GPT-3, sử dụng một lượng lớn dữ liệu mã từ GitHub. Nếu bạn đã từng đóng góp mã trên GitHub, có thể bạn đã gián tiếp hỗ trợ cho Codex. Mô hình này hiện vẫn cung cấp sức mạnh cho GitHub Copilot.

Một điểm thú vị là OpenAI đã phát triển hai phiên bản của Codex: một phiên bản lớn và một phiên bản nhỏ hơn gọi là Code Cushman. Phiên bản nhỏ hơn, Code Cushman, thực sự là mô hình đang được sử dụng cho GitHub Copilot. Lý do cho sự lựa chọn này là vì nó cung cấp phản

hồi nhanh hơn, điều này rất quan trọng khi làm việc với mã. Hơn nữa, việc sử dụng mô hình lớn hơn như DaVinci sẽ tốn kém hơn rất nhiều, đặc biệt khi phải xử lý nhiều thay đổi trong mã nguồn. Mặc dù là mô hình nhỏ hơn, Code Cushman vẫn rất hiệu quả và đáp ứng tốt nhu cầu của người dùng.

StarCoder 2 (Lozhkov et al. 2024)

Tiếp theo, chúng ta sẽ khám phá một số mô hình hiện đại, bắt đầu với StarCoder 2. Lý do tôi muốn đề cập đến mô hình này trước tiên không phải vì nó là mô hình tốt nhất, mặc dù nó rất xuất sắc, mà vì nó cung cấp thông tin chi tiết về dữ liệu và quy trình huấn luyện. StarCoder 2 được phát triển bởi dự án Big Science, do Hugging Face và Service Now dẫn đầu, với sự tham gia của nhiều trường đại học.

Về kiến trúc, StarCoder 2 chủ yếu theo phong cách LLaMA và có các biến thể 3B, 7B và 15B. Một điểm thú vị của các mô hình ngôn ngữ mã hóa (code LM) là khả năng xử lý ngữ cảnh dài, nhờ việc điều chỉnh tham số Theta trong RoPE để mở rộng ngữ cảnh.

Phần dữ liệu huấn luyện của mô hình này rất đáng chú ý. Nó được huấn luyện trên mã nguồn từ nhiều nguồn như GitHub, pull request, Jupyter notebooks, Kaggle notebooks, tài liệu và các biểu diễn trung gian từ LLVM. Ngoài ra, nó còn được huấn luyện trên một số tập dữ liệu ngôn ngữ tự nhiên liên quan đến viết mã.

Trong quá trình tiền xử lý, họ thêm các thẻ metadata như tên repo và tên tệp 50% thời gian để mô hình có thể hoạt động tốt cả khi có và không có chúng. Mô hình cũng được huấn luyện dựa trên việc điền vào chỗ trống (infilling). Đặc biệt, StarCoder 2 được huấn luyện trong 4-5 epoch, nhiều hơn so với thông thường chỉ một epoch, do lượng dữ liệu mã hóa ít hơn so với dữ liệu ngôn ngữ tự nhiên.

The Stack 2

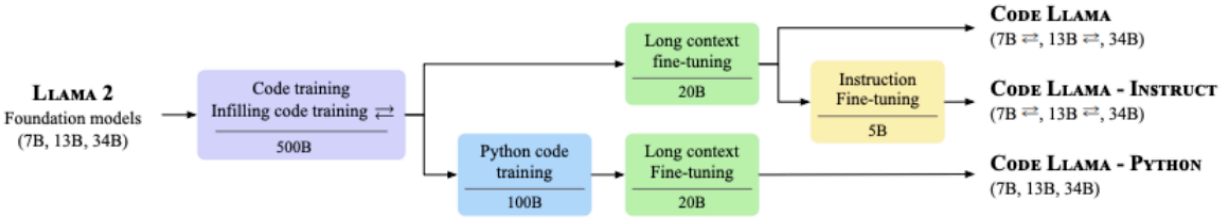
Một tập dữ liệu mới có tên là "The Stack 2", được sử dụng để tiền huấn luyện các mô hình ngôn ngữ mã nguồn. Một điểm thú vị mà nhóm nghiên cứu đã cân nhắc là vấn đề giấy phép. Như đã thảo luận trong buổi học trước, bản quyền là một thách thức lớn khi huấn luyện các mô hình ngôn ngữ lớn. Để giải quyết vấn đề này, nhóm đã tìm kiếm các mã nguồn có giấy phép cho phép trên GitHub. Nếu giấy phép trên GitHub là cho phép, họ đánh dấu dữ liệu đó là cho phép. Họ cũng cố gắng phát hiện các giấy phép và nếu tất cả đều cho phép, dữ liệu được đánh dấu là cho phép.

Trong bài báo, nhóm nghiên cứu đã trình bày một bảng lớn liệt kê tất cả các dữ liệu mà họ thu thập được. Một số ngôn ngữ lập trình có lượng dữ liệu lớn nhất bao gồm Java, PHP, Markdown, và Python. Ngoài ra, còn có nhiều ngôn ngữ lập trình khác, bao gồm cả những ngôn ngữ ít phổ biến hơn như Rust. Điều này cho thấy rằng, dù bạn yêu thích ngôn ngữ lập trình nào, từ phổ biến đến ít người biết đến, đều có dữ liệu sẵn sàng để sử dụng.

CodeLLaMA (Roziere et al. 2023)

CodeLlaMA, một mô hình cạnh tranh được phát triển bởi Meta. CodeLlaMA có kiến trúc tương tự như LlaMa2 và được tiếp tục huấn luyện từ LlaMa2. Điểm nổi bật của CodeLlaMA là khả năng xử lý ngữ cảnh đầu vào dài hơn nhờ mở rộng độ dài của "RoPE", những yếu tố quan trọng cho các mô hình ngôn ngữ mã nguồn.

CodeLlaMA được huấn luyện trên mã nguồn từ DED và dữ liệu hướng dẫn được tạo ra một cách tổng hợp, đặc biệt dành cho mã nguồn. Quá trình huấn luyện diễn ra theo từng giai đoạn với nhiều tập dữ liệu khác nhau. Đầu tiên, mô hình được huấn luyện trên 500 tỷ token mã nguồn, sau đó là tinh chỉnh ngữ cảnh dài trên 20 tỷ token, và cuối cùng là tinh chỉnh theo hướng dẫn.



Ngoài ra, CodeLlaMA còn có phiên bản chuyên biệt cho Python. Lý do không phải vì Python quan trọng hơn, mà vì nhiều tiêu chuẩn đánh giá hiện nay sử dụng Python, do các nhà nghiên cứu máy học thường ưa chuộng ngôn ngữ này. Do đó, phiên bản Python của CodeLlaMA được tối ưu hóa để đạt kết quả tốt trên các tiêu chuẩn đánh giá này.

DeepSeek Coder (Guo et al. 2024)

Cuối cùng, tôi muốn giới thiệu về mô hình DeepSeekCoder. Đây là một mô hình nổi bật vì độ mạnh mẽ của nó, có thể được coi là một trong những mô hình mạnh nhất trong số các mô hình mã hóa hiện nay. DeepSeekCoder được huấn luyện với 87% dữ liệu là mã nguồn, 10% là tiếng Anh từ các tài liệu markdown trên StackExchange, và 3% là tiếng Trung, do công ty DeepSeek của Trung Quốc phát triển.

Một điểm thú vị trong quá trình huấn luyện của mô hình này là việc bao gồm các thư viện phụ thuộc. Họ đã thu thập đồ thị phụ thuộc của các thư viện, lấy các tệp từ các thư viện được tham chiếu và sử dụng chúng trong quá trình huấn luyện. Điều này đặc biệt hữu ích khi bạn muốn mô hình có khả năng tham chiếu đến các thư viện bên ngoài.

Về kiến trúc, DeepSeekCoder khá tiêu chuẩn, tương tự như LLaMA, với các biến thể 1.3 tỷ, 6.7 tỷ và 33 tỷ tham số. Mô hình này được huấn luyện trên hai nghìn tỷ token.

Which to use?

Trong lĩnh vực lựa chọn mô hình mã hóa, câu hỏi đặt ra là nên sử dụng mô hình nào. Theo nghiên cứu trong bài báo StarCoder, các mô hình như DeepCoder và StarCoder có hiệu suất tương đối tương đồng. DeepCoder mạnh về các tác vụ lập trình tiêu chuẩn, trong khi StarCoder

lại nổi bật trong việc xử lý các notebook khoa học dữ liệu. Tuy nhiên, cả hai đều chưa đạt đến mức độ phức tạp như GPT-4.

Một điểm cần lưu ý với DeepCoder là nhiều mô hình trong nghiên cứu có thể đã được huấn luyện trên dữ liệu tương tự với bộ đánh giá HumanEval, do đó cần thận trọng khi diễn giải kết quả. Dù vậy, DeepCoder vẫn thể hiện tốt trên các bộ dữ liệu khác mà mô hình chưa từng thấy, chứng tỏ đây là một trong những mô hình mã hóa cạnh tranh nhất hiện nay, đặc biệt trên bộ dữ liệu mới LCB.

Discussion Question

Một câu hỏi thú vị đã được đặt ra về cách áp dụng các ràng buộc cú pháp trong quá trình giải mã mà không chỉ dựa vào xác suất của mô hình. Đối với mã nguồn, việc này không phải lúc nào cũng rõ ràng. Một cách đơn giản là tạo ra nhiều kết quả và loại bỏ những kết quả sai cú pháp. Tuy nhiên, nếu muốn thực hiện điều này ngay trong quá trình giải mã, cần có một bộ phân tích cú pháp gia tăng để loại bỏ các giả thuyết sai dần dần. Điều này dễ thực hiện với một số ngôn ngữ nhưng khó khăn hơn với các ngôn ngữ khác.

Hiện nay, JSON là một định dạng phổ biến mà nhiều người muốn tạo ra và sử dụng trong các tác vụ tiếp theo. Có một số thư viện hỗ trợ việc này, chẳng hạn như "outlines", cho phép tích hợp các ràng buộc cú pháp thông qua các cấu trúc như “weighted finite State automata”. Một thư viện khác là "guidance", phức tạp hơn nhưng cũng rất hữu ích cho việc tạo ra các đầu ra có ràng buộc. Cả hai thư viện này cung cấp các phương pháp khác nhau để thêm ràng buộc vào đầu ra.

Resources

1. https://phontron.com/class/anlp2024/lectures/#code-generation-march-19
2. Github Copilot
3. Github Copilot and Productivity
4. HumanEval (Chen et al. 2021)
5. CoNaLa (Yin et al. 2018)
6. ODEX (Wang et al. 2022)
7. CodeBLEU (Ren et al. 2020)
8. Design2Code (Si et al. 2024)
9. CodeBERTScore (Zhou et al. 2023)
10. ARCADE (Yin et al. 2022)
11. LiveCodeBench (Jain et al. 2024)
12. SWEBench (Jiminez et al. 2023)
13. InCoder (Fried et al. 2022)
14. Copilot Explorer (Thakkar 2023)
15. Retrieval-based Code Generation (Hayati et al. 2018)
16. DocPrompting (Zhou et al. 2022)
17. Code Generation w/ Execution (Shi et al. 2022)

18. InterCode (Yang et al. 2023)
19. Flashfill (Gulwani 2011)
20. Terpret (Gaunt et al. 2016)
21. CodeLLaMa (Roziere et al. 2023)
22. DeepSeek Coder (Guo et al. 2024)
23. StarCoder 2 (Lozhkov et al. 2024)