Thesis Proposal

External Knowledge Augmented Language Models for Code Generation and Agents

Fangzheng (Frank) Xu

January 12, 2024

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15123

Thesis Committee:

Graham Neubig (Chair)	Carnegie Mellon University
Daniel Fried	Carnegie Mellon University
Bogdan Vasilescu	Carnegie Mellon University
Karthik Narasimhan	Princeton University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: external knowledge, language models, LLM agents, code generation

Abstract

We interact with computers everyday. Although the user experience of operating a computer has never been easier nowadays, there are still hurdles that prevent users from making use of their full potential. One such example is the learning curve of programming languages. Programmers have always dreamt of easier and more intelligent tools to assist them with their work, and thus make developing more effortless. End users that do not know how to program would also benefit from a more natural way of instructing computers to accomplish certain customized tasks, *i.e.*, transforming computers into their personal agents that complete various tasks with high level of autonomy and instruction-following ability. This sharply contrasts to how effortlessly we communicate our goals and desires in natural language such as English.

In this thesis, we propose to bridge natural language with programming language and executable actions in daily life online environments. The anticipated outcomes of this work aim to construct models, environments and evaluations for both code generation tasks for programming jobs and large language model (LLM) agents for online daily life, with a special focus on utilizing large data sources and external knowledge for model pre-training and retrieval-augmented models. We examine the problem from four perspectives that constitute the four parts of the dissertation. In the first part we explore pre-training for code generation models. Next, in the second part we perform human study of code generation. Then, in the third part we improve retrieval-augmented models. Finally, In the fourth part we explore interactive use of LLMs as agents.

Contents

Chapter 1

Introduction

We interact with computers everyday. Although the user experience of operating a computer has never been easier nowadays, there are still hurdles that prevent users from making use of computers' full potential. One such example is the learning curve of programming languages [300]. Programmers have always dreamt of easier and more intelligent tools to assist them with their work, and thus make developing more effortless. End users that do not know how to program would also benefit from a more natural way (sometimes called "natural programming" [245]) of instructing computers to accomplish certain customized tasks [202, 264], *e.g.*, helping financial specialists with complex data analysis, enabling users to use computers as natural language personal assistants for online activities like shopping, or improve the accessibility of the current user interface via natural language instructions [196, 315]. This sharply contrasts to how effortlessly we communicate our goals and desires in natural language such as English.

Despite early skepticism towards the idea of "natural language programming" [74], researchers now widely agree on a range of scenarios where it can be useful to be able to formulate instructions using natural language and have the corresponding source code snippets automatically produced. For example, software developers can save keystrokes or avoid writing dull pieces of code [88, 250, 296, 359]; and non-programmers and practitioners in other fields, who require computation in their daily work, can get help with creating data manipulation scripts [107, 187]. Students can have access to more advanced tutoring systems that allow interaction in natural language [327], or instructors can get help grading programming assignments [272, 278]. This motivation started early research in constructing natural language interface [103, 127, 258] to computers to provide better human-computer interaction than graphical user interface. There were also lines of research for creating natural language interface to database so that end users without SQL and database schema knowledge could query the database and do data analysis

with natural language commands [16, 128, 145, 404]. In this thesis, we examine the problem from four perspectives: 1) pre-training for code generation (Part I), 2) human study of code generation (Part II), 3) study of retrieval-augmented models (Part III) and 4) interactive use of LLMs as agents (Part IV).

1.1 Pre-training for Code Generation

Pretraining for code generation models. Coding is among one of the most important ways of interacting with computer systems. My first line of work is closely aligned with both **natural language** and **software engineering**, and the **bridging** between the two. The eventual goal is to enable machine learning models to capture the procedural and intent semantics of the natural language commands, to understand the semantics and structures of programming languages, and to bridge the two drastically different domains of languages together, so that users could use natural language to instruct the computers in the future. A great part of software development involves conceptualizing or communicating the underlying procedures and logic that needs to be expressed in programs. One major difficulty of programming is turning *concept* into *code*, especially when dealing with the APIs of unfamiliar libraries. A key application at the intersection of the two domains is natural language to code generation, where a user gives a natural language intent and the model generates a code snippet that satisfies the user need.

When I first started the thesis in 2019, one of the main research focus in the field, semantic parsing, the task of generating machine executable meaning representations from natural language (NL) intents, has generally focused on limited domains [67, 395], or domain-specific languages with a limited set of operators [32, 76, 179, 205, 282, 389, 390, 391, 404]. However, recently there has been a move towards applying semantic parsing to automatically generating source code in general-purpose programming languages [2, 209, 377, 380, 388]. Automatically generating general domain programs given natural language is considered more generalized semantic parsing. Prior work in this area [77, 140, 211, 283, 335, 363, 381, 384, 386] used a variety of models, especially neural architectures, to achieve good performance. Many of these existing efforts in NL to code generation involve using large amounts of annotated data to train sequence-to-sequence models in a supervised fashion. Recognizing that such parallel data is costly, and motivated by the intuition that developers usually retrieve resources on the web when writing code, we proposed a method of incorporating two varieties of external knowledge into NL-to-code generation (Chapter 2): automatically mined NL-code pairs from the online programming QA forum StackOverflow and programming language API documentation [364].

We first pretrained the model on noisy mined data and then finetune on clean annotated data. The previous best model already can generate basic functions and copy strings/variables to the output, but we observed that incorporating external knowledge improves the results in two main ways: 1) better argument placement for APIs, and 2) better selection of which API call should be used for a certain intent. We can also see that the NL to code generation task is still challenging, especially with more complex intents that require nested or chained API calls, or functions with more arguments. We were also among the first to utilize pretraining to improve code models.

Era of large language models. Previously we focus on improving sequence-to-sequence task-specific models trained in supervised fashion, and now we explore a more general form of using language models to represent and generate code. As time went on, large language models (LMs) of code have recently shown tremendous promise in completing code and synthesizing code from natural language descriptions. Language models (LMs) assign probabilities to sequences of tokens, and are widely applied to natural language text [23, 31, 47]. Around 2021, LMs have shown impressive performance in modeling also source code, written in programming languages [11, 125, 131, 158]. These models excel at useful downstream tasks like code completion [296] and synthesizing code from natural language descriptions [72]. The current state-of-the-art large language models for code, such as Austin et al. [20], have shown significant progress for AI-based programming assistance. Most notably, one of the largest of these models, Codex [55] has been deployed in the real-world production tool GitHub Copilot[1], as an in-IDE developer assistant that automatically generates code based on the user's context.

Despite the great success of large language models of code, *the strongest models are not publicly available*. This prevents the application of these models outside of well-resourced companies and limits research in this field for low-resourced organizations. For example, Codex provides non-free access to the model's *output* through black-box API calls,[2] but the model's weights and training data are unavailable. This prevents researchers from fine-tuning and adapting this model to domains and tasks other than code completion. The lack of access to the model's internals also prevents the research community from studying other key aspects of these models, such as interpretability, distillation of the model for more efficient deployment, and incorporating additional components such as retrieval.

Several medium to large-sized pre-trained language models were publicly available at the

[1] https://copilot.github.com/
[2] https://openai.com/blog/openai-codex/

time, such as GPT-Neo [35], GPT-J [349] and GPT-NeoX [36]. Given the variety of model sizes and training schemes involved in these models and lack of comparisons between these, the impact of many modeling and training design decisions remains unclear. We presented a first systematic evaluation of existing models of code – Codex, GPT-J, GPT-Neo, GPT-NeoX, and CodeParrot – across various programming languages (Chapter 3). We aim to shed more light on the landscape of code modeling design decisions by comparing and contrasting these models, as well as providing a key missing link: thus far, no large open-source language model was trained exclusively on code from *multiple programming languages*. We provide three such models, ranging from 160M to 2.7B parameters, which we release under the umbrella name "PolyCoder", which at the time were the largest open-source code-specific pretrained language models. We perform an extensive comparison of the training and evaluation settings between PolyCoder, open-source models, and Codex. We evaluate the models on the HumanEval benchmark [55] and compare how do models of different sizes and training steps scale, and how different temperatures affect the generation quality. Finally, since HumanEval only evaluates the natural language to Python synthesis, we curate an unseen evaluation dataset in each of the 12 languages, to evaluate the perplexity of different models.

Although most open models at the time performed worse than OpenAI's Codex, we hope that this systematic study helps future research in this area to design more efficient and effective models. More importantly, through this systematic evaluation of different models, we encouraged the community to study and release medium-large scale language models for code. We believed that our efforts were a significant step towards democratization of large language models of code. Today, we have seen this goal come true, as more and more open-source large language models have come out, many including code understanding and generation ability, *e.g.*, LLAMA [339], CodeGen [254, 255], SantaCoder [4], StarCoder [198], Code Llama [307].

1.2 Human Study of Code Generation Models

From previous work, including that described above, we see improvements of code generation or retrieval performance on benchmark datasets over the time. However, these have primarily been evaluated purely based on retrieval accuracy or surface form overlap of generated code with developer-written code. For example BLEU score, which is a standard metric on some code generation tasks [387], is not necessarily a good proxy of the quality of generated code, as it only captures the surface token-based similarity with the ground truth, without considering the syntax, semantics, and runtime correctness of the code. For example, an empirical study on code

migration by Tran et al. [341] showed that the BLEU [265] accuracy score commonly used in natural language machine translation has only weak correlation with the semantic correctness of the translated source code [341]. The actual effect of these methods on the developer workflow is surprisingly unattested, and thus we asked a crucial question: are these models good enough to be useful to actual developers?

To answer this question, we conducted a user study on in-IDE natural language to code generation, where we perform the first comprehensive investigation of the promise and challenges of using such technology inside an IDE, asking "at the current state of technology does it improve developer productivity or accuracy, how does it affect the developer experience, and what are the remaining gaps and challenges?" (Chapter 4) We answer 3 concrete research questions: 1) How does using a NL2Code developer assistant affect task completion time and program correctness? 2) How do users query the NL2Code assistant, and how does that associate with their choice of generated vs retrieved code? 3) How do users perceive the usefulness of the in-IDE NL2Code developer assistant?

To facilitate the study, we first developed a plugin for the PyCharm IDE that implements a hybrid of code generation and code retrieval functionality, and orchestrate virtual environments to enable collection of many user events (e.g. web browsing, keystrokes, fine-grained code edits). We asked developers with various backgrounds to complete 7 varieties of 14 Python programming tasks ranging from basic file manipulation to machine learning or data visualization, with or without the help of the plugin. While qualitative surveys of developer experience were largely positive, quantitative results with regards to increased productivity, code quality, or program correctness were inconclusive. Further analysis identified several pain points that could improve the effectiveness of future machine learning-based code generation/retrieval developer assistants, and demonstrated when developers prefer code generation over code retrieval and vice versa. What this study made clear is that while these tools have significant potential, there are many open research questions that need to be tackled to make them practical.

1.3 Study of Retrieval-Augmented model Models

Retrieval-augmented model deep dive. One of the key pain point we discovered from the human study is that previous code generation models often lacks context. The study suggests that some queries may be better answered through code retrieval techniques, and others through code generation. Sometimes it is required to consider the user's local workspace context as part of the input. On some occasions, it is better to refer to the programming library documentations

and relevant open-source projects on the internet for more accurate code snippets.

This problem leads us to dive into studying retrieval-augmented language models. At the time, most current neural LMs are based on *parametric* neural networks, using RNN [236] or Transformer [347] architectures. These models make predictions solely using a fixed set of neural network parameters. Then, more and more neural LMs also incorporate *non-parametric* components [101, 110, 120, 165], or "retrieval-augmented LMs" [12, 38, 101, 110, 120, 166], which usually first select examples from an external source and then reference them during the prediction. For example, Khandelwal et al. [165] model the token-level probability by interpolating the parametric LM probability with a probability obtained from the nearest context-token pairs in an external datastore. Using such non-parametric components in LMs is beneficial because the model no longer needs to memorize everything about the language in its parameters. One of the most surprising results from Khandelwal et al. [166] is that kNN-LM reduces the perplexity of the base LM *even when the kNN component is retrieving examples from the same training set that the LM was originally trained on*, indicating that kNN-LM improves the ability to model the training data and is not simply benefiting from access to more data. Intrigued by this finding, we wonder why does kNN-LM work, and how does it improve already-trained strong transformer-based models? We set out to understand why kNN-LMs work even in this setting and concluded with some most important contributing factors to the improvement, as well as sharing many failed hypotheses for future insight (Chapter 6).

With more insights and better understanding of how non-parametric language models work, we could now incorporate retrieval with generation in these language models. To tackle the problem of code generation models at the time lacks user context when used inside an IDE, we explore the importance of **structural locality** in models of code (Chapter 5). Structural locality (*e.g.*, article and section levels, code path in project directories, *etc.*) is a ubiquitous feature of real-world datasets, wherein data points are organized into local hierarchies. We explore utilizing this structural locality within non-parametric language models, which generate sequences that reference retrieved examples from an external source [164]. The intuition is that source code files that are under the same project may be more useful for reference than random source code found on the Internet. We propose a simple yet effective approach for adding locality information into such models by adding learned parameters that improve the likelihood of retrieving examples from local neighborhoods. Experiments on two different domains, Java source code and Wikipedia text, have demonstrated that locality features improve model efficacy over models without access to these features, with interesting differences. We also performed an analysis of how and where locality features contribute to improved performance and why

the traditionally used contextual similarity metrics alone are not enough to grasp the locality structure.

Retrieval-augmented model applications With the discoveries in non-parametric language models, we turn to the applications of such retrieval-augmented methods in code generation domain. Many existing code generation models either learn directly from input-output pairs provided as training data [9, 11, 46, 141, 354, 364, 382], or learn the mapping between input and output implicitly from naturally occurring corpora of intertwined natural language and code [20, 253]. Nevertheless, all these works assume that *all libraries and function calls were seen in the training data*; and that at test time, the trained model will need to generate only *seen* libraries and function calls. However, new functions and libraries are introduced all the time, and even a seen function call can have unseen arguments. Thus, these existing models *inherently cannot* generalize to generate such unseen usages. In contrast to these existing models, human programmers frequently refer to manuals and documentation when writing code [192, 257]. This allows humans to easily use functions and libraries they have never seen nor used before. Inspired by this ability, we propose DocPrompting: a code generation approach that learns to retrieve code documentation before generating the code (Chapter 7).

In natural language domain, retrieval augmented LMs commonly use a retrieve-and-generate setup where they retrieve documents based on the user's input, and then generate a complete answer conditioning on the retrieved documents [54, 112, 142, 144, 149, 186, 189, 195, 248, 280, 308, 320]. Long-form generation (*e.g.*, long answer QA, summarization, *etc.*) presents complex information needs that are *not always evident from the input alone*. Similar to how humans gradually gather information as we create content such as papers, essays, or books, long-form generation with LMs would *require gathering multiple pieces of knowledge throughout the generation process*. Based on this intuition, our goal is to create a simple and generic retrieval augmented LM that *actively decides when and what to retrieve* throughout the generation process, and are applicable to a variety of long-form generation tasks. We propose Forward-Looking Active REtrieval augmented generation (FLARE) (Chapter 8). FLARE iteratively generates *a temporary next sentence*, use it as the query to retrieve relevant documents *if it contains low-probability tokens* and regenerate the next sentence until reaches the end. It is applicable to any existing LMs at inference time without additional training.

1.4 From Code Generation to LLM Agents

Throughout my thesis, prior work mostly focus on generating general domain programs given natural language. However, another important part of the over-arching goal of interacting with computers via natural language is to make computers as personal agents that execute tasks that are not limited to programming, but rather actually operating the computer graphical interface as if the agent is human: *e.g.*, online shopping, spreadsheet analysis, document management, *etc.* During the course of my thesis work, large language models, as well as generative AI has gone through significant development. Latest large language models such as OpenAI's GPT-4 have shown impressive zero-shot ability in many tasks and domains [49]. With advances in generative AI, there is now potential for autonomous agents to manage daily tasks via natural language commands. Autonomous agents that perform everyday tasks via human natural language commands could significantly augment human capabilities, improve efficiency, and increase accessibility.

However, current agents are primarily created and tested in simplified synthetic environments, leading to a disconnect with real-world scenarios. Current environments for evaluating agents tend to *over-simplify* real-world situations. As a result, the functionality of many environments is a limited version of their real-world counterparts, leading to a lack of task diversity [15, 99, 237, 319, 323, 324, 371]. In addition, these simplifications often lower the complexity of tasks as compared to their execution in the real world [279, 323, 371]. Finally, some environments are presented as a static resource [71, 319] where agents are confined to accessing only those states that were previously cached during data collection, thus limiting the breadth and diversity of exploration. For evaluation, many environments focus on comparing the textual *surface form* of the predicted action sequences with reference action sequences, disregarding the *functional correctness* of the executions and possible alternative solutions [71, 146, 204, 279, 366]. These limitations often result in a discrepancy between simulated environments and the real world, and can potentially impact the generalizability of AI agents to successfully understand, adapt, and operate within complex real-world situations.

At the beginning, we focused on automating web browsing tasks as they cover a large amount of daily tasks from online shopping to sending emails. To kickstart research in agents for goal-oriented web browsing tasks, we created WebArena, a *realistic* and *reproducible* web environment designed to facilitate the development of autonomous agents capable of executing tasks (Chapter 9). Our environment comprises four fully operational, self-hosted web applications, each representing a distinct domain prevalent on the internet: online shopping, discussion

forums, collaborative development, and business content management. It introduces a multi-step, long action horizon execution-based evaluation benchmark on interpreting high-level realistic natural language commands to concrete web-based interactions. It is also complemented by an extensive collection of documentation and knowledge bases that vary from general resources like English Wikipedia to more domain-specific references, such as manuals for using the integrated development tool [81]. The content populating these websites is extracted from their real-world counterparts, preserving the authenticity of the content served on each platform.

Along with WebArena, we also release a ready-to-use benchmark with 812 long-horizon web-based tasks. Each task is described as a high-level natural language intent, emulating the abstract language usage patterns typically employed by humans [34]. We focus on evaluating the *functional correctness* of these tasks, *i.e.*, does the result of the execution actually achieve the desired goal. This evaluation is not only more reliable [55, 356, 403] than comparing the textual surface-form action sequences [71, 279] but also accommodate a range of potential valid paths to achieve the same goal, which is a ubiquitous phenomenon in sufficiently complex tasks.

We use this benchmark to evaluate several agents that can follow NL command and perform web-based tasks. These agents are implemented in a few-shot in-context learning fashion with powerful large language models (LLMs) such as GPT-4 and PALM-2. Experiment results show that the best GPT-4 agent performance is somewhat limited, with an end-to-end task success rate of only 14.41%, while the human performance is 78.24%.

These outcomes underscore the necessity for further development towards robust and effective agents [188]. Based on the limited performance for the current state-of-the-art systems in our benchmark, we propose several directions of work in further investigating and improving LLM agents. First, we argue that even though web browsing is a natural way of interacting with computers for humans, it may not necessary be the best interface for LLM agents, as the representation of the current webpage, either being HTML source code or screenshots are sometimes too complex and contains a lot of irrelevant distractors for the LLM to use. Besides, they also tend to take up a lot of valuable context length available to the LLM input. We propose to do a comparison study for the same tasks in WebArena, but performed with a web browsing environment and agent, against an agent that interacts with the same websites only by programmatically calling APIs (Chapter 10). We argue that APIs may provide more concise information with better machine readability for LLM agents, and we propose an API calling LLM agent that could retrieve the correct APIs to call and learn how to use the responses based on the task and step contexts.

Second, we propose to improve existing LLMs through large amounts of instructions, manuals,

tutorials and demonstrations found online about how to perform certain tasks (Chapter 11). We believe that by pre-training or retrieving relevant instructions, LLMs agents can more probably generate actions that may have been showcased online about how to tackle the given situation. More generally, web-browsing LLM agents iteratively use LLM to generate next-step actions and iteratively take in the newly updated observation states of the environment.

Finally, based on the iterative nature of predicting actions then generating next steps based on the feedback of the previous actions in the given environment, we propose to make this iterative framework more general towards code generation and API-calling. We propose to generate longer code completions or API calling sequences by iteratively generating part of it and then consider the intermediate execution responses to inform a more accurate generation later on (Chapter 11). Furthermore, the agent could also operate in a conversational style, which enables the ability to ask for clarification or to let the users step in to correct an erroneous step. Coincidentally, this is also inspired by the results from our human study in code generation tools inside IDEs (Chapter 4), where some user pain points could be alleviated by incorporating dialogue-based query capability. Dialogue-based querying could allow users to refine their natural language intents until they are sufficiently precise for the underlying models to confidently provide output.

1.5 Summary of Progress

Part I, Part II and Part III are completed. The natural language to code generation model that first utilizes pretraining external knowledge (Chapter 2) is published at ACL 2020. One of the largest open-source pretrained code language model at the time of release, supporting multiple programming languages and a comprehensive evaluation study (Chapter 3) is published in Deep Learning For Code (DL4C) Workshop at ICLR and The 6th Annual Symposium on Machine Programming (MAPS) at PLDI, 2022. The human study of code generation in real-world IDE scenarios (Chapter 4) is published in ACM Transactions on Software Engineering and Methodology. A better non-parametric language model utilizing "structural locality" information (Chapter 5) is published in ICLR 2022. A comprehensive study of why non-parametric language models like kNN-LM work (Chapter 6) is published in ICML 2023. DocPrompting that first retrieves relevant library documentation and then generate code based on the reference (Chapter 7) is published in ICLR 2023. FLARE that actively decides when and what to retrieve during long-form language model generation (Chapter 8) is published in EMNLP 2023. WebArena, a realistic web environment for building autonomous agents (Chapter 9) is currently under review.

The comparison study between web-based WebArena tasks versus API-based and a new API calling LLM agent is proposed as future work. In addition, a better agent that could learn from large amount of demonstration and instructions available online, and an interactive and iterative code generation model with execution context are both proposed as future work.

The work presented here has also inspired my other relevant research projects not included in the thesis. These include probing and prompting language models for factual knowledge (TACL paper), a benchmark for structured procedural knowledge extraction from cooking videos (EMNLP 2020 Workshop paper), a code model learning structural edits via incremental tree transformations (ICLR 2021 paper), a neuro-symbolic language model with automaton-augmented retrieval (ICML 2022 paper), a benchmark called MCoNaLa for code generation from multiple natural languages (EACL 2023 paper) and a hierarchical prompting method that assists LLMs on web navigation (EMNLP 2023 paper).

1.6 Proposed Timeline

Jan 2024	Thesis Proposal
Jan 2024 — Dec. 2024	Finish the Proposed Work
Feb. 2025	Thesis Defense

Part I

Pre-training for Code Generation

Chapter 2

Incorporating External Knowledge through Pre-training for Natural Language to Code Generation (Completed)

Open-domain code generation aims to generate code in a general-purpose programming language (such as Python) from natural language (NL) intents. Motivated by the intuition that developers usually retrieve resources on the web when writing code, we explore the effectiveness of incorporating two varieties of external knowledge into NL-to-code generation: automatically mined NL-code pairs from the online programming QA forum StackOverflow and programming language API documentation. Our evaluations show that combining the two sources with data augmentation and retrieval-based data re-sampling improves the current state-of-the-art by up to 2.2% absolute BLEU score on the code generation testbed CoNaLa. The code and resources are available at https://github.com/neulab/external-knowledge-codegen.

2.1 Introduction

Semantic parsing, the task of generating machine executable meaning representations from natural language (NL) intents, has generally focused on limited domains [67, 395], or domain-specific languages with a limited set of operators [32, 76, 179, 205, 282, 389, 390, 391, 404]. However, recently there has been a move towards applying semantic parsing to automatically

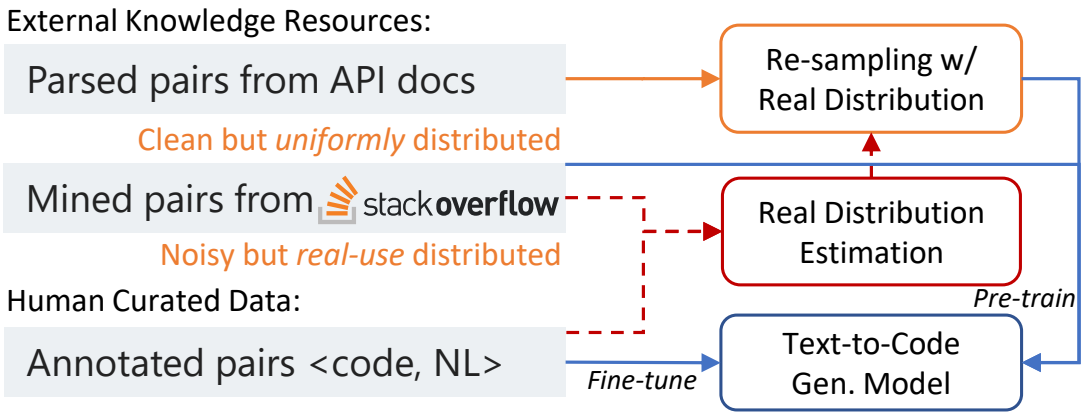


Figure 2.1: Our approach: incorporating external knowledge by data re-sampling, pre-training and fine-tuning.

generating source code in general-purpose programming languages [2, 209, 377, 380, 388]. Prior work in this area [77, 140, 211, 283, 335, 363, 381, 384, 386] used a variety of models, especially neural architectures, to achieve good performance.

However, open-domain code generation for general-purpose languages like Python is challenging. For example, given the intent to *choose a random file from the directory contents of the C drive, 'C:\\'*, one would expect the Python code snippet `random.choice(os.listdir('C:\\'))`, that realizes the given intent. This would involve not just generating syntactically correct code, but also using (and potentially combining) calls to APIs and libraries that implement some of the desired functionality. As we show in § 2.3, current code generation models still have difficulty generating the correct function calls with appropriate argument placement. For example, given the NL intent above, although the state-of-the-art model by Yin and Neubig [384] that uses a transition-based method to generate Python abstract syntax trees is guaranteed to generate *syntactically correct* code, it still incorrectly outputs `random.savefig(random(compile(open('C:\\'))+100).isoformat())`.

A known bottleneck to training more accurate code generation models is the limited number of manually annotated training pairs available in existing human-curated datasets, which are insufficient to cover the myriad of ways in which some complex functionality could be implemented in code. However, increasing the size of labeled datasets through additional human annotation is relatively expensive. It is also the case that human developers rarely reference such paired examples of NL and code, and rather take external resources on the web and modify them into the desired form [42, 43, 105]. Motivated by these facts, we propose to improve the performance of code generation models through a novel training strategy: pre-training the model on data

extracted automatically from external knowledge resources such as existing API documentation, before fine-tuning it on a small manually curated dataset (§ 2.2.1). Our approach, outlined in Figure 2.1, combines pairs of NL intents and code snippets mined automatically from the Q&A website StackOverflow (§ 2.2.2), and API documentation for common software libraries (§ 2.2.3).[1]

While our approach is model-agnostic and generally applicable, we implement it on top of a state-of-the-art syntax-based method for code generation, TranX [384], with additional hypothesis reranking [386]. Experiments on the CoNaLa benchmark [388] show that incorporating external knowledge through our proposed methods increases BLEU score from 30.1 to 32.3, outperforming the previous state-of-the-art model by up to 2.2% absolute. Qualitatively analyzing a sample of code snippets generated by our model reveals that the generated code is more likely to use the correct API calls for desired functionality and to arrange arguments in the right order.

2.2 Approach

2.2.1 Over-arching Framework

The overall strategy for incorporating external knowledge that we take on this work is to (1) *pre-train* the model on the NL-code pairs obtained from external resources, then (2) *fine-tune* on a small manually curated corpus. This allows the model to first learn on larger amounts of potentially noisy data, while finally being tailored to the actual NL and code we want to model at test time. In order to perform this pre-training we need to convert external data sources into NL-code pairs, and we describe how to do so in the following sections.

2.2.2 Mined NL-code Pairs

When developers code, most will inevitably search online for code snippets demonstrating how to achieve their particular intent. One of the most prominent resources online is StackOverflow,[2] a popular programming QA forum. However, it is not the case that all code on StackOverflow actually reflects the corresponding intent stated by the questioner – some may be methods defining variables or importing necessary libraries, while other code may be completely irrelevant. Yin et al. [388] propose training a classifier to decide whether an NL-code pair is valid, resulting in a large but noisy parallel corpus of NL intents and source code snippets. The probability

[1] Of course external knowledge for code covers a large variety of resources, other than these two types.
[2] https://stackoverflow.com

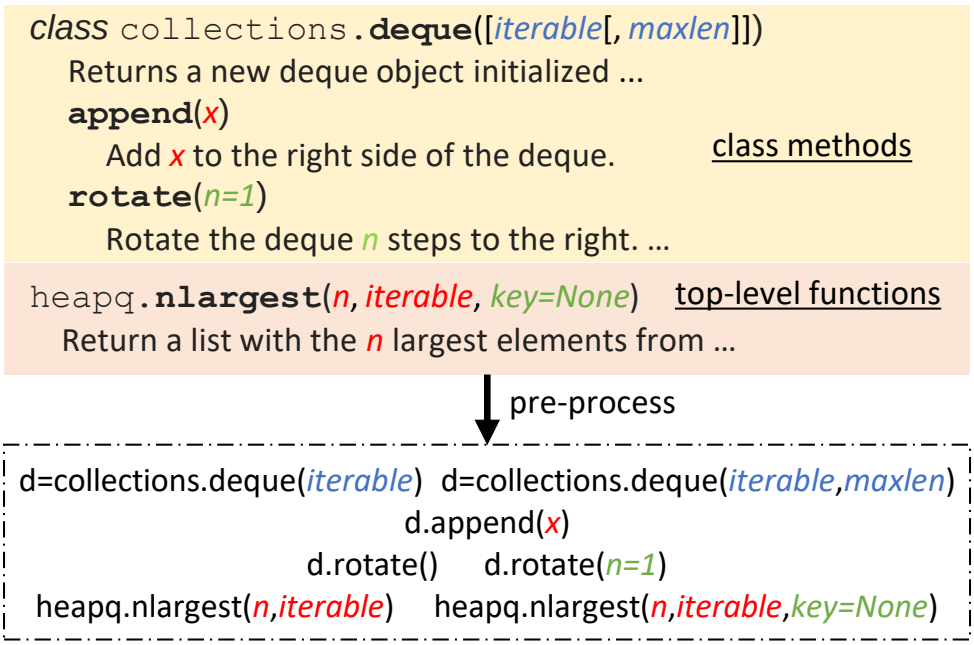


Figure 2.2: Examples from Python API documentation and pre-processed code snippets, including class constructors, methods, and top-level functions. We use red, blue, and green to denote required, optional positional, and optional keyword arguments respectively.

assigned by the method can serve as confidence, representing the quality of the automatically mined NL-code pairs. We use these mined pairs as a first source of external knowledge.

2.2.3 API Documentation

Second, motivated by the intuition that much of modern software development relies on libraries, and that developers often turn to programming language and software library references for help while writing code, we consider API documentation as another source of external knowledge.

Figure 2.2 shows some examples from the Python standard library API documentation. It contains descriptions of libraries, classes, methods, functions, and arguments. The documentation is already in a paired form consisting of code signatures and their descriptions. However, the signatures shown in the documentation mainly provide the prototype of the API rather than valid API usages appearing in source code. The text descriptions in the documentation tend to be verbose for clarity, while real questions from developers are usually succinct. We use a few heuristics to transform these to emulate real inputs a code generation system may face.

Most APIs define required and optional arguments in the signature. In real usage, developers usually provide none or only some of those arguments. To simulate this, we permute all possible

combinations (with a limit) of the optional arguments and append them to the required arguments, following correct syntax. For class constructors and methods, we create a heuristic variable name based on the class name to store the instantiated class object and to call methods upon. To make concise description for each code snippet created, we preserve only the first sentence in the corresponding documentation, as well as the first sentences that contain mentions of each argument in the snippet. In the rare case where arguments are not found in the original description, we add another sentence containing these arguments to the end of the NL snippet, ensuring all variables in code are covered in the NL.

We then describe detailed heuristics used for API documentation preprocessing. The goal is to harvest NL-code pairs with API docs as a source.

Arguments

Most APIs will have arguments, either required or optional. For the required arguments, we leave them "as-is". We deal with two types of optional arguments, positional arguments and keyword arguments through permutation and sampling. In the Python documentation, optional positional arguments are bracketed in "`[.., [..]]`". Nested brackets are commonly used to represent more than one possible optional positional arguments. Another type of optional arguments are implemented using keyword arguments in the form of `key=default`.

In real usage, developers usually only provide none or some of those arguments. To simulate this, we permute all possible combinations of the optional arguments, and append them to the required arguments. For example, if the code signature in the documentation writes "`collections.deque([iterable[, maxlen]])`", we produce all 3 possible usages: "`collections.deque()`", "`collections.deque(iterable)`", and "`collections.deque(iterable, maxlen)`". For keyword arguments like "`heapq.nlargest(n, iterable, key=None)`", we will also include "`heapq.nlargest(n, iterable)`" in addition. The total number of permutations is $n + 1$ for a function with n optional positional arguments, and $2^n = \binom{n}{0} + \binom{n}{1} + ... + \binom{n}{n}$ for a function with n optional keyword arguments, which leads to exponentially large number of samples for functions with many optional keywords. Motivated by the observation that developers rarely specify all of the optional arguments, but rather tend to use default values, we only keep the top 10 permutations with the least number of optional arguments.

Class Initializers and Methods

Other heuristics are used to transform code signatures related to classes to emulate real usage. For class initializers in the documentation, we construct an assignment statement with lower-cased variable name using the first character of the class name to store the instantiated class, e.g. `d = collections.deque(iterable)`. For class methods, we prepend a heuristically created variable name to the method call, emulating a real method call on an instantiated class, e.g. `d.append(x)`.

Documentation

Official documentation tends to be verbose for clarity, while real questions from developers are usually succinct. Thus we use the following heuristics to keep only sentences in the document that are necessary for generating the code as the intent text. We include the first sentence because it usually describes the functionality of the API. For each argument in the emulated API usage code snippet, we include the first sentence in the documentation that mentions the argument through string matching. For arguments not mentioned in the documentation, we add a sentence in the end: "With arguments `'arg_name'` ..." to ensure all arguments are covered verbatim in the intent text.

2.2.4 Re-sampling API Knowledge

External knowledge from different sources has different characteristics. NL-code pairs automatically mined from StackOverflow are good representatives of the questions that developers may ask, but are inevitably noisy. NL-code pairs from API documentation are clean, but there may be a topical distribution shift from real questions asked by developers. For example, the library `curses` has significantly more API entries than `json` (178 vs. 17),[3] while `json` is more frequently asked about and used. This distributional shift between pre-training and fine-tuning causes performance degradation, as shown later in § 2.3.2.

To mitigate this problem, we propose a retrieval-based re-sampling method to close the gap between the API documentation and the actual NL-code pairs we want to model. We use both human annotated data $\mathcal{D}_{\text{ann}}$ and mined data $\mathcal{D}_{\text{mine}}$ to model the distribution of NL-code pairs because they are both produced by real users. For each sample in this real usage distribution,

[3] https://docs.python.org/3.7/library/curses.html and https://docs.python.org/3.7/library/json.html

we retrieve k NL-code pairs from the set of pairs harvested from API documentation $\mathcal{D}_{\text{API}}$ and aggregate the frequencies of each pair $y \in \mathcal{D}_{\text{API}}$ being retrieved:

$$\text{freq}(y) = \sum_{x \in \mathcal{D}_{\text{ann+mined}}} \delta(y \in R(x, \mathcal{D}_{\text{API}}, k)),$$

where $R(x, \mathcal{D}_{\text{API}}, k)$ retrieves the top k most similar samples from $\mathcal{D}_{\text{API}}$ given x, either according to NL intent or code snippet. $\delta(\cdot)$ is Kronecker's delta function, returning 1 if the internal condition is true, and 0 otherwise. We use the BM25 retrieval algorithm [152] implemented in ElasticSearch.[4] We take this frequency and calculate the probability distribution after smoothing with a temperature $\tau \in [1, \infty]$:

$$P(y) = \text{freq}(y)^{1/\tau} / \sum_{y' \in \mathcal{D}_{\text{API}}} \text{freq}(y')^{1/\tau}$$

As τ changes from 1 to ∞, $P(y)$ shifts from a distribution proportional to the frequency to a uniform distribution. Using this distribution, we can sample NL-code pairs from the API documentation that are more likely to be widely-used API calls.

2.3 Experiments

2.3.1 Experimental Settings

Dataset and Metric: Although the proposed approach is generally applicable and model-agnostic, for evaluation purposes, we choose CoNaLa [388] as the human-annotated dataset (2,179 training, 200 dev and 500 test samples). It covers real-world English queries about Python with diverse intents. We use the same evaluation metric as the CoNaLa benchmark, corpus-level BLEU calculated on target code outputs in test set.

Mined Pairs: We use the CoNaLa-Mined [388] dataset of 600K NL-code pairs in Python automatically mined from StackOverflow (§ 2.2.2). We sort all pairs by their confidence scores, and found that approximately top 100K samples are of reasonable quality in terms of code correctness and NL-code correspondence. We therefore choose the top 100K pairs for the experiments.

API Documentation Pairs: We parsed all the module documentation including libraries, built-in types and functions included in the Python 3.7.5 distribution.[5] After pre-processing

[4] https://github.com/elastic/elasticsearch. When retrieving with code snippets, all the punctuation marks are removed.

[5] https://docs.python.org/release/3.7.5/library/index.html

Data Strategy	Method		BLEU
Man			27.20
Man+Mine	50k		27.94
	100k		28.14
Man+Mine+API	w/o re-sampling		27.84
	direct	intent	29.66
	dist.	intent	29.31
	direct	code	30.26
	dist.	code	**30.69**
Man			30.11
Man+Mine(100k)	+rerank		31.42
Our best			**32.26**

Table 2.1: Performance comparison of different strategies to incorporate external knowledge.

(§ 2.2.3), we create about 13K distinct NL-code pairs (without re-sampling) from Python API documentation. For fair comparison, we also sample the same number of pairs for the re-sampling setting (§ 2.2.4).

Methods: We choose the current state-of-the-art NL-to-code generation model TranX [384] with hypothesis reranking [386] as the base model. Plus, we incorporate length normalization [58] to prevent beam search from favoring shorter results over longer ones. **Man** denotes training solely on CoNaLa. **Man+Mine** refers to first pre-training on mined data, then fine-tuning on CoNaLa. **Man+Mine+API** combines both mined data and API documentation for pre-training. As a comparison to our distribution-based method (denoted by **dist.**, § 2.2.4), we also attempt to directly retrieve top 5 NL-code pairs from API documents (denoted by **direct**).[6]

Implementation Details: We experiment with $k = \{1, 3, 5\}$ and $\tau = \{1, 2, 5\}$ in re-sampling, and find that $k = 1$ and $\tau = 2$ perform the best. We follow the original hyper-parameters in TranX, except that we use a batch size of 64 and 10 in pre-training and fine-tuning respectively.

2.3.2 Results

Results are summarized in Table 2.1. We can first see that by incorporating more noisy mined data during pre-training allows for a small improvement due to increased coverage from the

[6]We choose 5 to obtain comparable amount of pairs.

much larger training set. Further, if we add the pairs harvested from API docs for pre-training without re-sampling the performance drops, validating the challenge of distributional shift mentioned in § 2.2.4.

Comparing the two re-sampling strategies **direct** vs. **dist.**, and two different retrieval targets NL intent vs. code snippet, we can see that **dist.** performs better with the code snippet as the retrieval target. We expect that using code snippets to retrieve pairs performs better because it makes the generation *target*, the code snippet, more similar to the real-world distribution, thus better training the decoder. It is also partly because API descriptions are inherently different than questions asked by developers (e.g. they have more verbose wording), causing intent retrieval to be less accurate.

Lastly, we apply hypothesis reranking to both the base model and our best approach and find improvements afforded by our proposed strategy of incorporating external knowledge are mostly orthogonal to those from hypothesis reranking.

After showing the effectiveness of our proposed re-sampling strategy, we are interested in the performance on more-used versus less-used APIs for the potentially skewed overall performance. We use string matching heuristics to obtain the standard Python APIs used in the dataset and calculated the average frequency of API usages in each data instance. We then select the top 200 and the bottom 200 instances out of the 500 test samples in terms of API usage frequencies. Before and after adding API docs into pre-training, the BLEU score on both splits saw improvements: for high-frequency split, it goes from 28.67 to 30.91 and for low-frequency split, it goes from 27.55 to 30.05, indicating that although the re-sampling would skew towards high-frequency APIs, with the appropriate smoothing temperature experimentation, it will still contribute to performance increases on low-frequency APIs.

Besides using BLEU scores to perform holistic evaluation, we also perform more fine-grained analysis of what types of tokens generated are improving. We apply heuristics on the abstract syntax tree of the generated code to identify tokens for API calls and variable names in the test data, and calculated the token-level accuracy for each. The API call accuracy increases from 31.5% to 36.8% and the variable name accuracy from 41.2% to 43.0% after adding external resources, meaning that both the API calls and argument usages are getting better using our approach.

Open a file “`f.txt`” in write mode. ✓ `f=open('f.txt', 'w')` ♠ `f=open('f.txt', 'f.txt')` ♣ `f=open('f.txt', 'w')`
lower a string *text* and remove non-alphanumeric characters aside from space. ✓ `re.sub(r'[^\sa-zA-Z0-9]', '', text).lower().strip()` ♠ `text.decode.translate(text.strip(), 'non-alphanumeric', '')` ♣ `re.sub(r'[^\sa-zA-Z0-9]', '', text)`
choose a random file from the directory contents of the C drive, 'C:\\'. ✓ `random.choice(os.listdir('C:\\'))` ♠ `random.savefig(random(compile(open('C:\\'))+100).isoformat())` ♣ `random.choice(os.path.expanduser('C:\\'))`

Table 2.2: Examples, where ✓ is the ground-truth code snippet, ♠ is the original output, and ♣ is the output with our proposed methods. Correct and erroneous function calls are marked in blue and red respectively.

2.3.3 Case Study

We further show selected outputs from both the baseline and our best approach in Table 2.2. In general, we can see that the NL to code generation task is still challenging, especially with more complex intents that require nested or chained API calls, or functions with more arguments. The vanilla model already can generate basic functions and copy strings/variables to the output, but we observe that incorporating external knowledge improves the results in two main ways: 1) better argument placement for APIs, and 2) better selection of which API call should be used for a certain intent.

In the first example, we can see that although the baseline gets the function call “`open()`” correct, it fails to generate the correct second argument specifying write mode, while our approach is able to successfully generate the appropriate `'w'`. In the second and third example,

we can see that the baseline uses the wrong API calls, and sometimes "makes up" APIs on its own (e.g. "`random.savefig()`"). However, our approach's outputs, while not perfect, are much more successful at generating correct API calls that actually exist and make sense for the intent.

On a closer look, we can observe that both the addition of mined examples and API docs may have brought the improvement. The example of the "`open()`" function added from API docs uses the default mode "`r`", so learning the meaning of "`w`" argument is due to the added mined real examples, but learning the argument placement (first file name as a string, second a shorthand mode identifier as a character) may have occurred from the API docs. In other examples, "`random.choice()`" and "`re.sub()`" both are Python standard library APIs so they are included in the API doc examples.

2.4 Conclusion and Future Work

We proposed a model-agnostic approach based on data augmentation, retrieval and data re-sampling, to incorporate external knowledge into code generation models, which achieved state-of-the-art results on the CoNaLa open-domain code generation task.

In the future, evaluation by automatically executing generated code with test cases could be a better way to assess code generation results. It will also likely be useful to generalize our re-sampling procedures to zero-shot scenarios, where a programmer writes a library and documents it, but nobody has used it yet. For example, developers may provide relative estimates of each documented API usages to guide the re-sampling; or we could find nearest neighbors to each API call in terms of semantics and use existing usage statistics as estimates to guide the re-sampling.

Chapter 3

A Systematic Evaluation of Large Language Models of Code (Completed)

The previous chapter introduced a novel way of utilize pre-training to improve the performance of natural language to code generation. However, they are based on specifically trained, task-specific code generation models.

In this chapter, we explore a more general form of representing and generating code. Large language models (LMs) of code have recently shown tremendous promise in completing code and synthesizing code from natural language descriptions. However, the current state-of-the-art code LMs (e.g., Codex [55]) are not publicly available, leaving many questions about their model and data design decisions. We aim to fill in some of these blanks through a systematic evaluation of the largest existing models: Codex, GPT-J, GPT-Neo, GPT-NeoX-20B, and CodeParrot, across various programming languages. Although Codex itself is not open-source, we find that existing open-source models do achieve close results in some programming languages, although targeted mainly for natural language modeling. We further identify an important missing piece in the form of a large open-source model trained exclusively on a multi-lingual corpus of code. We release a new model, PolyCoder, with 2.7B parameters based on the GPT-2 architecture, that was trained on 249GB of code across 12 programming languages on a single machine. In the C programming language, *PolyCoder outperforms all models including Codex.* Our trained models are open-source and publicly available at `https://github.com/VHellendoorn/Code-LMs`, which enables future research and application in this area.

3.1 Introduction

Language models (LMs) assign probabilities to sequences of tokens, and are widely applied to natural language text [23, 31, 47]. Recently, LMs have shown impressive performance in modeling also source code, written in programming languages [11, 125, 131, 158]. These models excel at useful downstream tasks like code completion [296] and synthesizing code from natural language descriptions [72]. The current state-of-the-art large language models for code, such as Austin et al. [20], have shown significant progress for AI-based programming assistance. Most notably, one of the largest of these models, Codex [55] has been deployed in the real-world production tool GitHub Copilot[1], as an in-IDE developer assistant that automatically generates code based on the user's context.

Despite the great success of large language models of code, *the strongest models are not publicly available*. This prevents the application of these models outside of well-resourced companies and limits research in this field for low-resourced organizations. For example, Codex provides non-free access to the model's *output* through black-box API calls,[2] but the model's weights and training data are unavailable. This prevents researchers from fine-tuning and adapting this model to domains and tasks other than code completion. The lack of access to the model's internals also prevents the research community from studying other key aspects of these models, such as interpretability, distillation of the model for more efficient deployment, and incorporating additional components such as retrieval.

Several medium to large-sized pre-trained language models are publicly available, such as GPT-Neo [35], GPT-J [349] and GPT-NeoX [36]. Despite being trained on a mixture of a wide variety of text including news articles, online forums, and just a modest selection of (GitHub) software repositories [91], these language models can be used to generate source code with a reasonable performance [55]. In addition, there are a few open-source language models that are trained solely on source code. For example, CodeParrot [344] was trained on 180 GB of Python code.

Given the variety of model sizes and training schemes involved in these models and lack of comparisons between these, the impact of many modeling and training design decisions remains unclear. For instance, we do not know the precise selection of data on which Codex and other private models were trained; however, we do know that some public models (e.g., GPT-J) were trained on a mix of natural language and code in multiple programming languages, while other

[1] https://copilot.github.com/
[2] https://openai.com/blog/openai-codex/

models (e.g., CodeParrot) were trained solely on code in one particular programming language. Multilingual models potentially provide better generalization, because different programming languages share similar keywords and properties, as shown by the success of *multilingual* models for natural language [64] and for code [411]. This may hint that *multilingual* LMs can *generalize* across languages, outperform monolingual models and be useful for modeling low-resource programming languages, but this is yet to be verified empirically.

In this paper, we present a systematic evaluation of existing models of code – Codex, GPT-J, GPT-Neo, GPT-NeoX, and CodeParrot – across various programming languages. We aim to shed more light on the landscape of code modeling design decisions by comparing and contrasting these models, as well as providing a key missing link: thus far, no large open-source language model was trained exclusively on code from *multiple programming languages*. We provide three such models, ranging from 160M to 2.7B parameters, which we release under the umbrella name "PolyCoder". First, we perform an extensive comparison of the training and evaluation settings between PolyCoder, open-source models, and Codex. Second, we evaluate the models on the HumanEval benchmark [55] and compare how do models of different sizes and training steps scale, and how different temperatures affect the generation quality. Finally, since HumanEval only evaluates the natural language to Python synthesis, we curate an unseen evaluation dataset[3] in each of the 12 languages, to evaluate the perplexity of different models. We find that although Codex is allegedly focused on Python ([55] §3.1), Codex performs surprisingly well in other programming languages too, and even better than GPT-J and GPT-NeoX that were trained on the Pile [91]. Nonetheless, in the C programming language, *our PolyCoder model achieves a lower perplexity than all these models, including Codex.*

Although most current models perform worse than Codex, we hope that this systematic study helps future research in this area to design more efficient and effective models. More importantly, through this systematic evaluation of different models, we encourage the community to study and release medium-large scale language models for code, in response to the concerns expressed by Hellendoorn and Sawant [126]:

> *[...] this exploding trend in cost to achieve the state of the art has left the ability to train and test such models limited to a select few large technology companies—and way beyond the resources of virtually all academic labs.*

We believe that our efforts are a significant step towards democratization of large language models of code.

[3]The exact training set that Codex was trained on is unknown.

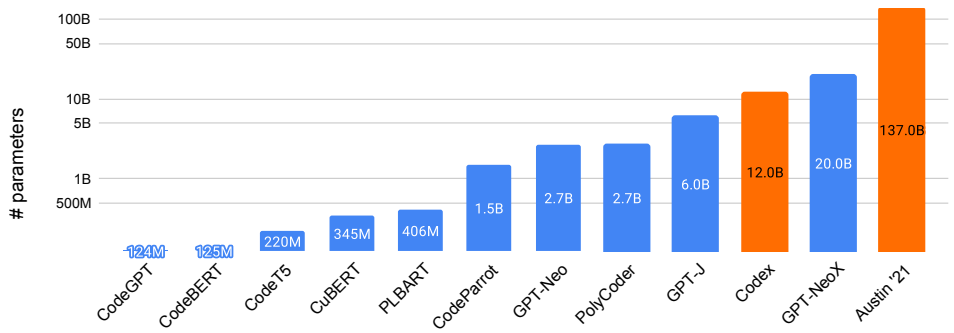


Figure 3.1: Existing language models of code, their sizes and availability (open source vs. not open-source).

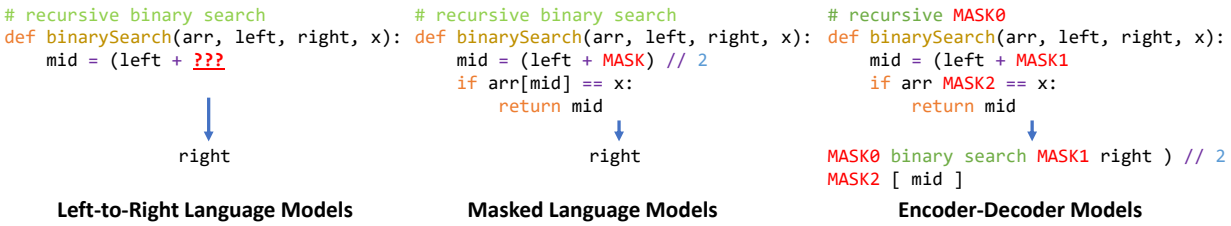


Figure 3.2: Three types of pretrained language models.

3.2 Related Work

At the core of code modeling lies ongoing work on pretraining of language models (LMs). Large-scale pretraining of LMs has had an astounding impact on natural language processing in recent years [114]. Figure 3.1 provides an overview of how different models compare in size and availability.

3.2.1 Pretraining Methods

We discuss three popular pretraining methods used in code language modeling. An illustration of these methods are shown in Figure 3.2.

Left-to-Right Language Models (Figure 3.2, left) Auto-regressive, Left-to-right LMs, predict the probability of a token given the previous tokens. In code modeling, CodeGPT (124M) [220], CodeParrot (1.5B) [344], GPT-Neo (2.7B) [35], GPT-J (6B) [349], Codex (12B) [55], GPT-NeoX (20B) [36], and Google's (137B) [20] belong to this category. The left-to-right nature of these

models makes them highly useful for program generation tasks, such as code completion. On the other hand, as code is usually not written in a single, left-to-write pass, it is not trivial to leverage context that appears "after" the location of the generation. In this paper, we focus on this family of models and will discuss the existing models in more detail in the following sections.

Masked Language Models (Figure 3.2, middle) While auto-regressive language models are powerful for modeling the probability of sequences, their unidirectional nature makes them less suitable for producing effective whole-sequence representations for downstream tasks such as classification. One popular bidirectional objective function used widely in representation learning is masked language modeling [73], where the aim is to predict masked text pieces based on surrounding context. CodeBERT (125M) [85] and CuBERT (345M) [156] are examples of such models in code. In programming contexts, these methods provide useful representations of a sequence of code for downstream tasks such as code classification, clone detection, and defect detection.

Encoder-decoder Models (Figure 3.2, right) An encoder-decoder model first uses an encoder to encode an input sequence, and then uses a left-to-right LM to decode an output sequence conditioned on the input sequence. Popular pretraining objectives include masked span prediction [288] where the input sequence is randomly masked with multiple masks and the output sequence are the masked contents in order, and denoising sequence reconstruction [193] where the input is a corrupted sequence and the output is the original sequence. These pretrained models are useful in many sequence-to-sequence tasks [288]. In code, CodeT5 (220M) [354], and PLBART (406M) [3] use the two objectives mentioned above respectively, and performs well in conditional generation downstream tasks such as code commenting, or natural language to code generation.

3.2.2 Pretraining Data

Some models (e.g. CodeParrot and CodeT5) are trained on GitHub code only, with corpora extracted using either Google BigQuery's GitHub dataset [4], or CodeSearchNet [137]. Others (e.g., GPT-Neo and GPT-J) are trained on "the Pile" [91], a large corpus containing a blend of natural language texts and code from various domains, including Stack Exchange dumps, software

[4] https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-co

documentations, and popular (>100 stars) GitHub repositories. The datasets on which other proprietary models (Codex, Google's) were trained on are unknown. One goal of our study is to try to shed light on what corpora might be the most useful for pretraining models of code.

3.3 Evaluation Settings

We evaluate all models using both extrinsic and intrinsic benchmarks, as described below.

Extrinsic Evaluation One of the most popular downstream tasks for code modeling is code generation given a natural language description. Following [55], we evaluate all models on the HumanEval dataset. The dataset contains 164 prompts with descriptions in the form of code comments and function definitions, including argument names and function names, and test cases to judge whether the generated code is correct. To generate code given a prompt, we use the same sampling strategy as Chen et al. [55], using softmax with a temperature parameter $\texttt{softmax}(x/T)$. We evaluate using a wide range of temperatures $T = [0.2, 0.4, 0.6, 0.8]$ to control for the confidence of the model's predictions. Similarly to Codex, we use nucleus sampling [135] with top-$p = 0.95$. We sample tokens from the model until we encounter one of the following stop sequences that indicate the end of a method:[5] '`\nclass`', '`\ndef`', '`\n#`', '`\nif`', or '`\nprint`'. We randomly sample 100 examples per prompt in the evaluation dataset.

Intrinsic Evaluation To evaluate the intrinsic performance of different models, we compute the perplexity for each language on an unseen set of GitHub repositories. To prevent training-to-test data leakage for models such as GPT-Neo and GPT-J, we remove repositories in our evaluation dataset that appeared in the GitHub portion of the Pile training dataset [6]. To evaluate Codex, we use OpenAI's API [7], choosing the `code-davinci-001` engine. We note that the data that this model was trained on is *unknown*, so we cannot prevent data leakage from the training to the test set for Codex. We sampled 100 random files for each of the 12 programming languages in our evaluation dataset. To make perplexity comparable across different tokenization methods used in different models, we use Pygments [8] to equally normalize the log-likelihood sum of each

[5] The absence of whitespace, which is significant in Python, signals an exit from the method body.
[6] https://github.com/EleutherAI/github-downloader
[7] https://beta.openai.com/docs/engines/codex-series-private-beta
[8] https://pygments.org/docs/lexers/

model, when computing perplexity.[9]

3.4 Compared Models

3.4.1 Existing Models

As discussed in Section 3.2, we mainly focus on auto-regressive left-to-right pretrained language models, most suitable for code completion tasks.

We evaluate Codex, as it is currently deployed in real-world and has impressive performance in code completion [55]. Codex uses the GPT-3 language model [47] as its underlying model architecture. Codex was trained on a dataset spanning 179GB (after deduplication) covering over 54 million public Python repositories obtained from GitHub on May 2020. As reflected in its impressive results in other programming languages than Python, we suspect that Codex was also trained on large corpora of additional programming languages. The model available for querying through a non-free API.

As for open-source models, we compare GPT-Neo, GPT-J and GPT-NeoX, the largest variants having 2.7, 6 and 20 billion parameters, respectively. GPT-NeoX is the largest open-source pretrained language models available. These models are trained on the Pile dataset, so they are a good representatives of models that were trained on both natural language texts from various domains and source code from GitHub. We also compare CodeParrot with at most 1.5 billion parameters, a model that was only trained on Python code from GitHub. CodeParrot follows the process used in [55] that obtained over 20M files Python files from Google BigQuery Github database, resulting in a 180GB dataset, which is comparable to Codex's *Python* training data, but the model itself is much smaller.

There was no large open-source language model trained almost exclusively on code from multiple programming languages. To fill this gap, we train a 2.7 billion model, PolyCoder, on a mixture of repositories from GitHub in 12 different programming languages.

3.4.2 PolyCoder's Data

Raw Code Corpus Collection GitHub is an excellent source for publicly available source code of various programming languages. We cloned the most popular repositories for 12 popular

[9] Every model uses its original tokenizer for predicting the next token. We use the shared tokenizer only for computing the perplexity given the log-likelihood sum.

Language	Repositories	Files	Size Before Filtering	Size After Filtering
C	10,749	3,037,112	221G	55G
C#	9,511	2,514,494	30G	21G
C++	13,726	4,289,506	115G	52G
Go	12,371	1,416,789	70G	15G
Java	15,044	5,120,129	60G	41G
JavaScript	25,144	1,774,174	66G	22G
PHP	9,960	1,714,058	21G	13G
Python	25,446	1,550,208	24G	16G
Ruby	5,826	674,343	5.0G	4.1G
Rust	4,991	304,842	5.2G	3.5G
Scala	1,497	245,100	2.2G	1.8G
TypeScript	12,830	1,441,926	12G	9.2G
Total	147,095	24,082,681	631.4G	253.6G

Table 3.1: Training corpus statistics.

programming languages with at least 50 stars (stopping at about 25K per language to avoid a too heavy skew towards popular programming languages) from GitHub in October 2021. For each project, each file belonging to the majority-language of that project was extracted, yielding the initial training set. This initial, unfiltered dataset spanned 631GB and 38.9M files.

Data Preprocessing The detailed data preprocessing strategy comparison with other models are analyzed in Table 3.2. In general, we tried to follow Codex's design decisions, although there is a fair bit of ambiguity in the description of its data preprocessing.

Deduplication and Filtering Similarly to Codex and CodeParrot, very large (>1MB) and very short (<100 tokens) files were filtered out, reducing the size of the dataset by 33%, from 631GB to 424GB. This only reduced the total *number* of files by 8%, showing that a small number of files were responsible for a large part of the corpus.[10]

[5] has shown that code duplication that commonly manifests in datasets of code adversely effects language modeling of code. Therefore, we deduplicated files based on a hash of their

[10]Codex additionally mentions removing "auto-generated" files, but the definition of this was not clear, so we omitted this step.

	PolyCoder	CodeParrot	Codex
Dedup	Exact	Exact	Unclear, mentions “unique”
Filtering	Files > 1 MB, < 100 tokens	Files > 1MB, max line length > 1000, mean line length > 100, fraction of alphanumeric characters < 0.25, containing the word "auto-generated" or similar in the first 5 lines	Files > 1MB, max line length > 1000, mean line length > 100, auto-generated (details unclear), contained small percentage of alphanumeric characters (details unclear)
Tokenization	Trained GPT-2 tokenizer on a random 5% subset (all languages)	Trained GPT-2 tokenizer on train split	GPT-3 tokenizer, add multi-whitespace tokens to reduce redundant whitespace tokens

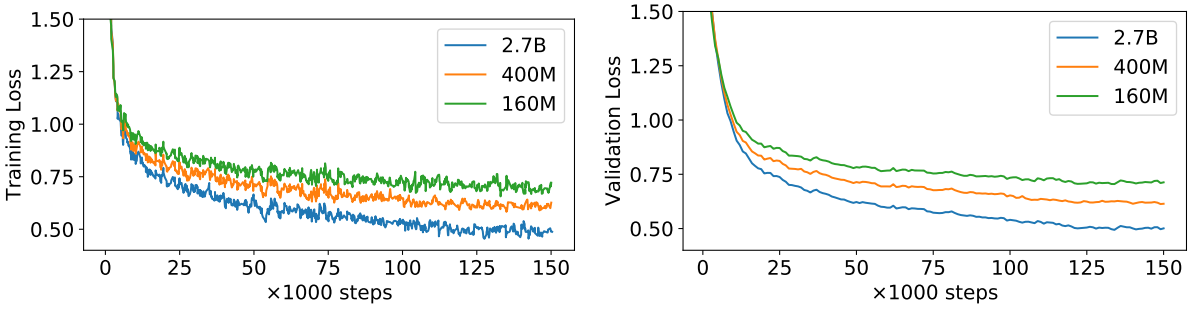
Table 3.2: Comparison of data preprocessing strategies of different models.

content, which reduced the number of files by nearly 30%, and the dataset size by additional 29%, leaving 24.1M files and 254GB of data.

Overall, the filtering of very large and very short files plus deduplication, reduced the number of files by 38%, and the dataset size by 61%, roughly on par with the 70% dataset size reduction reported by CodeParrot. A key difference that remains is that other approaches use more fine-grained filtering strategies, such as limiting the maximum line length or average line length, filtering of probable auto-generated files, etc. For example, Chen et al. [55] have filtered only 11% of their training data.

The dataset statistics are shown in Table 3.1, showcasing data sizes per language before and after filtering. Our dataset contains less Python code (only 16G) than Codex or CodeParrot, and instead covers many different programming languages.

Tokenizer We train a GPT-2 tokenizer (using BPE [313]) on a random 5% subset of all the pretraining data, containing all the languages. Codex uses an existing trained GPT-3 tokenizer, with the addition of multi-whitespace tokens to reduce the sequence length after tokenization, as consecutive whitespaces are more common in code than in text.



(a) Training

(b) Validation

Figure 3.3: Training and validation loss during the 150K step training process.

3.4.3 PolyCoder's Training

Considering our budget, we chose the GPT-2 [285] as our model architecture. To study the effect of scaling of model size, we train 3 different sized models, with 2.7 billion, 400 million and 160 million parameters, as the largest 2.7B model being on par with GPT-Neo for fair comparison. The 2.7 billion model is a 32 layer, 2,560 dimensional Transformer model, with a max context window of 2048 tokens, trained with a batch size of 128 sequences (262K tokens). The model is trained for 150K steps. The 400 million model is a 24 layer, 1,024 dimensional variant, and the 160 million model is a 12 layer, 768 dimensional variant, otherwise idem. We use GPT-NeoX toolkit [11] to train the model efficiently in parallel with 8 Nvidia RTX 8000 GPUs on a single machine. The wall time used to train the largest 2.7B model is about 6 weeks. In its default configuration, this model should train for 320K steps, which was not feasible with our resources. Instead, we adjusted the learning rate decay to half this number and trained for up to 150K steps (near-convergence). The training and validation loss curves for different sized models are shown in Figure 3.3. We see that even after training for 150K steps, the validation losses are still decreasing. This, combined with the shorter training schedule and faster learning rate decay, strongly signals that the models are still under-fitting and could benefit from longer training.

We compare the training setting and hyperparameters with CodeParrot and Codex in Table 3.3. Due to high computational costs, we were unable to perform hyperparameter search. Most hyperparameters are the same as those used in their respective GPT-2 model training [12] to provide a good default with regards to the corresponding model size. Some key differences include context window sizes to allow for more tokens as context, batch sizes and tokens trained,

[11] https://github.com/EleutherAI/gpt-neox

[12] https://github.com/EleutherAI/gpt-neox/tree/main/configs

	PolyCoder (2.7B)	CodeParrot (1.5B)	Codex (12B)
Model Initialization	From scratch	From scratch	Initialized from GPT-3
NL Knowledge	Learned from comments in the code	Learned from comments in the code	Natural language knowledge from GPT-3
Learning Rate	1.6e-4	2.0e-4	1e-4
Optimizer	AdamW	AdamW	AdamW
Adam betas	0.9, 0.999	0.9, 0.999	0.9, 0.95
Adam eps	1e-8	1e-8	1e-8
Weight Decay	-	0.1	0.1
Warmup Steps	1600	750	175
Learning Rate Decay	Cosine	Cosine	Cosine
Batch Size (#tokens)	262K	524K	2M
Training Steps	150K steps, 39B tokens	50K steps, 26B tokens	100B tokens
Context Window	2048	1024	4096

Table 3.3: Comparison of design decisions and hyper-parameters in training different models of code.

as well as model initialization with or without natural language knowledge.

3.5 Results

3.5.1 Extrinsic Evaluation

The overall results are shown in Table 3.4.[13] The numbers are obtained by sampling with different temperatures and picking the best value for each metric. Among existing models, PolyCoder is worse than similarly sized GPT-Neo and the even smaller Codex 300M. Overall, PolyCoder lies after Codex, GPT-Neo/J, while performing stronger than CodeParrot. PolyCoder, which was trained only on code, falls behind a similar sized model (GPT-Neo 2.7B) trained on the Pile, a blend of natural language texts and code. Looking at the rightmost columns in Table 3.4 offers a potential explanation: in terms of total Python tokens seen during training, all models

[13]Due to the large model size of GPT-NeoX (20B) and limited computational budget, we did not include it in the HumanEval experiment.

Model	Pass@1	Pass@10	Pass@100	Tokens Trained	Code Tokens	Python Tokens
PolyCoder (160M)	2.13%	3.35%	4.88%	39B	39B	2.5B
PolyCoder (400M)	2.96%	5.29%	11.59%	39B	39B	2.5B
PolyCoder (2.7B)	5.59%	9.84%	17.68%	39B	39B	2.5B
CodeParrot (110M)	3.80%	6.57%	12.78%	26B	26B	26B
CodeParrot (1.5B)	3.58%	8.03%	14.96%	26B	26B	26B
GPT-Neo (125M)	0.75%	1.88%	2.97%	300B	22.8B	3.1B
GPT-Neo (1.3B)	4.79%	7.47%	16.30%	380B	28.8B	3.9B
GPT-Neo (2.7B)	6.41%	11.27%	21.37%	420B	31.9B	4.3B
GPT-J (6B)	11.62%	15.74%	27.74%	402B	30.5B	4.1B
Codex (300M)	13.17%	20.37%	36.27%	100B*	100B*	100B*
Codex (2.5B)	21.36%	35.42%	59.50%	100B*	100B*	100B*
Codex (12B)	28.81%	46.81%	72.31%	100B*	100B*	100B*

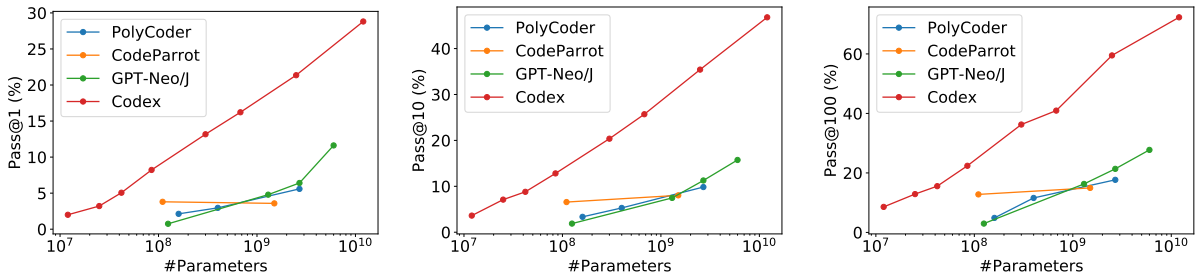
*Codex is initialized with another pretrained model, GPT-3.

Table 3.4: Results of different models on the HumanEval benchmark, and the number of different types of tokens seen during the training process.

substantially exceed ours. This in partly because they use a higher proportion of Python code (we aimed to balance data volume across programming languages), and in part because of resource limitations, which lead to PolyCoder not observing its entire training data. In addition, the natural language blend in the training corpus may help code language modeling as well, especially with code-related texts such as Stack Exchange dumps being included.

Compared to GPT-Neo (2.7B), PolyCoder has seen fewer Python tokens, but more code tokens in other programming languages, hinting that transfer from other languages to Python helps to achieve a similar performance. This suggests that future research could benefit from blending code in different programming languages, as well as natural language text.

Scaling Effect To further understand the effect of the number of model parameters with respect to HumanEval code completion performance, we show the Pass@1, Pass@10 and Pass@100 percentage with respect the the model size in Figure 3.4. We can see that the performance of the Codex models are significantly better than all the other open-source models across all numbers of parameters. The performance on HumanEval benchmark increases linearly with



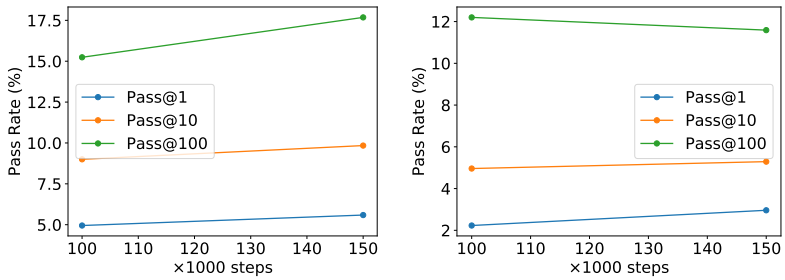
(a) Pass@1 (b) Pass@10 (c) Pass@100

Figure 3.4: The scaling effect of HumanEval performance on different models.

the magnitude (log scale) of the number of parameters in the model. Similar scaling effects could be found on PolyCoder and GPT-Neo/J models. Interestingly, the CodeParrot models that are trained only on Python seem to have reached a saturating performance with respect to increasing number of parameters, where the training corpus being focused on Python may have some effect. With higher number of parameters (2.7B), PolyCoder's performance is trending worse than that of GPT-Neo/J. Comparing GPT-Neo/J that is trained on Pile dataset containing a blend of text, Stack Exchange dumps and GitHub data, with PolyCoder that are trained on only GitHub repositories of popular programming languages, we hypothesize that the added text, especially texts in technical and software engineering domains, may be crucial for the larger model to boost the performance.

We compare the performance difference between the model trained after 100K steps versus the model after 150K steps in Figure 3.5, and find that training for longer helps the larger model more as it is still under-fitted. We can see that in the larger 2.7B model, by training the model longer till 150K steps, the performance increases uniformly, with Pass@100 increasing the most. However, for a smaller model such as the 400M model, by training the model longer till 100K steps, the improvements are subdued and Pass@100 drops. This suggests that with the larger model, training for longer may provide additional boost in performance. This echoes with the observation from the training curve (Figure 3.3) as well.

Temperature Effect All the above results are obtained by sampling the language model with different temperatures and picking the best value for each metric. We are also interested in how different choices of temperature affects the final generation quality. We summarize the results in Figure 3.6. The general trend is for Pass@1, lower temperatures are better, and for Pass@100, a higher temperature will help, while for Pass@10 a temperature in the middle is better suited. We



(a) 2.7B Model (b) 400M Model

Figure 3.5: HumanEval performance comparison after training the model for longer.

hypothesize that this is because a higher temperature during generation makes the model less confident in its predictions and thus allow for more exploration and more diverse outputs, resulting in better accuracy at Pass@100. Too high a temperature (0.8) is also hurtful if the model is capable enough.

On the contrary, a lower temperature makes the model output very confident in its prediction and thus will be better suited for generating very few correct examples, and thus the better performance for Pass@1.

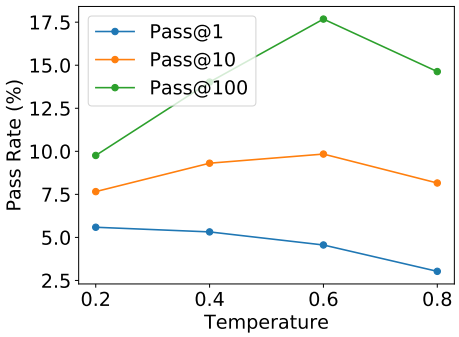
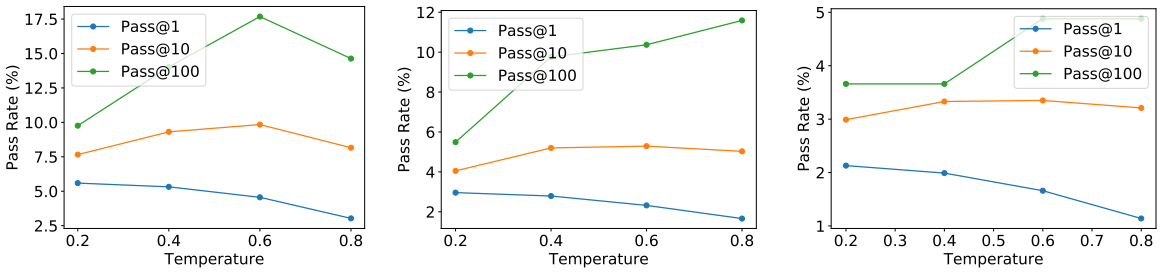


Figure 3.6: HumanEval performance with different softmax temperatures during generation.

We also show how temperature affects HumanEval performance on model of all three sizes in Figure 3.7. We find that for a larger model, e.g., the 2.7B model, a temperature as high as 0.8 is actually hurting the performance for Pass@100, suggesting that if the model is good enough, a very high temperature may cause the outputs to be too diverse, thus hurting the correctness. This suggests the importance of temperature and the need to tune it individually for different model capacity and different generation scenarios.

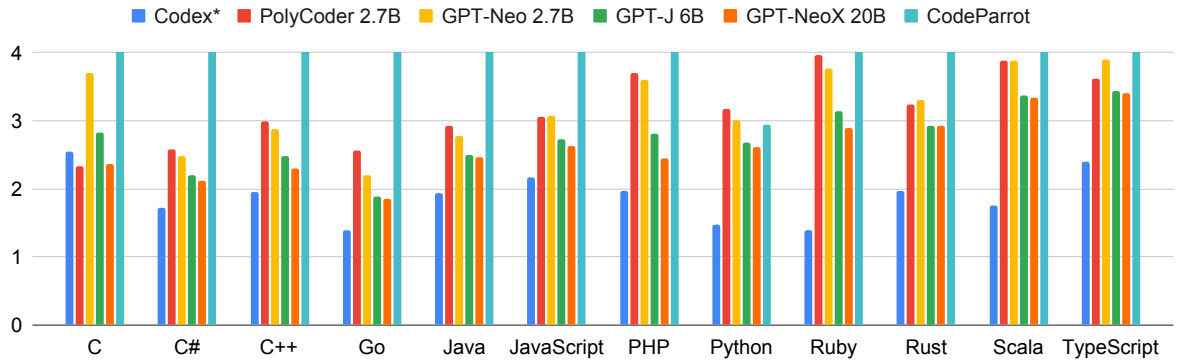
3.5.2 Intrinsic Evaluation

The perplexity results on the evaluation datasets are shown in Figure 3.8. We show the detailed perplexity of different models on different languages in Table 3.5. The number of tokens shown in the table is obtained after tokenizing the code in each language using their respective lexers, by Pygments. This number of tokens is used to normalize the perplexity scores to make them



(a) 2.7B Model (b) 400M Model (c) 160M Model

Figure 3.7: HumanEval performance using different softmax temperatures during generation.



* Since the exact training set of Codex is unknown, it may include files from these test sets rendering Codex's results overly-optimistic.

Figure 3.8: Perplexity comparison on our evaluation dataset of different models on different programming languages. Note that the y-axis is capped at 4; CodeParrot's entropy on all languages other than Python is much higher than shown here (see Table 3.5).

comparable across models. Note that CodeParrot is only trained on Python data and thus performs poorly in other languages.

The plot caps the perplexity score to 4 as CodeParrot performs poorly in languages other than Python. It is important to note that although Codex's perplexities are lower than other models in most languages, Codex might have been trained on the test sets, and its results are thus over-optimistic.

Notably, *PolyCoder outperforms Codex and all other models in the C language.* Comparing the open-source models only, PolyCoder performs better than the similarly sized GPT-Neo 2.7B in C, JavaScript, Rust, Scala and TypeScript.

In the other 11 languages other than C, all other open-source models, including ours, are

Language	#tokens	Codex*	PolyCoder 2.7B	GPT-Neo 2.7B	GPT-J 6B	GPT-NeoX	CodeParrot
C	55,333	2.55	2.33	3.69	2.82	2.37	19.23
C#	67,306	1.72	2.58	2.49	2.20	2.12	7.16
C++	69,627	1.95	2.99	2.87	2.47	2.32	8.48
Go	79,947	1.39	2.57	2.19	1.89	1.85	10.00
Java	65,484	1.94	2.92	2.78	2.49	2.47	6.79
JavaScript	54,620	2.17	3.06	3.07	2.73	2.62	9.23
PHP	45,682	1.98	3.70	3.61	2.81	2.45	19.91
Python	79,653	1.47	3.18	3.00	2.68	2.61	2.95
Ruby	46,537	1.39	3.96	3.77	3.13	2.89	14.26
Rust	107,717	1.96	3.24	3.30	2.92	2.92	8.68
Scala	65,756	1.75	3.87	3.88	3.37	3.33	12.91
TypeScript	55,895	2.40	3.61	3.90	3.43	3.41	12.54

* Since the exact training set of Codex is unknown, it might have been trained on these test sets, and Codex's results are over-optimistic.

Table 3.5: Perplexity of different models for different programming languages on our evaluation dataset.

significantly worse (higher perplexity) than Codex. We hypothesize that this is due to the fact that PolyCoder is trained on an imbalanced mixture of different languages, with C and C++ being closely related and the two most dominant in the entire training corpus (Section 3.4.2). Thus, the larger volume in total (because of long files) makes C the most "favored" language by PolyCoder. The reason why PolyCoder does not outperform Codex in C++ is possibly due to the complexity of C++ language and Codex's significantly longer context window size (4096, compared to PolyCoder's 2048), or because Codex is possibly trained on more C++ training data.

With the same pretraining corpus, the gain from a 2.7B model (GPT-Neo) to a 6B model (GPT-J) is significant over all languages. However, when increasing the model size further to 20B, the improvement varies across different languages. For example, the performance on Go, Java, Rust, Scala, TypeScript do not increase significantly when the model size increases by 3 times. This suggests that for some programming languages, and given the amounts of data, the capacity of GPT-J is sufficient. Interestingly, these languages seem to coincide with languages where PolyCoder outperforms a similarly sized model trained on Pile. This may hint that for the languages in which larger models do not provide additional gains, training the model only using

code may be enough or slightly more helpful than training on both natural language and code.

We can see that comparing different models, perplexity trends for Python correlates well with the HumanEval benchmark performance of the extrinsic evaluation (Section 3.5.1). This suggests that perplexity is a useful and low-cost metric to estimate other, downstream, metrics.

3.6 Conclusion

In this paper, we perform a systematic evaluation of large language models for code. The performance generally benefits from larger models and longer training time. We also believe that the better results of GPT-Neo over PolyCoder in some languages show that training on natural language text *and* code can benefit the modeling of code. To help future research in the area, we release PolyCoder, a large open-source language model for code, trained exclusively on code in 12 different programming languages. In the C programming language, *PolyCoder achieves lower perplexity than all models including Codex.*

Part II

Human Study of Code Generation Models

Chapter 4

In-IDE Code Generation from Natural Language: Promise and Challenges (Completed)

A great part of software development involves conceptualizing or communicating the underlying procedures and logic that needs to be expressed in programs. One major difficulty of programming is turning *concept* into *code*, especially when dealing with the APIs of unfamiliar libraries. Recently, there has been a proliferation of machine learning methods for code generation and retrieval from *natural language queries*, but these have primarily been evaluated purely based on retrieval accuracy or overlap of generated code with developer-written code, and the actual effect of these methods on the developer workflow is surprisingly unattested. In this paper, we perform the first comprehensive investigation of the promise and challenges of using such technology inside the PyCharm IDE, asking "at the current state of technology does it improve developer productivity or accuracy, how does it affect the developer experience, and what are the remaining gaps and challenges?" To facilitate the study, we first develop a plugin for the PyCharm IDE that implements a hybrid of code generation and code retrieval functionality, and orchestrate virtual environments to enable collection of many user events (e.g. web browsing, keystrokes, fine-grained code edits). We ask developers with various backgrounds to complete 7 varieties of 14 Python programming tasks ranging from basic file manipulation to machine learning or data visualization, with or without the help of the plugin. While qualitative surveys of developer experience are largely positive, quantitative results with regards to increased productivity, code quality, or program correctness are inconclusive. Further analysis

identifies several pain points that could improve the effectiveness of future machine learning based code generation/retrieval developer assistants, and demonstrates when developers prefer code generation over code retrieval and vice versa. We release all data and software (https://github.com/neulab/tranx-study) to pave the road for future empirical studies on this topic, as well as development of better code generation models.

4.1 Introduction

One of the major hurdles to programming is the time it takes to turn ideas into code [238]. All programmers, especially beginners but even experts, frequently reach points in a program where they understand conceptually what must be done next, but do not know how to create a concrete implementation of their idea, or would rather not have to type it in if they can avoid it. The popularity of the Stack Overflow Q&A website is a great example of this need. Indeed, developers ask questions about how to transform ideas into code all the time, *e.g.*, "How do I check whether a file exists without exceptions?,"[1] "How can I merge two Python dictionaries in a single expression?,"[2] etc. Moreover, this need is likely to continue in the future, as new APIs appear continuously and existing APIs change in non-backwards compatible ways [241], requiring recurring learning effort [175, 244].

Despite early skepticism towards the idea of "natural language programming" [74], researchers now widely agree on a range of scenarios where it can be useful to be able to formulate instructions using natural language and have the corresponding source code snippets automatically produced. For example, software developers can save keystrokes or avoid writing dull pieces of code [88, 250, 296, 359]; and non-programmers and practitioners in other fields, who require computation in their daily work, can get help with creating data manipulation scripts [107, 187].

Given a natural language query carrying the intent of a desired step in a program, there are two main classes of methods to obtain code implementing this intent, corresponding to two major research thrusts in this area. On the one hand, *code retrieval* techniques aim to search for and retrieve an existing code fragment in a code base; given the abundance of code snippets online, on platforms such as Stack Overflow, it is plausible that a lot of the code that one might write, especially for lower level functionality and API usage primitives, already exists somewhere, therefore the main challenge is search. On the other hand, *code generation* techniques aim to

[1] https://stackoverflow.com/q/82831
[2] https://stackoverflow.com/q/38987

synthesize code fragments given natural language descriptions of intent. This is typically a harder challenge than retrieval and therefore more ambitious, but it may be particularly useful in practice if those exact target code fragments do not exist anywhere yet and can be generated instead.

The early attempts at general-purpose code generation from natural language date back to the early to mid 2000s, and resulted in groundbreaking but relatively constrained grammatical and template-based systems, *e.g.*, converting English into Java [276] and Python [345]. Recent years have seen an increase in the scope and diversity of such programming assistance tools, as researchers have devised code generation techniques that promise to be more flexible and expressive using machine (deep) learning models trained on data from "Big Code" repositories like GitHub and Stack Overflow; see Allamanis et al. [10] for an excellent survey of such techniques. Code retrieval systems have also improved dramatically in recent years, thanks to the increasing availability of source code online and more sophisticated information retrieval and machine learning techniques; perhaps the most popular current code retrieval system is Microsoft's Bing Developer Assistant [359], which is an adaptation of the Bing search engine for code.

While both types of methods (generation and retrieval) for producing appropriate code given natural language intents have received significant interest in machine learning circles, there is a surprising paucity of research using human-centered approaches [246] to evaluate the usefulness and impact of these methods *within the software development workflow*. An important open question is to what extent the typically high accuracy scores obtained during automatic evaluations on benchmark datasets will translate to real-world usage scenarios, involving software developers completing actual programming tasks. The former does not guarantee the latter. For example, an empirical study on code migration by Tran et al. [341] showed that the BLEU [265] accuracy score commonly used in natural language machine translation has only weak correlation with the semantic correctness of the translated source code [341].

In this paper, we take one step towards addressing this gap. We implemented two state-of-the-art systems for natural language to code (NL2Code) generation and retrieval as in-IDE developer assistants, and carried out a controlled human study with 31 participants assigned to complete a range of Python programming tasks *with and without* the use of the two varieties of NL2Code assistance. Our results reveal that while participants in general enjoyed interacting with our IDE plugin and the two code generation and retrieval systems, surprisingly *there were no statistically significant gains in any measurable outcome when using the plugin.* That is, tasks with code fragments automatically generated or retrieved using our plugin were, on

average, neither completed faster nor more correctly than tasks where participants did not use any NL2Code assistant. This indicates that despite impressive improvements in the intrinsic performance of code generation and retrieval models, there is a clear need to further improve the accuracy of code generation, and we may need to consider other extrinsic factors (such as providing documentation for the generated code) before such models can make sizable impact on the developer workflow.

In summary, the **main contributions** of this paper are: (i) A hybrid code generation and code retrieval plugin for the Python PyCharm IDE, that takes as input natural language queries. (ii) A controlled user study with 31 participants observed across 7 types of programming tasks (14 concrete subtasks). (iii) An analysis of both quantitative and qualitative empirical data collected from the user study, revealing how developers interact with the NL2Code assistant and the assistant's impact on developer productivity and code quality. (iv) A comparison of code snippets produced by the two models, generation versus retrieval. (v) An anonymized dataset of events from our instrumented IDE and virtual environment, capturing multiple aspects of developers' activity during the programming tasks, including plugin queries and edits, web browsing activities, and code edits.

4.2 Overview of Our Study

The goal of our research is to elucidate to what extent and in what ways current natural language programming techniques for code generation and retrieval can be useful within the development workflow as NL2Code developer assistants. Our main interest is evaluating the usefulness in practice of state-of-the-art NL2Code *generation* systems, which have been receiving significant attention from researchers in recent years, but have so far only been evaluated on benchmark datasets using standard NLP metrics. However, as discussed above, code generation and code retrieval are closely related problems, with increasingly blurred lines between them; *e.g.*, recent approaches to align natural language intents with their corresponding code snippets in Stack Overflow for retrieval purposes [388] use similar deep learning technology as some code generation techniques [383]. Therefore, it is important to also consider code retrieval systems when experimenting with and evaluating code generation systems.

Given this complementarity of the two tasks, we select as a representative example of state-of-the-art techniques for code generation the semantic parsing approach by Yin and Neubig [383]. In short, the approach is based on a tree-based neural network model that encodes natural language utterances and generates corresponding syntactically correct target code snippets;

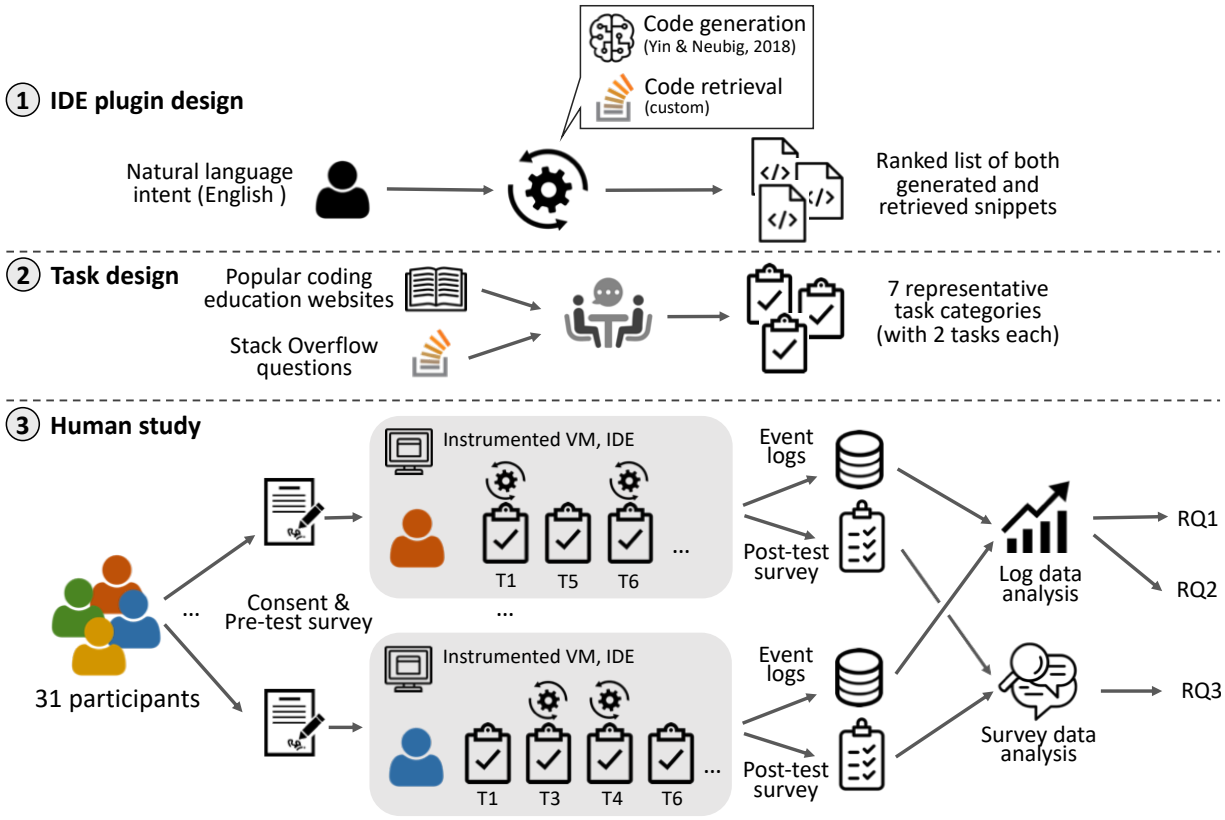


Figure 4.1: Overview of our study.

for example, the model can generate the Python code snippet "`x.sort(reverse=True)`" given the natural language input "sort list `x` in reverse order". We chose the approach by Yin and Neubig [383] over similar approaches such as those of Iyer et al. [139] and Agashe et al. [1] as it is the most general purpose and most naturally comparable to code retrieval approaches; see Section 4.9 for a discussion. For code retrieval, the closest analogue is Microsoft's proprietary Bing Developer Assistant [359], which takes English queries as input and returns existing matching code fragments from the Web, using the Bing search engine. However, given the proprietary nature of this system, we build a custom Stack Overflow code search engine inspired by it rather than use the system itself.

We then designed and carried out the controlled human study summarized in Figure 4.1. First, we implement the two code generation and retrieval techniques as a custom plugin for the PyCharm[3] IDE, which takes as input natural language text intents and displays as output the corresponding code snippets generated and retrieved by the respective underlying models.

[3] https://www.jetbrains.com/pycharm/

Second, we compile 14 representative Python programming tasks across 7 task categories with varying difficulty, ranging from basic Python to data science topics. Third, we recruit 31 participants with diverse experience in programming in Python and with the different task application domains. Then, using an instrumented virtual environment and our IDE plugin, we collect quantitative and qualitative data about task performance and subjective tool use from each participant, as well as over 170 person hours of telemetry data from the instrumented environment.

Finally, we analyze these data to answer three research questions, as follows.

RQ$_1$. *How does using a NL2Code developer assistant affect task completion time and program correctness?* This research question investigates quantitatively differences in outcome variables between tasks completed in the treatment and control conditions. To this end, we use the log data from our instrumented virtual environment to compute task completion times, and rubric-based manual scoring of the solutions submitted by study participants to evaluate program correctness. Then, we use multivariate mixed-effects regression modeling to analyze the data. We expect that using the plugin developers can complete tasks faster, without compromising solution quality.

RQ$_2$. *How do users query the NL2Code assistant, and how does that associate with their choice of generated vs retrieved code?* This research question investigates quantitatively three dimensions of the inputs and outputs of the NL2Code plugin. Again using log data from our instrumented virtual environment, we first model how the natural language input queries differ when study participants favor the code snippets returned by the code generation model over those returned by the code retrieval model. Second, we evaluate the quality of the natural language queries input by study participants in terms of their ability to be answerable by an oracle (human expert), which is also important for the success of NL2Code systems in practice, in addition to the quality of the underlying code generation or retrieval systems. Third, we study how the length and the frequency of different types of tokens changes after study participants edit the candidate code snippets returned by the NL2Code plugin, which could indicate ways in which even the chosen code snippets are still insufficient to address the users' needs.

RQ$_3$. *How do users perceive the usefulness of the in-IDE NL2Code developer assistant?* Finally, this research question investigates qualitatively the experience of the study participants interacting with the NL2Code plugin and underlying code generation and retrieval models.

In the remainder of this paper, Sections 4.3–4.4 describe our study setup in detail; then Sections 4.5–4.7 present our answers to the research questions; Section 4.8 discusses implications; and Section 4.9 discusses related work.

Following best practices for empirical software engineering research [326, 360], we make our study replicable, publishing our plugin prototype, instrumented virtual environment, data extraction and analysis scripts, and the obtained anonymized raw data; see the online appendices at https://github.com/neulab/tranX-plugin and https://github.com/neulab/tranX-study.

4.3 NL2Code IDE Plugin Design

We designed and built a joint NL2Code generation and retrieval plugin for PyCharm, a popular Python IDE. Our plugin is open source and available online.[4] As mentioned above, the plugin takes as input an English query describing the user's intent, and gives as output a ranked list of the most relevant code snippets produced by each of the two underlying code generation and retrieval systems. Using IDE plugins to query Web resources such as Stack Overflow is expected to be less disruptive of developers' productivity than using an external Web browser, since it reduces context switching [22, 273]. Moreover, there exist already a number of IDE plugins for Web / Stack Overflow search and code retrieval [50, 273, 292, 359], therefore the human-computer interaction modality should feel at least somewhat natural to study participants.

The Underlying Code Generation System. For code generation, we use the model by Xu et al. [364] (available online[5]), which is an improved version of the tree-based semantic parsing model by Yin and Neubig [385], further pre-trained on official API documentation in addition to the original training on Stack Overflow questions and answers.[6]

This model reports state-of-the-art accuracy on the CoNaLa benchmark dataset [388], a benchmark dataset of intent/code pairs mined from Stack Overflow and standardly used to evaluate code generation models. Accuracy is computed using the BLEU score [265], a standard metric used in the NLP community, that measures the token-level overlap between the generated code and a reference implementation. As discussed above, the BLEU score (and similar automated metrics) are typically not sufficiently sensitive to small lexical differences in token sequence that can greatly alter the semantics of the code [341], hence our current human-centered study. Still, qualitatively, it appears that the model can generate reasonable code fragments given short text inputs, as shown in Table 4.1. Note how the model can generate syntactically correct code

[4] At https://github.com/neulab/tranX-plugin

[5] https://github.com/neulab/external-knowledge-codegen

[6] We deployed the model on an internal research server and exposed a HTTP API that the plugin can access; queries are fast enough for the plugin to be usable in real time.

Open a file "`f.txt`" in write mode. ✓ `f = open('f.txt', 'w')` ♣ `f = open('f.txt', 'w')` ♠ `with open("users.txt", "a")as f: f.write(username + "\n")`
Remove first column of dataframe *df*. ✓ `df = df.drop(df.columns[[0]], axis=1)` ♣ `df.drop(df.columns[[0]])` ♠ `del df['column_name']`
Lower a string *text* and remove non-alphanumeric characters aside from space. ✓ `re.sub(r'[^\sa-zA-Z0-9]', '', text).lower().strip()` ♣ `re.sub(r'[^\sa-zA-Z0-9]', '', text)` ♠ `re.sub(r'[^\sa-zA-Z0-9]', '', text).lower().strip()`

Table 4.1: Examples, where ✓ is the ground-truth code snippet, ♣ is the output from the state-of-the-art code generation model, and ♠ is the first candidate retrieved from Stack Overflow using Bing Search.

snippets by construction; demonstrates ability to identify and incorporate a wide variety of API calls; and also has the ability to copy important information like string literals and variable names from the input natural language intent, in contrast to the code retrieval results. When displaying multiple generation results in the plugin described below, these results are ordered by the conditional probability of the generated code given the input command.

The Underlying Code Retrieval System. For code retrieval, similarly to a number of recent works on the subject [50, 273, 359], we implement a wrapper around a general-purpose search engine, specifically the Bing[7] search engine.[8] The wrapper queries this search engine for relevant questions on Stack Overflow,[9] the dominant programming Q&A community, and retrieves code from the retrieved pages. A dedicated search engine already incorporates advanced indexing and ranking mechanisms in its algorithms, driven by user interaction data, therefore it is preferable

[7] https://www.bing.com/

[8] We chose Bing rather than other alternatives such as Google due to the availability of an easily accessible search API.

[9] https://stackoverflow.com/

to using the internal Stack Overflow search engine directly [359].

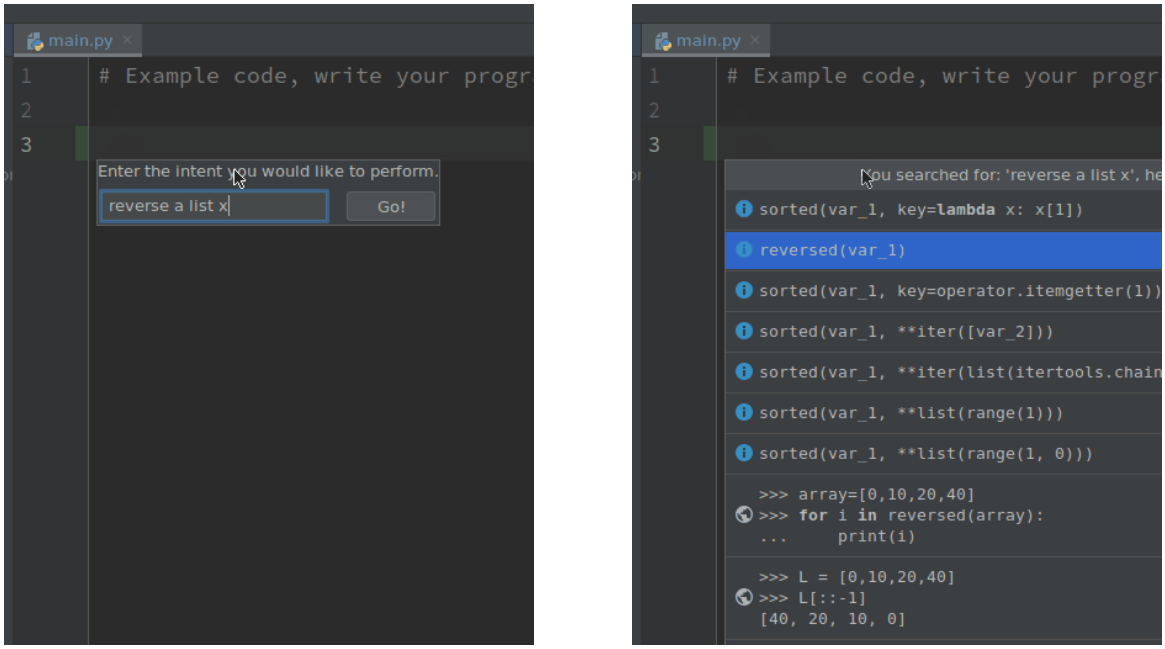
Specifically, we add the "Python" prefix to all user queries to confine the search to the Python programming language domain, and add "site:stackoverflow.com" to confine the results to the Stack Overflow platform. We do not structurally alter the queries otherwise, *e.g.*, we do not remove variables referenced therein, if any, although we do strip away grave accents that are part of the code generation model's syntax.[10] For the query example mentioned above, the actual query string for Bing search would become "Python reverse a list x site:stackoverflow.com". For each Stack Overflow question page retrieved, we then extract the code snippets from the top 3 answers into a ranked list, sorted descending by upvotes. The code snippet extraction procedure follows Yin et al. [388] for identifying the code part of the answer, based on Stack Overflow-specific syntax highlighting and heuristics. When displaying multiple retrieval results, these results are ordered by the order they appeared in Bing search engine results and the ordering of answers inside SO posts is done by upvotes.

Table 4.1 shows a few example outputs. Note how the retrieval results sometimes contain spurious code, not part of the natural language intent (first example), and otherwise seem to complement the generation results. Indeed, in the second example the generation result is arguably closer to the desired answer than the retrieval result, with the opposite situation in the third example.

Interacting With the Plugin. Figure 4.2 illustrates the plugin's user interface. The user first activates the query interface by pressing a keyboard shortcut when the cursor is in the IDE's editor. A popup appears at the current cursor position (Figure 4.2a), and the user can enter a command in natural language that they would like to be realized in code (*e.g.*, "reverse a list `x`"[11]). The plugin then sends the request to the underlying code generation and code retrieval systems, and displays a ranked list of results, with the top 7 code generation results at the top,

[10]To mitigate concerns that user queries using the specified syntax (command form sentences and including variable names) may adversely affect the retrieval results, after the full study was complete we modified 59 user-issued queries that were indeed complete sentences with full variable names, converting them into short phrases without variable names and re-ran the retrieval. We then compared the results and manually annotated the number of times the search engine returned a result that we judged was sufficient to understand how to perform the programming task specified by the user's intent. As a result, the user-written full intent resulted in a sufficient answer 34/59 times, and the simplified intent without variable names returned a sufficient answer 36/59 times, so it appears that including variable names has a marginal to no effect on whether the search engine was able to provide a good top-1 result. We also measured the exact-match overlap between the top-1 results, and found it to be 22/59, and overlap between the top-7 result lists was 182/(59*7).

[11]Note the special syntax used to mark explicit variables; see Appendix 4.11.6 for full syntax details.



(a) Query input interface

(b) Code snippet candidates

Figure 4.2: Screenshots of the in-IDE plugin taking a natural language query as input and listing code snippet candidates from both code generation and code retrieval.

followed by the top 7 code retrieval results (Figure 4.2b); 14 results are displayed in total.[12]

The number 7 was chosen subjectively, trying to maximize the amount and diversity of resulting code snippets while minimizing the necessary screen space to display them and, therefore, the amount of scrolling expected from study participants looking to inspect all the plugin-returned results. After completing the current study, we found that the most relevant code snippets are typically within the top 3 results, and thus a smaller number of candidates may be sufficient. While the number and ordering of candidates has the potential to have a significant impact on the efficiency and efficacy of the developer assistant, a formal evaluation of this impact is beyond the scope of this work.

If a code snippet is selected, the code snippet is then inserted in the current cursor's position in the code editor. The user's selection is also recorded by our instrumentation in the back end. Understandably, some returned code snippets may not be directly suitable for the context inside

[12]We note that the main motivation for this ordering is that the generation results tend to be significantly more concise than the retrieval results (Figure 4.6). If we put the retrieval results first it is likely that the users would rarely scroll past the retrieval results and view the generation results due to issues of screen real-estate. It is important to consider that alternative orderings may result in different experimental results, although examining alternate orderings was not feasible within the scope of the current study.

(a) Generated code with errors in the context

(b) The user fixes the error and uploads

Figure 4.3: Screenshots of fixing the small errors in generated code and upload the correct snippet.

the editor, so the user is welcome (and encouraged by the instructions we give as part of our human study) to edit the auto-inserted code snippets to fit their specific intent. After the edit is done, the user is asked to upload their edits to our server, along with the context of the code, using a dedicated key combination or the IDE's context menu. The process is illustrated in Figure 4.3. The edit data enable us to analyze how many and what kind of edits the users need to make to transform the auto-generated code to code that is useful in their context.[13]

4.4 Human Study Design

Given our NL2Code joint code generation and retrieval IDE plugin above, we designed and carried out a human study with 31 participants assigned to complete a range of Python programming tasks in both control (no plugin) and treatment (plugin) conditions.

4.4.1 Task Design

To emulate real world Python development activities, but also fit within the scope of a user study, we compiled a set of 14 reasonably sized Python programming tasks, organized into 7 categories (2 tasks per category) that span a diversity of levels of difficulty and application domains.

We started by identifying representative task categories that many users would encounter in practice. To that end, we analyzed two sources. First, we manually reviewed all the Python pro-

[13]The edit data may also be helpful as training data for improving code generation and retrieval models. We release our data publicly to encourage this direction in future work.

gramming courses listed on three popular coding education websites, Udacity,[14] Codecademy,[15] and Coursera,[16] to identify modules commonly taught across all websites that indicate common usage scenarios of the Python language. Second, we cross checked if the previously identified use cases are well represented among frequently upvoted questions with the [python] tag on Stack Overflow, which would further indicate real programmer needs. By searching the category name, we found that each of our identified categories covers more than 300 questions with more than 10 upvotes on Stack Overflow. We iteratively discussed the emerging themes among the research team, refining or grouping as needed, until we arrived at a diverse but relatively small set of use cases, covering a wide range of skills a Python developer may need in practice.

In total, we identified 7 categories of use cases, summarized in Table 4.2. For each of the 7 categories, we then designed 2 tasks covering use cases in the most highly upvoted questions on Stack Overflow. To this end, we searched Stack Overflow for the "python" keyword together with another keyword indicative of the task category (*e.g.*, "python matplotlib," "python pandas"), selected only questions that were asking how to do something (*i.e.*, excluding questions that ask about features of the language, or about how to install packages), and drafted and iteratively refined after discussion among the research team tasks that would cover 3-5 of the most frequently upvoted questions.

We illustrate this process with the following example task for the "Data visualization" category:[17]

> By running `python3 main.py`, draw a scatter plot of the data in `shampoo.csv` and save it to `shampoo.png`. The plot size should be 10 inches wide and 6 inches high. The `Date` column is the x axis (some dates are missing from the data and in the plot the x axis should be completed with all missing dates without sales data). The date string shown on the plot should be in the format `(YYYY-MM-DD)`. The `Sales` column is the y axis. The graph should have the title "Shampoo Sales Trend". The font size of the title, axis labels, and x & y tick values should be 20pt, 16pt, and 12pt respectively. The scatter points should be colored `purple`.

This task covers some of the top questions regarding data visualization with `matplotlib` found on Stack Overflow through the approach described above:

1. How do you change the size of figures drawn with matplotlib?[18]

[14] https://www.udacity.com/courses/all
[15] https://www.codecademy.com/catalog
[16] https://www.coursera.org/
[17] Corresponding to the search https://stackoverflow.com/search?tab=votes&q=python%20matplotlib.
[18] https://stackoverflow.com/questions/332289/how-do-you-change-the-size-of-figures-drawn-with-matplotlib

2. How to put the legend out of the plot?[19]
3. Save plot to image file instead of displaying it using Matplotlib?[20]
4. How do I set the figure title and axes labels font size in Matplotlib?[21]

For each task designed, we also provide the user with required input data or directory structure for their program to work on, as well as example outputs (console print-outs, output files & directories, etc.) so that they could verify their programs during the user study.

Table 4.2 summarizes the 14 tasks. The full task descriptions and input/output examples can be found online, as part of our replication package at https://github.com/neulab/tranx-study. The tasks have varying difficulties, and on average each task would take about 15-40 minutes to complete.

Table 4.2: Overview of our 14 Python programming tasks.

Category	*Tasks*	
Basic Python	T1-1	Randomly generate and sort numbers and characters with dictionary
	T1-2	Date & time format parsing and calculation with timezone
File	T2-1	Read, manipulate and output CSV files
	T2-2	Text processing about encoding, newline styles, and whitespaces
OS	T3-1	File and directory copying, name editing
	T3-2	File system information aggregation
Web Scraping	T4-1	Parse URLs and specific text chunks from web page
	T4-2	Extract table data and images from Wikipedia page
Web Server & Client	T5-1	Implement an HTTP server for querying and validating data
	T5-2	Implement an HTTP client interacting with given blog post APIs
Data Analysis & ML	T6-1	Data analysis on automobile data of performance metrics and prices
	T6-2	Train and evaluate a multi-class logistic regression model given dataset
Data Visualization	T7-1	Produce a scatter plot given specification and dataset
	T7-2	Draw a figure with 3 grouped bar chart subplots aggregated from dataset

[19] https://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot
[20] https://stackoverflow.com/questions/9622163/save-plot-to-image-file-instead-of-displaying-it-using-mat
[21] https://stackoverflow.com/questions/12444716/how-do-i-set-the-figure-title-and-axes-labels-font-size-i

4.4.2 Participant Recruitment & Task Assignments

Aiming to recruit participants with diverse technical backgrounds but at least some programming experience and familiarity with Python so as to be able to complete the tasks, we advertised our study in two ways: (1) inside the university community through personal contacts, mailing lists, and Slack channels, hoping to recruit researchers and students in computer science or related areas; (2) on the freelancer platform Upwork,[22] hoping to attract participants with software engineering and data science experience. We promised each participant US$5 per task as compensation; each participant was expected to complete multiple tasks.

To screen eligible applicants, we administered a pre-test survey to collect their self-reported levels of experience with Python and with each of the 7 specific task categories above; see Appendix 4.11.2 for the actual survey instrument. We only considered as eligible those applicants who reported at least some experience programming in Python, *i.e.*, a score of 3 or higher given the answer range [1: very inexperienced] to [5: very experienced]; 64 applicants satisfied these criteria.

We then created personalized task assignments for each eligible applicant based on their self reported levels of experience with the 7 specific task categories (see Appendix 4.11.3 for the distributions of participants' self reported experience across tasks), using the following protocol:

1. To keep the study relatively short, we only assign participants to a total of 4 task categories (8 specific tasks, 2 per category) out of the 7 possible.
2. Since almost everyone eligible for the study reported being at least somewhat experienced with the first 2 task categories (Basic Python and File), we assigned everyone to these 2 categories (4 specific tasks total). Moreover, we assigned these 2 categories first and second, respectively.
3. For the remaining 5 task categories, sorted in increasing complexity order,[23] we rank them based on a participant's self reported experience with that task genre, and then assign the participant to the top 2 task categories with most experience (another 4 specific tasks total).

Note that this filtering by experience is conducive to allowing participants to finish the tasks in a reasonable amount of time, and reflective of a situation where a developer is working in their domain of expertise. However, at the same time it also means that different conclusions might be reached if novice programmers or programmers without domain expertise used the plugin

[22] https://www.upwork.com/

[23] The task identifiers in Table 4.2 reflect this order.

instead.

Next, we randomly assigned the first task in a category to either the treatment condition, *i.e.*, the NL2Code plugin is enabled in the virtual environment IDE and the participants are instructed to use it,[24] or the control condition, *i.e.*, the NL2Code plugin is disabled. The second task in the same category is then automatically assigned to the other condition, *e.g.*, if the plugin is on for task1-1, it should be off for task1-2. Therefore, each participant was asked to complete 4 tasks out of 8 total using the NL2Code plugin, and 4 without.

Finally, we invited all eligible applicants to read the detailed study instructions, access the virtual environment, and start working on their assigned tasks. Only 31 out of the 64 eligible applicants after the pre-test survey actually completed their assigned tasks.[25] Their backgrounds were relatively diverse; of the 31 participants, 12 (39%) were software engineers and 11 (35%) were computer science students, with the rest being researchers (2, 6%), and other occupations (6, 19%). Our results below are based on the data from these 31 participants.

4.4.3 Controlled Environment

Participants worked on their assigned tasks inside a custom instrumented online virtual environment, accessible remotely. Our virtual machine is preconfigured with the PyCharm Community Edition IDE[26] and the Firefox Web browser; and it has our NL2Code plugin either enabled or disabled inside the IDE, depending on the condition. See Appendix 4.11.1 for complete technical details.

In addition, the environment logs all of the user's interactions with the plugin in the PyCharm IDE, including queries, candidate selections, and edits; all of the user's fine-grained IDE editor activities; the user's Web search/browsing activities inside Firefox; all other keystrokes inside the VM; and the source code for each one of the user's completed tasks.

To get a sense of how the source code evolves, whenever the user does not make modifications to the code for at least 1.5 seconds, the plugin also automatically uploads the current snapshot of the code to our server. The intuition behind this heuristic is that after a user makes some

[24] Despite these instructions, some participants did not use the plugin even when it was available and when instructed. We discovered this while analyzing the data collected from the study and filtered out 8 participants that did not use the plugin at all. They do not count towards the final sample of 31 participants we analyze data from, even though they completed tasks.

[25] Note that 4 of the 31 participants did not complete all 8 of their assigned tasks. We include their data from the tasks they completed and do not consider the tasks they did not finish.

[26] https://www.jetbrains.com/pycharm/download/

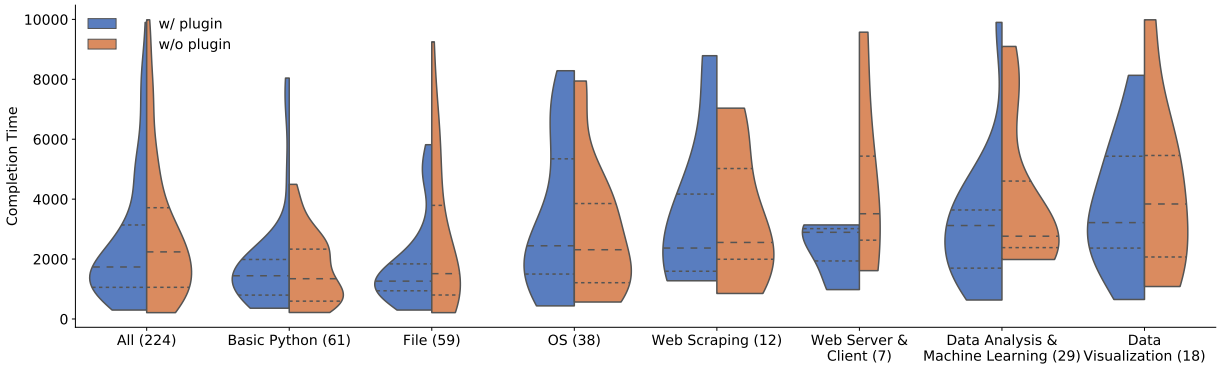


Figure 4.4: Distributions of task completion times (in seconds) across tasks and conditions (w/ and w/o using the plugin). The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

type of meaningful edit, such as adding or modifying an argument, variable, or function, they usually pause for a short time before the next edit. This edit activity granularity can be more meaningful than keystroke/character level, and it is finer grained than intent level or commit level edits.

Given that it is identifiable, we record participants' contact information (only for compensation purposes) separately from their activity logs. This Human Subjects research protocol underwent review and was approved by the Carnegie Mellon University Institutional Review Board.

4.4.4 Data Collection

To answer our research questions (Section 4.2), we collect the following sets of data.

Task Performance Data (RQ_1). The first research question compares measurable properties of the tasks completed with and without the help of or NL2Code IDE plugin and its underlying code generation and code retrieval engines. One would expect that if such systems are useful in practice, developers would be able to complete programming tasks faster without compromising on output quality. To investigate this, we measure two variables related to how well study participants completed their tasks and the quality of the code they produced:

- *Task Completion Time.* Since all activity inside the controlled virtual environment is logged, including all keystrokes and mouse movements, we calculate the time interval between when a participant started working on a task (first keystroke inside the IDE) and when they uploaded their final submission to our server.

Recall that participants worked asynchronously and they may have decided to take breaks; we designed our virtual environment to account for this, with explicit pause/resume functionality. To account for possible breaks and obtain more accurate estimates of time spent on task, we further subtract the time intervals when participants used our explicit pause/resume functionality, as well as all intervals of idle time in which participants had no mouse or keyboard activity for 2 minutes or more (they may have taken a break without recording it explicitly).

Figure 4.4 shows the distributions of task completion times across the two conditions (with and without the plugin).

- *Task Correctness.* Following the common practice in computer science education [51, 69, 104], we design a rubric for each task concurrently with designing the task, and later score each submission according to that rubric. We weigh all tasks equally, assigning a maximum score of 10 points to each. For each task, the rubric covers both basic aspects (*e.g.*, runs without errors/exceptions; produces the same output as the example output provided in the task description) as well as implementation details regarding functional correctness (*e.g.*, considers edge cases, implements all required functionality in the task description).

  For example, for the data visualization task described in Section 4.4.1, we created the following rubric, with the number in parentheses representing the point value of an item, for a total of 10 points: (i) Runs without errors (2); (ii) Correct image output format (png) (2); (iii) Read in the raw data file in correct data structure (1); (iv) Correct plot size (1); (v) Correctly handle missing data points (1); (vi) Date (x axis) label in correct format (1); (vii) Title set correctly (1); (viii) Font size and color set according to specification (1).

  To reduce subjectivity, we graded each submission blindly (*i.e.*, not knowing whether it came from the control or treatment condition) and we automated rubric items when possible, *e.g.*, using input-output test cases for the deterministic tasks and checking if the abstract syntax tree contains nodes corresponding to required types (data structures) such as dictionaries. See our online appendix[27] for the complete rubrics and test cases for all tasks.

  Figure 4.5 shows the distributions of scores across tasks, between the two conditions.

Plugin Queries, Snippets and User Edits (RQ_2). We record user queries using the plugin,

[27] https://github.com/neulab/tranx-study/blob/master/rubrics.md

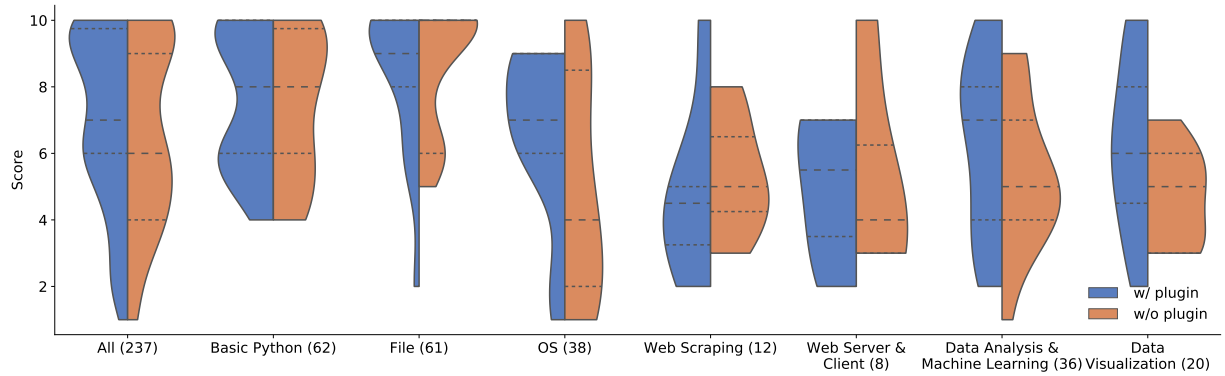


Figure 4.5: Distributions of task correctness scores (0–10 scale) across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

both the generated and retrieved code snippet candidates returned for the query, and the user selection from the candidates to insert into their source code. We use the data to analyze the NL queries and whether users preferred to use generated vs. retrieved code. In addition, we also record the user edits after inserting the code snippet from the plugin, along with the code context for the analysis on post edits required after using the plugin.

Participant Perceptions of Tool Use (RQ_3). We ran short post-test surveys after every task and a final post-test survey at the end of the study as a whole (see Appendix 4.11.4 for instruments) to collect data on the participants' subjective impressions of using the NL2code plugin and interacting with the code generation and code retrieval systems. We asked Likert-style and open-ended questions about aspects of using the plugin the participants enjoyed, and aspects they wish to see improved.

Next we describe how we analyzed these data and we answer each of our research questions.

4.5 RQ_1: NL2Code Plugin Effects on Task Completion Time and Program Correctness

We start by describing our shared data analysis methodology, applied similarly to both variables corresponding to RQ_1, then present our results for each variable.

Methodology. Recall, we assign each participant a total of 8 tasks, 2 per task category, based on their experience levels with those categories; in each category, we randomly assign one of the 2 tasks to the NL2Code plugin (treatment) condition and the other task to the no plugin (control) condition. We then compute the three sets of outcome variables above.

The key idea behind our analysis is to compare the distributions of outcome variables between tasks completed in the treatment and control conditions. However, this comparison is not straightforward. First, our study design imposes a hierarchical structure during data collection, therefore the individual observations are not independent—by construction, the same participant will have completed multiple tasks over the course of the study. Moreover, tasks vary in difficulty, again by construction, therefore it is expected that their corresponding response variables, *e.g.*, task completion times, can be correlated with the tasks themselves; *e.g.*, on average, more complex tasks will take longer to complete. Finally, the participants vary in their self reported levels of Python and individual task category experience; we should separate experience-related effects from effects of using the plugin, if any.

Therefore, we use mixed-effects [95] as opposed to the more common fixed-effects regression models to analyze our data. Fixed-effects models assume that residuals are independently and identically distributed, which is an invalid assumption in our case given the hierarchical nature of our data: *e.g.*, responses for the different measurement occasions (tasks) within a given individual are likely correlated; a highly experienced Python programmer completing one task quickly is more likely to complete other tasks quickly as well. Mixed-effects models address this issue by having a residual term at each level, *e.g.*, the observation level and the study participant level, in which case the individual participant-level residual is the so-called random effect. This partitions the unexplained residual variance into two components: higher-level variance between higher-level entities (study participants) and lower-level variance within these entities, between measurement occasions (tasks).

We consider two model specifications for each response variable. Our default model includes random effects for the individual and task, per the rationale above, a fixed effect for task category experience (*e.g.*, participants with more machine learning experience should complete the machine learning task faster, on average), and a dummy variable to indicate the condition (plugin vs no plugin). For example, for the task completion time response, we estimate the model:[28]

$$\texttt{completion\_time} = \texttt{experience} + \texttt{uses\_plugin} + (1|\texttt{user}) + (1|\texttt{task}) \tag{4.1}$$

As specified, our default model may suffer from heterogeneity bias [29]. Task category experience, a higher-level (*i.e.*, individual-level as opposed to observation-level) predictor, varies both within and across study participants: within participants, `experience` can vary across the 4 task categories—a user may be more experienced with basic Python than with data science;

[28] We are using the R syntax to specify random effects.

and across participants, `experience` with any given task category is likely to vary as well—some participants report higher experience with data science-related tasks than others. This means that `experience` (a fixed effect) and `user` (a random effect) may be "correlated." In turn, this may result in biased estimates, because both the within- and between-effect are captured in one estimate.

There are two sources of variation that can be used to explain changes in the outcome: (1) overall, more experienced programmers may be more efficient at completing tasks (group-level pattern); and (2) when becoming more experienced, programmers may also become more efficient at completing tasks (individual-level pattern). Therefore, to address potential heterogeneity bias, we split our fixed effect (`experience`) into two variables, each representing a different source of variation: a participant's average experience across all task categories (`experience_btw`), and the deviation for each task from the participants's overall mean experience (`experience_wi`). This process is known as de-meaning or person-mean centering [95]. This way, mixed-effects models can model both within- and between-subject effects [29], as recommended for a long time by Mundlak [240]. Taking the same task completion time response variable as an example (other variables are modeled analogously), our refined model becomes:

$$\texttt{completion\_time} = \texttt{experience\_btw} + \texttt{experience\_wi} + \texttt{uses\_plugin} + (1|\texttt{user}) + (1|\texttt{task}) \quad (4.2)$$

In both cases, the estimated coefficient for `uses_plugin` indicates the effect of using the plugin, *while holding fixed the effects of experience and other random user and task effects.*

For estimation we used the functions `lmer` and `lmer.test` in R. We follow the traditional level for statistical significance when interpreting coefficient estimates, *i.e.*, $p < 0.05$. As indicators of goodness of fit, we report a marginal (R^2_m) and a conditional (R^2_c) coefficient of determination for generalized mixed-effects models [151, 247], as implemented in the `MuMIn` package in R: R^2_m describes the proportion of variance explained by the fixed effects alone; R^2_c describes the proportion of variance explained by the fixed and random effects together.

Threats to Validity. Besides potential threats to statistical conclusion validity arising from the very nature of the data we are regressing over, discussed above and mitigated through our choice of mixed-effects regression models and their specific designs, we note the standard threats to statistical conclusion validity affecting linear regression models in general. To mitigate these, we take standard precautions. First, we removed as outliers the top 1% most extreme values. Second, we checked for collinearity among the predictors we use the variance inflation factor (VIF) [62]; all were below 3, *i.e.*, multicollinearity is not an issue [176]. Finally, we acknowledge that additional time may be spent as the users are asked to upload their edits, increasing the

amount of time necessary in the plugin setting. However the time spent for uploading is minimal as the plugin automatically helps the user to remove the auto-generated comments with only a press of a keyboard shortcut.

Results. Table 4.3 summarizes our default specification mixed-effects regressions for both response variables; the models with our second specification (de-meaned task experience) are equivalent, see Appendix 4.11.7. All models include controls for the amount of users' experience with the respective task categories as well as other random user and task effects. In all cases, the models fit the data reasonably well (ranging from $R_c^2 = 29\%$ for task correctness scores, to $R_c^2 = 64\%$ for task completion time), with most of the variance explained attributable to the two random effects (task and user)—there is significant user-to-user and task-to-task variability in all response variables.

Analyzing the models we make the following observations. First, looking at the completion time model (1), there is no statistically significant difference between the two conditions. Stated differently, we do not find sufficient evidence to conclude that users in the plugin condition complete their tasks with different speed on average than users in the control group, contrary to our expectation.

Second, and this time in line with our expectation, there is no statistically significant difference between the two conditions in task correctness scores (model (2)). That is, the code written by users in the plugin condition appears statistically indistinguishably as correct from the code written by users in the control group.

We investigate more differences between the code written by study participants in each of the two conditions in more detail in the next section.

4.6 RQ$_2$: Comparison of Generated vs Retrieved Code

In this section we focus on *how* study participants are interacting with the code generation and retrieval systems. Specifically, we dive deeper into both the inputs to and the outputs of the plugin, *i.e.*, we analyze the quality of the queries issued by study participants and of the code snippets produced in return, contrasting code generation to retrieval throughout. We analyze these data along three dimensions, detailed next.

4.6.1 For What Queries do Users Tend to Favor Generation vs Retrieval Answers

First, we investigate whether there are any discernible characteristics of the natural language queries (and therefore tasks) that associate with study participants tending to favor the code snippets returned by the code generation model over those returned by the code retrieval model.

Methodology. Using our instrumented environment, we collect all *successful* queries issued by the study participants, *i.e.*, those for which a code snippet from among the listed candidates was selected, and we record which of the two sources (generation or retrieval) the snippet came from. See Table 4.10 in Appendix 4.11.8 for the complete set of queries from our 31 participants, organized per task. We then build a binary logistic regression model with snippet source as outcome variable and bag-of-words features of the natural language input queries as predictors.

If this model is able to predict the source of the code snippet better than by chance, then we can conclude that there is some correlation between the type of input query and the users' preference for generated versus retrieved code snippets. Moreover, the word feature weights in the logistic regression model could shed some light on what features are the most representative of queries that were effectively answered using generation or retrieval. For our analysis, we manually review the top 20 (approx. 7%) contributing query features for each value of the outcome variable ("generation" vs "retrieval") and discuss patterns we observe qualitatively, after thematic analysis.

Specifically, for each query, we tokenize it, filter out English stop words, and compute a bag-of-words and bag-of-bigrams vector representation, with each element of the vector corresponding to the number of times a particular word or bigram (two-word sequence) occurred in the query. The number of distinct words in all queries is 302, and the number of distinct bigrams in all queries is 491, and thus the dimensionality of the query vector is 793.[29] We then estimate the model:

$$Pr(\text{chosen snippet is "generated"}) = \frac{\exp(\mathbf{X}\beta)}{1 + \exp(\mathbf{X}\beta)}, \tag{4.3}$$

where $\mathbf{X}$ here represents a k-dimensional bag-of-word vector representation of a given query, and β are the weights to be estimated. To this end, we randomly split all the collected query and candidate selection pairs into training (70% of the data) and held-out test (30%) sets. We then train the model using 5-fold cross-validation until it converges, and subsequently test it on the

[29] We also experimented with other features, *e.g.*, query length, query format compliance, etc., but did not notice a significant difference in prediction accuracy.

held-out set. We use 0.5 as a cutoff probability for our binary labels. In addition, we also build a trivial baseline model that always predicts "retrieval."

The baseline model is 55.6% accurate (among the successful queries in our sample there are slightly more code snippets retrieved rather than generated). Our main logistic regression model is 65.9% accurate, *i.e.*, the model was able to learn some patterns of differences between those queries that result in code generation results being chosen over code retrieval ones and vice versa.

Threats to Validity. One potentially confounding factor is that the plugin always displays code generation results first, before code retrieval. Ordering effects have been reported in other domains [299] and could also play a role here. Specifically, users who inspect query results linearly, top-down, would see the code generation results first and might select them more frequently than if the results were displayed in a different order. That is, we might infer that users prefer code generation to retrieval only because they see code generation results first, thus overestimating the users' preference for code generation versus retrieval.

Even though testing ordering effects experimentally was not practical with our study design, we could test a proxy with our log data—to what extent the code generation results overlap with the code retrieval ones. High overlap could indicate that code retrieval results might have been chosen instead of code generation ones, if presented earlier in the candidates list. Whenever study participants chose a snippet returned by the code generation model, we compared (as strings) the chosen snippet to all candidates returned by the code retrieval engine. Only 6 out of 173 such unique queries (~3.5%) also contained the exact chosen code generation snippet among the code retrieval results, therefore we conclude that this scenario is unlikely.[30]

Another potentially confounding factor is that an icon indicative of generation or retrieval is displayed next to each result in the plugin UI. This means that users know which model produced which candidate snippet and might choose a snippet because of that reason rather than because of the snippet's inherent usefulness. More research is needed to test these effects. We hypothesize that biases may occur in both directions. On the one hand, holding other variables like ordering fixed, users might prefer code generation results because of novelty effects. On the other hand, users might prefer code retrieval results because of general skepticism towards automatically generated code, as has been reported, *e.g.*, about automatically generated unit tests [89, 306].

[30]Note that this only considers exact substring matches. There may be additional instances of functionally equivalent code that is nonetheless not an exact match.

Regarding the analysis, we use an interpretable classifier (logistic regression) and follow standard practice for training and testing (cross-validation, held-out test set, *etc.*), therefore we do not expect extraordinary threats to validity related to this part of our methodology. However, we do note the typical threats to trustworthiness in qualitative research related to our thematic analysis of top ranking classifier features [256]. To mitigate these, we created a clear audit trail, describing and motivating methodological choices, and publishing the relevant data (queries, top ranking features after classification, *etc.*). Still, we note potential threats to transferability that may arise if different features or different classifiers are used for training, or a different number/fraction of top ranking features is analyzed qualitatively for themes.

Results. In Table 4.4, we show the top features that contributed to predicting each one of the two categories, and their corresponding weights. Inspecting the table we make two observations.

First, we observe that for code generation, the highest ranked features (most predictive tokens in the input queries) refer mostly to basic Python functionality, *e.g.*, "open, read csv, text file" (opening and reading a file), "sort, list, number, dictionary, keys" (related to basic data structures and operations in Python), "random number" (related to random number generation), "trim" (string operations), etc. For example, some stereotypical queries containing these tokens that result in the code generation snippets being chosen are "open a csv file `data.csv` and read the data", "get date and time in gmt", "list all text files in the data directory", etc.

In contrast, we observe that many queries that are more likely to succeed through code retrieval contain terms related to more complex functionality, some usually requiring a series of steps to fulfill. For example, "datetime" (regarding date and time operations), "cross validation, sklearn, column csv" (regarding machine learning and data analysis), "matplotlib" (data visualization), etc. are all among the top features for queries where users more often chose the code retrieval snippets.

In summary, it seems predictable (substantially more so than by random chance) whether natural language user queries to our NL2Code plugin are more likely to succeed through code generation vs code retrieval on average, given the contents (words) of the queries.

4.6.2 How Well-Specified Are the Queries

Search is a notoriously hard problem [137, 213], especially when users do not start knowing exactly what they are looking for, and therefore are not able to formulate clear, well-specified search queries. In this subsection we investigate the quality of the input natural language queries, and attempt to delineate it from the quality of the underlying code generation and retrieval

systems—either one or both may be responsible for failures to obtain desirable code snippets for a given task.

Anecdotally, we have observed that input queries to our NL2Code plugin are not always well-specified, even when the participants selected and inserted into their code one of the candidate snippets returned by the plugin for that query. A recurring issue seems to be that study participants sometimes input only a few keywords as their query (*e.g.*, "move file"), perhaps as they are used to interacting with general purpose search engines like Google, instead of more detailed queries as expected by our plugin. For example, study participants sometimes omit (despite our detailed instructions) variable names part of the intent but defined elsewhere in the program (*e.g.*, "save dataframe to csv" omits the DataFrame variable name). Similarly, they sometimes omit flags and arguments that need to be passed to a particular API method (*e.g.*, "load json from a file" omits the actual JSON filename).

Methodology. The key idea behind our investigation here is to replace the underlying code generation and retrieval systems with an oracle assumed to be perfect—a human expert Python programmer—and study how well the oracle could have produced the corresponding code snippet given a natural language input query. If the oracle could successfully produce a code snippet implementing the intent, then we deem the query "good enough", or well-specified; otherwise, we deem the query under-specified. The fraction of "good enough" queries to all queries can be considered as an upper bound on the success rate of a perfect code generation model.

Concretely, we randomly sampled 50 queries out of all successful queries issued during the user study (see Table 4.11 in Appendix 4.11.9 for the sample), and had the first author of this paper, an proficient programmer with 8 years of Python experience, attempt to generate code based on each of them. The oracle programmer considered two scenarios: (1) generating code given the input query as is, without additional context; (2) if the former attempt failed, generating code given the input query together with the snapshot of the source file the study participant was working in at the time the query was issued, for additional context.

For each query, we record three binary variables: two indicating whether each of the oracle's attempts succeeded, without and with additional context, respectively,[31] and the third indicating whether the code snippet actually chosen by the study participant for that query came from the code generation model or the code retrieval one; see Table 4.11 in Appendix 4.11.9.[32]

[31]The former implies the latter but not vice versa.

[32]Note that on the surface, when looking at the data in Table 4.11, the values of the former two binary variables (the oracle's determination) may not always seem intuitive given the query. For example, the oracle determined the

We then measure the correlation, across the 50 queries, between each of the two oracle success variables and the code snippet source variable, using the phi coefficient ϕ [65], a standard measure of association for two binary variables similar to the Pearson correlation coefficient in its interpretation. This way we can assess how close the code generation model is from a human oracle (the *good enough as is* scenario), and whether contextual information from the source code the developer is currently working on might be worth incorporating into code generation models in the future (the *good enough with context* scenario); note that the code generation model we used in this study [364, 385] does not consider such contextual information.

Threats to Validity. We follow standard practice for the statistical analysis in this section, therefore we do not anticipate notable threats to statistical conclusion validity. Due to the limitations of our telemetry system, we did not record unsuccessful queries (i.e. queries that the user entered but no candidate is selected). As a result, queries that favor neither generation nor retrieval cannot be compared. However, we acknowledge three other notable threats to validity. First, we used only one expert programmer as oracle, which may introduce a threat to construct validity given the level of subjectivity in determining which queries are "good enough". To mitigate this, we discussed among the research team, whenever applicable, queries for which the expert programmer was not highly confident in the determination. Second, our random sample of 50 queries manually reviewed by the expert programmer is only representative of the population of 397 queries with 95% confidence and 13% margin of error, which may introduce a threat to internal validity. However, the relatively small sample size was necessary for practical reasons, given the high level of manual effort involved in the review. Finally, we note a potential threat to construct validity around the binary variable capturing the source (generation or retrieval) of the candidate code snippets selected by the study participants. There is an implicit assumption here that study participants know what the right answer (code snippet) should be given a natural language query, and are able to recognize it among the candidates provided by the NL2Code plugin; therefore, we assume that the snippet source variable captures actual quality differences between code snippets produced by the generation and retrieval

query "pandas to csv" to be *not good enough*, even with context, while the query "pandas output csv", seemingly equivalent, was found to be *good enough with context*. In both cases, the intent appears to be exporting a pandas dataframe (a popular data science Python library) as a csv file. However, in the first example the snapshot of the source file the study participant was working in at the time of the query did not yet include *any* such dataframe objects; the user appears to have issued the query ahead of setting up the rest of the context. A context-aware code generation model would also not be able to extract any additional information in this case, similarly to the human oracle.

models, respectively. However, this may not be the case. To test this, we reviewed all the candidate snippets returned by the plugin for the first 6 among the 50 queries analyzed. Across the $6 \cdot 2$ models (generation/retrieval)$\cdot 7$ candidates per model = 84 candidate snippets, we only discovered one case where the study participant could have arguably chosen a more relevant snippet. Therefore, we expect the incidence of violations of this assumption to be rare enough to not materially affect our results.

Results. Table 4.5 shows contingency tables for each of the two oracle comparison scenarios. Note that the "good enough with context" category includes all queries that are "good enough as is", by construction. Inspecting the results in the table, we make the following observations.

First, the natural language queries analyzed are more often than not insufficiently well-specified for even the human expert to be able to write code implementing those intents; only 20 out of 50 queries (40%) are deemed "good enough as is" by the oracle. Representative examples of failures from Table 4.11 are the queries consisting of a few keywords (*e.g.*, "csv writer", "defaultdict") rather than queries containing sufficient details about the user's intent (*e.g.*, "remove first column from csv file"). Considering the source file the user was editing at query time helps, with 34 (68%) of the queries now being deemed "good enough with context" by the oracle.

Second, there is moderately high and statistically significant association between the success of the code generation model (*i.e.*, the study participant choosing one of those candidate code snippets) and the quality of queries in both scenarios: $\phi = 0.37$ ($p = 0.008$) for already well-specified queries and $\phi = 0.45$ ($p = 0.001$) for queries that become informative enough given additional context. This suggests that input query quality can have a big impact on the performance of the code generation model, and that incorporating additional contextual information may help.

Analyzing the failure rate of the code generation model (generation = False), we observe that it is relatively high in general (31 out of 50 queries, or 62%). However, most of these cases are in response to under-specified queries (23 out of the 31 failures; 74%), for which even the human oracle failed to generate the corresponding code. Still, there are 8 (26%) failure cases where the human expert could directly implement the natural language intent without additional context: "date now", "for loop on range 100", "generate random letters", "get now one week from now", "get time and date", "open "data.csv" file", "how to remove an item from a list using the index", and "plt create 3 subplots". All but the last one seem to refer to basic Python functionality. These queries are targets where further improved code generation techniques could improve the utility

of the plugin.

Interestingly, we also observe a non-trivial number of under-specified queries (7 out of 30; 23%) for which the code generation model succeeded despite the human oracle failing: “call `pick_with_replacement`”, “copy a file to dist”, “pandas round value”, “pandas to csv”, “rename column pandas”, “plt ax legend”, and “scatter”.

4.6.3 How Much Are the Code Snippets Edited After Plugin Use

Choosing (and inserting into the IDE source file) one of the candidate code snippets returned by the NL2Code plugin indicates that the code snippet was generally useful. However, while useful, the code snippet may still be far from an ideal solution to the user’s query. To get a sense of how appropriate the accepted code snippets are given the user intent, we compare the distributions of snippet lengths before (*i.e.*, as returned by the plugin) and after potential edits in the IDE.

Methodology. When inserting a code snippet a user selected from among the plugin-returned candidates, we also insert special code comments in the source file around the snippet, to mark the start and end of the code fragment corresponding to that particular intent (as shown in Figure 4.3). Study participants are instructed to use a certain key combination when they are done editing that code fragment to remove the delimiters and submit the edited version of the code fragment back to our server. Our analysis in this section compares the length of code snippets and types of tokens present between these two versions.

Specifically, we first tokenize and tag each version of a code snippet using a Python tokenizer, and then compare the pairs of distributions of lengths before and after edits for code snippets originating from each of the two underlying models, generation and retrieval, using the non-parametric Wilcoxon signed-rank test; in addition, as a measure of effect size we compute the median difference between members of the two groups, *i.e.*, the Hodges–Lehman estimator [134]. We also compute and report on the Levenshtein edit distance between the two versions, in terms of number of tokens. Figure 4.6 visualizes these different distributions.

Threats to Validity. We note two potential threats to construct and external validity related to the analysis in this section. First, we have no way of enforcing that study participants contain their code edits related to a particular intent to the section of the source file specially delimited by code comments for this purpose. One may include unrelated edits in the same code region, or make related edits outside of the designated region. Therefore, our measurement of *snippet length post edits* may not accurately reflect the construct of snippet length as related to a particular intent. To mitigate this, we gave clear instructions to participants at the beginning of the study and

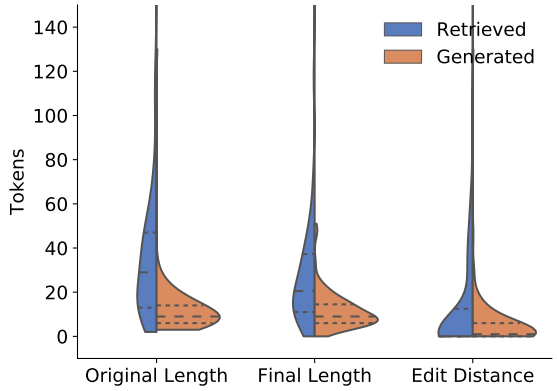


Figure 4.6: Split violin plots comparing the length (in tokens) of the code snippets chosen by the study participants across all successful queries, before and after potential edits in the IDE. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

manually reviewed a small sample of the edited versions of a snippet, not discovering any obvious noise. Second, not all study participants followed our instructions every time they used the plugin, and submitted their final (edited or not) version of the snippet back to our server. Only 303 out of the 397 successful queries recorded (76.3%) had final code snippets uploaded back to our server. Since this was not a random sample, our findings on this sample may not generalize to the entire population of 397 successful queries. To assess the severity of this potential threat, we compared the distributions of plugin-returned code snippet lengths between all successful queries and just the 303 queries where study participants uploaded their edits onto our server; for both generated (Wilcoxon $p = 0.54$) and retrieved ($p = 0.93$) code snippets, we found the respective two distributions statistically indistinguishable, therefore we expect this to not be a sizable threat to validity.

Results. Comparing the two distributions of token lengths for acceptable code snippets from the code generation model before and after edits, we do not find any statistically significant differences in their mean ranks ($p = 0.345$). The mean edit distance between the two versions of these snippets is 5.2 tokens (min 0, max 130, median 1).

In contrast, comparing the two distributions of token lengths for acceptable code snippets from the code retrieval engine before and after edits, we find a statistically significant difference in their mean ranks ($p = 1.195 \times 10^{-7}$). The Hodges–Lehman median difference between the edited and unedited versions of these snippets is 18 tokens, with a 95% confidence interval from 11 to 23 tokens. The edit distance metric paints a similar picture—acceptable code snippets from the code retrieval engine, before and after edits, are at a mean edit distance of 13.2 tokens from

each other (min 0, max 182, median 0).

We also note that code retrieval snippets tend to be longer than code generation ones both before ($p < 2.2 \times 10^{-16}$; median difference 18 tokens, with a 95% confidence interval from 14 to Infinity) and after edits ($p = 2.657 \times 10^{-14}$; median difference 10 tokens, with a 95% confidence interval from 7 to Infinity). This may help explain why the retrieved snippets require more edits to correct the code to better suit the current programming code context, compared to the generated snippets.

Diving deeper into the edits to the plugin-supplied version of the different snippets, we compute the frequency distribution of tokens in both versions (plugin and final), normalized based on total token count in each corpus. Table 4.6 highlights the tokens with the greatest increases and decreases in relative frequency during editing. We observe that study participants seem to add common keywords such as "in, for, if, with", built-in names and functions such as "key, print", and common variable names such as "line, filename" to the generated/retrieved candidates. Stated differently, in these cases the code snippets seem to miss substantive parts and relevant functionality, which also may be partly due to the lack of specificity described in the previous section.

In contrast, study participants seem to delete number and string literals from the code snippets. This may be explained by the fact that the tool used retrieved code snippets as they appeared on Stack Oveflow, and thus many retrieved code snippets contain additional boilerplate code required for initialization or setup, and hard-coded example inputs and outputs. We also observe some commonly used variable names like "df, plt" that get deleted, suggesting that variable replacement is one of the common operations when reusing the code snippets. An interesting observation here is that "In" and "Out" are getting deleted frequently. We find that it's mostly due to some of the code snippets retrieved from Stack Overflow being in the format of IPython REPL, which uses "In" and "Out" to separate the Python source code and execution outputs. When integrating these snippets, the users will have to remove this superfluous text. Figure 4.7 shows a representative example of such user edits after selecting a candidate snippet, which involves deleting IPython REPL contents, variable replacement and addition, as well as literal replacements.

Furthermore, following the previous observations on actual tokens, we are interested in how the frequency of different *types* of tokens changes before and after users edit the plugin-returned code snippets. We use the `tokenize`[33] Python 3 library to parse and tag the code snippets, and

[33] https://docs.python.org/3/library/tokenize.html

Unedited

```
In [479]: df
Out[479]:
    ID  birthyear   weight
0   619040  1962    0.123123
1   600161  1963    0.981742
2   25602033    1963    1.312312
3   624870  1987    0.942120

In [480]: df["weight"].mean()
Out[480]: 0.83982437500000007
```

Edited

```
car_prices = car_prices["price"].mean()
```

Figure 4.7: Representative example of user edits to a code snippet retrieved from Stack Overflow.

compare the frequency changes by token type, similar to the previous analysis.[34] The results are shown in Table 4.7. We find that users add new NAME (identifiers, keywords) tokens the most, with the frequency of STRING (string literal) tokens slightly increased, and COMMENT (comment strings) tokens staying roughly the same after the edits. NUMBER (numeric literal) tokens are deleted the most, in line with the observation above, again suggesting that many plugin-returned snippets are not tailored to specific identifiers and parameters that the user desires. Interestingly, we also see a slight decrease in frequency of NEWLINE tokens, representing a decrease in the number of logical lines of Python code after edits. This suggests that the plugin-returned code snippets are not concise enough in some cases.

4.7 RQ$_3$: User Perceptions of the NL2Code Plugin

Our last research question gauges how study participants perceived working with the NL2Code plugin, their pain points, and their suggestions for improvement.

Methodology. As part of our post-test survey, we asked the participants open-ended questions about what worked well when using the plugin and, separately, what they think should be improved. In addition, we asked participants to rate their overall experience using the plugin on a Likert scale, ranging from 1 (very bad) to 5 (very good). We then qualitatively coded the answers to open-ended questions to identify themes in the responses for the 31 who completed all their assigned tasks.

[34] 3 of the retrieved snippets cannot be parsed and thus are omitted. See full explanation of different token types at https://www.asmeurer.com/brown-water-python/tokens.html. We also left out some uninteresting token types, such as ENCODING, ENDMARKER, NL.

Threats to Validity. We acknowledge usual threats to trustworthiness and transferability from qualitatively analyzing a relatively small set of open-ended survey data [256], as also discussed above. In particular, we note that only one researcher was involved in coding. To mitigate these threats, we release all verbatim survey responses as part of our replication package.

Results. Overall, study participants report having a neutral (15/31; 48.4%) or at least somewhat positive (15/31; 48.4%) experience using the NL2Code plugin, with only one participant rating their experience as somewhat negative.

Among the aspects the participants report as **positive**, we distill two main themes:

The plugin helps find code snippets the developer is aware of but cannot fully remember (P1, P2, P8, P10, P11, P19, P20, P21, P22, P30, P31) These tend to be small commands or less familiar API calls and API usage patterns, that users have seen before. Two participants summarize this well:

> "On a few occasions, the plugin very conveniently gave me the snippet of code I was looking for, [which] was "on the tip of my tongue"." (P10)

> "Sometimes I just cannot remember the exact code, but I remember the shape. I could select the correct one easily." (P2)

Respondents expressed appreciation for both the generation and retrieval results, and there was little expression of preference for one method over the other, *e.g.*:

> "Even just having the snippets mined from Stack Overflow visible in the IDE was a good memory refresher / source of ideas" (P10)

> "It was somewhat convenient to not have to switch tabs to Google things, ..., based on my memory, that most of the suggestions I got were from the internet anyway." (P5)

> "It has all resources needed at one place." (P6)

Using an in-IDE plugin is less disruptive than using a web browser (P1, P4, P5, P6, P7, P10, P18, P20, P24, P27) Many of our respondents who were positive about the plugin reiterate expected context-switching benefits of not leaving the IDE while programming, *e.g.*:

> "I like that the plugin stops me having to go and search online for solutions. [...] It can be very easy to get distracted when searching for solutions online." (P20)

> "Compared with manual search, this is faster and less disruptive." (P1)

Participants also describe many aspects of the plugin that **could be improved**.

The quality of code generation and retrieval results could be higher (P3, P4, P5, P7, P9, P13, P14, P23, P27, P29, P31) Respondents mentioned that it was "rare" (P7) when they could directly use code from plugin, without modifications. In some cases, results from the plugin were "not related to the search" (P14), and users "didn't find what [they were] searching for" (P31). As one respondent humbly summarized it:

> "The model needs some improvements." (P4)

The insufficient quality of the plugin's results was especially felt as the tasks became more complex and involved APIs with complex usage patterns. One participant summarized this well:

> "For easy tasks, like walking through a directory in the filesystem, the plugin saves me time because what I did previously was to go to Stack Overflow and copy the code. But for difficult tasks like data processing or ML, the plugin is not helpful. Most snippets are not useful and I have to go to the website of sklearn to read the full doc to understand what I should do." (P3)

A particular related pain point is that the snippets from the code retrieval engine often contain spurious elements (as also noted above). In one participant's words:

> "When inserting the code into my program, I would like to **not** copy the input/output examples, and I can't imagine ever wanting those in the program itself." (P5)

Users could benefit from additional context (P3, P5, P8, P18, P19, P20, P24, P26, P27) Some respondents mention that it would be useful to include additional (links to) explanations and documentation alongside the returned code snippets so that the user could understand what the snippet is supposed to do, or even "which of the suggestions is the correct one when you are not familiar with a module" (P11). In two participants' words:

> "It would be nice if the examples from the internet could contain the relevant context of the discussion (e.g., things to consider when using this suggestion), as well as the input/output examples." (P5)

> "I hope the generated code snippet can have more comments or usage [examples]. Otherwise I still need to search the web to understand what it is." (P3)

A closely related theme is that *using the plugin assumes one has a "good background understanding of the underlying principles/modules/frameworks"* (P11), and they primarily need help with "look[ing] up little syntax bits that you have forgotten" (P11). (P1, P11, P16, P25) One participant was especially critical:

> "For more complex problems, I think the plugin does not help at all, because the programmer needs to know the theoretical background." (P16)

The plugin could benefit from additional context (P4, P9, P10, P17, P30) Some participants suggest that the plugin could be "smarter" if it becomes more aware of the local context in the developer's IDE, *e.g.*:

> "Sometimes I want to generate an expression to be inserted somewhere, to be assigned to a variable, or to match the indentation level, without having to tell the plugin this explicitly. I didn't feel like the plugin was aware of context." (P10)

Participants also comment on how *the plugin's query syntax takes some getting used to* (P2, P12, P15), referring in particular to the way the code generation model expects queries to include variables, while the web search code retrieval engine allows users to only use keywords. For example:

> "[It became] useful to me towards the end when I got the hang of it and could formulate questions in the correct way (which I feel is somewhat of a skill in itself)" (P15)

> "It is not very natural for me to 'instantiate' my questions, I mostly like to search [using] keywords or just a description of what I want to achieve." (P2)

Querying the plugin could be interactive (P11, P20, P30) Finally, some participants suggest to make querying interactive, dialogue-based, rather than unidirectional. This could with refining queries until they are sufficiently well-specified, or to decompose complex functionality into smaller steps, *e.g.*:

> "A chatbot [...] could identify the rough area in which the user needs assistance, [and] could help narrow it down further, helping to pinpoint an exact solution." (P20)

4.8 Discussion and Implications

Recent years have seen much progress from machine learning and software engineering researchers developing techniques to better assist programmers in their coding tasks, that exploit the advancements in (deep) learning technology and the availability of very large amounts of data from Big Code repositories like GitHub and Stack Overflow. A particularly promising research direction in this space has been that addressing the decades-old problem of "natural language

programming" [74], *i.e.*, having people instruct machines in the same (natural) language they communicate in with each other, which can be useful in many scenarios, as discussed in the Introduction. However, while excited about this research direction and actively contributing to it ourselves, we are also questioning whether the most impact from such work can be had by focusing primarily on making technological advancements (*e.g.*, as we write this, a one-trillion parameter language model has just been announced [82], as only the most current development in a very rapidly evolving field) without also carefully considering how such proposed solutions can fit within the software development workflow, through human-centered research.

In this spirit, we have presented the results of a controlled experiment with 31 participants with diverse background and programming expertise, observed while completing a range of Python programming tasks with and without the help of a NL2Code IDE plugin. The plugin allows users to enter descriptions of intent in natural language, and have corresponding code snippets, ideally implementing said intent, automatically returned. We designed the plugin with two research goals in mind. First, we sought to evaluate, to our knowledge for the first time using a human-centered approach, the performance of some NL2Code *generation* model with state-of-the-art performance on a benchmark dataset, but unknown performance "in the wild". Second, we sought to contrast the performance and user experience interacting with such a relatively sophisticated model to those of a relatively basic NL2Code *retrieval* engine, that "merely" retrieves existing code snippets from Stack Overflow given natural language search queries. This way, we could estimate not only how far we are from not having to write *any* code while programming, but also how far we have come on this problem given the many recent advancements in learning and availability of datasets.

Main Results. Overall, our results are mixed. First, after careful statistical analysis in **RQ$_1$**, comparing tasks completed with and without using the NL2Code plugin (and either of its underlying code generation or retrieval systems), we found no statistically significant differences in task completion times or task correctness scores.

The results for **code metrics** (SLOC and CC) can be seen as mixed. One the one hand, the code containing automatically generated or retrieved fragments is not, on average, any more complex or any less maintainable than the code written manually, insofar as the CC and SLOC metrics can distinguish. One the other hand, one could have expected the opposite result, *i.e.*, that since NL2Code tools are typically trained on idiomatic code, using them should lead to "better", more idiomatic code overall, which might suggest lower SLOC and CC values, on average.

Among the possible explanations for why we don't find supporting evidence for the "better code" hypothesis, two stand out: (i) the two metrics are only crude approximations of the complex, multifaceted concept of code quality; and (ii) even when writing code "manually", developers still consult the Web and Stack Overflow (*i.e.*, the same resources that these NL2Code tools were trained on) and copy-paste code therein. To better understand the interaction between using the plugin and using a traditional Web browser, we used the event logs from our instrumented environment and compared the distributions of in-browser Web searches between tasks where the 31 study participants used the NL2Code plugin (median 3, mean 5, min 0, max 35 searches per user per task) and tasks where they did not (median 4, mean 7, min 0, max 48). A mixed-effects regression model similar to the ones in Section 4.5, controlling for individual self-reported experience and with random effects for `user` and `task`, reveals a statistically significant effect of using the plugin on the number of in-browser Web searches: on average, using the plugin is associated with 2.8 *fewer* in-browser Web searches; however, this effect is smaller than the standard deviation of the random `user` intercept (~4 in-browser Web searches). We conclude that developers still search the Web when using the plugin, even if slightly less than when not using the plugin.

Using a similar argument, the result for **task correctness scores** can be seen as mixed. Code containing automatically generated or retrieved snippets is not, on average, any less appropriate for a given task as per our rubric than code written manually. However, using the NL2Code plugin doesn't seem to help our study participants significantly improve their scores either, despite there being room for improvement. Even though across our sample the median score per task was 7 out of 10 when using the plugin and 6 when not using the plugin, the multivariate regression analysis did not find the difference to be statistically significant.

The result for **task completion times** can be seen as negative and, thus, is perhaps the most surprising of our results: on average, study participants do not complete their tasks statistically significantly faster when using the NL2Code plugin compared to when they are not using it. There are several possible explanations for this negative result. First, we acknowledge fundamental limitations of our study design, which we hope future researchers can improve on. In particular, our tasks, despite their diversity and, we believe, representativeness of real-world Python use, may not lend themselves sufficiently well to NL2Code queries and, therefore, study participants may not have sufficient opportunities to use, and benefit from, the plugin. Moreover, our study population (31 participants) may not be large enough for us to detect effects with small sizes, should they exist.

However, even with these limitations, considering also our results for **RQ$_2$** and **RQ$_3$** we

argue that another explanation is plausible: *our NL2Code plugin and its main underlying code generation technology, despite state-of-the-art (BLEU-score) performance on a benchmark dataset, is not developed enough to be markedly useful in practice just yet.* Our telemetry data (**RQ**$_2$) shows not only that study participants still carry out in-browser Web searches even though the NL2Code plugin was available, as discussed above, but also that the code snippets returned by the plugin, when used, undergo edits after insertion in the IDE, suggesting insufficient quality to begin with. Our qualitative survey data (**RQ**$_3$) paints a similar picture of overall insufficient quality of the NL2Code results.

Implications. While our study suggests that state-of-the-art learning-based natural language to code generation technology is ways away from being useful in practice, our results should be interpreted more optimistically.

First, we argue that **the problem is worth working on**. In contemporary software development, which involves countless and constantly changing programming languages and APIs, natural language can be a useful medium to turn ideas into code, even for experienced programmers. A large fraction of our study participants commended NL2Code developer assistants for helping them remember the precise syntax or sequence of API calls and their arguments, required to implement some particular piece of functionality. When integrated into the development workflow, *e.g.*, through an IDE plugin, such systems can help developers focus by reducing the need for context switching, further improving their productivity. Our quantitative task performance results for the current version of this NL2Code plugin, while negative, do not imply that future, better performing such systems will also not be markedly useful in practice; the qualitative data from our our study participants already suggests otherwise, as does quantitative data from prior research on the usefulness of in-IDE code search plugins [274].

Second, we argue that **this particular style of code *generation* is worth working on**. Our analysis of input queries and resulting code snippets for **RQ**$_2$ shows that the code generation model produces fundamentally different results than the (simple) code retrieval engine we used for comparison, and that study participants choose snippets returned by the code generation model almost as frequently as they do snippets from the code retrieval engine. In turn, this suggests that, at least within the scope of the current study, one type of model cannot be used as a substitute for the other. As discussed above, the code generation model does almost always produce different results than the code retrieval model. However, it was unclear from that analysis whether the generated code snippets reflect some fundamentally higher level of sophistication inherent to the code generation model, or whether the code retrieval engine we

used for comparison is simply too naive.

To further test this, we performed an additional analysis. Specifically, we looked up the chosen code generation snippets in the manually-labeled Stack Overflow dataset used for training the code generation model, to assess whether the model is simply memorizing the training inputs. Only 13 out of the 173 unique queries (~7.5%) had as the chosen code fragment snippets found verbatim in the model's training dataset. Therefore, the evidence so far suggests that the code generation model does add some level of sophistication, and customization of results to the developers' intent (*e.g.*, composing function calls), compared to what *any* code retrieval engine could.

Third, we provide the following **concrete future work recommendations** for researchers and toolsmiths in this area, informed by our results:

- *Combine code generation with code retrieval.* Our results suggest that some queries may be better answered through code retrieval techniques, and others through code generation. We recommend that future research continue to explore these types of approaches jointly, *e.g.*, using hybrid models [115, 118] that may be able to combine the best of both worlds.
- *Consider the user's local context as part of the input.* Our oracle comparison revealed that users' natural language queries can often be disambiguated by considering the local context provided by the source files they were working in at the time, which in turn could lead to better performance of the code generation model. There is already convincing evidence from prior work that considering a user's local context provides unique information about what code they might type next [343]. In addition, some work on code retrieval has also considered how to incorporate context to improve retrieval results [50]; this may be similarly incorporated.
- *Consider the user's local context as part of the output.* Considering where in their local IDE users are when invoking an NL2Code assistant can also help with localizing the returned code snippets for that context. Some transformations are relatively simple, *e.g.*, pretty printing and indentation. Other transformations may require more advanced program analysis but are still well within reach of current technology, *e.g.*, renaming variables used in the returned snippet to match the local context (the Bing Developer Assistant code retrieval engine [359] already does this), or applying coding conventions [7].
- *Provide more context for each returned snippet.* Our study shows that NL2Code generation or retrieval systems can be useful when users already know what the right answer is, but they

need help retrieving it. At the same time, many of our study participants reported lacking sufficient background knowledge, be it domain-specific or API-specific, to recognize when a plugin-returned code snippet is the right one given their query, or what the snippet does in detail. Future research should consider incorporating more context and documentation together with the plugin's results, that allows users to better understand the code, *e.g.*, links to Stack Overflow, official documentation pages, explanations of domain-specific concepts, other API usage examples. One example of this is the work of Moreno et al. [239], which retrieves usage examples that show how to use a specific method.

- *Provide a unified and intuitive query syntax.* We observed that users are not always formulating queries in the way that we would expect, perhaps because they are used to traditional search engines that are more robust to noisy inputs and designed for keyword-based search. The NL2Code generation model we experimented with in this study was trained on natural language queries that are not only complete English sentences, but also include references to variables or literals involved with an intent, specially delimited by dedicated syntax (grave accents). As our respondents commented in the post-test survey, getting used to formulating queries this way takes some practice. Future research should consider not only what is the most natural way for users to describe their intent using natural language, but also how to provide a unified query syntax for both code generation and code retrieval, to minimize confusion. Robust semantic parsing techniques [19, 287] may also help with interpreting ill-specified user queries.

- *Provide dialogue-based query capability.* Dialogue-based querying could allow users to refine their natural language intents until they are sufficiently precise for the underlying models to confidently provide some results. Future systems may reference work on query reformulation in information retrieval, where the user queries are refined to improve retrieval results both for standard information retrieval [18] and code retrieval [113, 130]. In addition, in the NLP community there have been notable advancements recently in interactive semantic parsing [157, 378], *i.e.*, soliciting user input when dealing with missing information or ambiguity while processing the initial natural language query, which could be of use as well.

- *Consider new paradigms of evaluation for code generation and retrieval systems.* Usage log data, such as the ones we collected here, is arguably very informative and useful for researchers looking to evaluate NL2Code systems. However, compared to automated

metrics such as BLEU, such data is much less readily available. We argue that such data is worth collecting even if only in small quantities. For example, with little but high quality data, one could still train a reranker [386] to try to select the outputs that a human user selected; if the predictive power exceeds that of BLEU alone, then the trained reranker could be used to automatically evaluate the quality of the generated or retrieved code more realistically than by using BLEU.

4.9 Related Work

Finally, we more extensively discuss how this work fits in the landscape of the many other related works in the area.

4.9.1 NL2Code Generation

While we took a particular approach to code generation, there are a wide variety of other options. Researchers have proposed that natural language dialogue could be a new form of human-computer interaction since nearly the advent of modern computers [74, 97, 124, 234]. The bulk of prior work either targeted domain-specific languages (DSLs), or focused on task-specific code generation for general-purpose languages, where more progress could be made given the relatively constrained vocabulary and output code space. Examples include generating formatted input file parsers [191]; structured, idiomatic sequences of API calls [290]; regular expressions [181, 226, 266]; string manipulation DSL programs [297]; card implementations for trading card games [212]; and solutions to the simplest of programming competition-style problems [25].

With the recent boom of neural networks and deep learning in natural language processing, generating arbitrary code in a general-purpose language [383, 385] are becoming more feasible. Some have been trained on both official API documentation and Stack Overflow questions and answers [364]. There are also similar systems[35] able to generate class member functions given natural language descriptions of intent and the programmatic context provided by the rest of the class [139], and to generate the API call sequence in a Jupyter Notebook code cell given the

[35]This is, of course, among the many other use cases for neural network models of code and natural language such as code summarization [138, 376], or embedding models that represent programming languages together with natural languages [84]. Allamanis et al. [10] provide a comprehensive survey of the use cases of machine learning models in this area.

natural language and code history up to that particular cell [1].

4.9.2 NL2Code Retrieval

Code retrieval has similarly seen a wide variety of approaches. The simplest way to perform retrieval is to start with existing information retrieval models designed for natural language search, and adapt them specifically for the source code domain through query reformulation or other methods [113, 130, 161, 219, 348, 359]. Other research works utilize deep learning models [8, 106, 137, 138] to train a relevance model between natural language queries and corresponding code snippets. It is also possible to exploit code annotations to generate additional information to help improve code retrieval performance [379] or extracted abstract programming patterns and associated natural language keywords for more content-based code search [161]. Many of the models achieve good performance on human annotated relevance benchmark datasets between natural language and code snippets. Practically, however, many developers simply rely on generic natural-language search engines like Google to find appropriate code snippets by first locating pages that contain code snippets through natural language queries [309] on programming QA websites like Stack Overflow.

4.9.3 Evaluation of NL2Code Methods

In order to evaluate whether NL2Code methods are succeeding, the most common way is to create a "reference" program that indeed implements the desired functionality, and measure the similarity of the generated program to this reference program. Because deciding whether two programs are equivalent is, in the general case, undecidable [298], alternative means are necessary. For code generation in limited domains, this is often done by creating a small number of input-output examples and making sure that the generated program returns the same values as the reference program over these tests [33, 180, 355, 375, 394, 395, 397, 402, 404]. However, when scaling to broader domains, creating a thorough and comprehensive suite of test cases over programs that have a wide variety of assumptions about the input and output data formats is not trivial.

As a result, much research work on code generation and retrieval take a different tack. Specifically, many code generation methods [1, 139, 364, 383] aim to directly compare generated code snippets against ground truth snippets, using token sequence comparison metrics borrowed from machine translation tasks, such as BLEU score [265]. However, many code snippets are equivalent in functionality but differ quite largely in terms of token sequences, or differ only

slightly in token sequence but greatly in functionality, and thus BLEU is an imperfect metric of correctness of a source code snippet [341].

Code retrieval, on the other hand, is the task of retrieving relevant code given a natural language query, that is related to other information retrieval tasks. Since code retrieval is often used to search for vague concepts and ideas, human-annotated relevance annotations are needed for evaluation. The common methods used in research work [106, 137, 376] compare the retrieved code snippet candidates given a natural language query, with a human annotated list of code snippet relevance, using common automatic information retrieval metrics like NDCG, MRR, etc. [225] The drawback of this evaluation method is that the cost of retrieval relevance annotation is high, and often requires experts in the specific area. Also, since the candidate lists are usually long, only a few unique natural language queries could be annotated. For example, one of the most recent large scale code search challenge CodeSearchNet [137] contains only 99 unique natural language queries, along with their corresponding code snippet relevance expert annotations, leading to smaller coverage of real world development scenarios in evaluation.

Regardless of the automatic metrics above, in the end our final goal is to help developers in their task of writing code. This paper fills the gap of the fundamental question of whether these methods will be useful within the developer workflow.

4.9.4 In-IDE Plugins

Similarly, many research works on deploying plugins inside IDEs to help developers have been performed. Both Ponzanelli et al. [273] and Ponzanelli et al. [274] focus on reducing context switching in IDE by incorporating Stack Overflow, by using the context in the IDE to automatically retrieve pertinent discussions from Stack Overflow. Subramanian et al. [334] proposes a plugin to enhance traditional API documentation with up-to-date source code examples. Rahman and Roy [291] and Liu et al. [216] designs the plugin to help developers find solutions on the Internet to program exceptions and errors. Following the similar route, Brandt et al. [41] studies opportunistic programming where programmers leverage online resources with a range of intentions, including the assistance that could be accessed from inside the IDE.

Besides plugin developed to reduce context-switching to other resources in developer workflows, Amann et al. [13] focus on collecting data of various developer activities from inside the IDE that fuel empirical research on the area [277].

This paper proposes an in-IDE plugin that incorporates code generation in addition to code retrieval to test the user experience in the real development workflow. In the meantime it also

collects fine-grained user activities interacting with the plugin as well as editing the code snippet candidates, to provide public data for future work.

4.9.5 End-User Development

The direction of exploring using natural language intents to generate code snippets is closely related to end-user development [207], which allows end-users (people who are not professional software developers) to program computers. Cypher et al. [66] is among the first work that enables end-user to program by demonstration.

Traditionally, programming has been performed by software developers who write code directly in programming languages for the majority of functionality they wish to implement. However, acquiring the requisite knowledge to perform this task requires time-consuming training and practice, and even for skilled programmers, writing programs requires a great amount of time and effort. To this end, there have been many recent developments on no-code or low-code software development platforms that allow both programmers and non-programmers to develop in modalities of interaction other than code [310]. Some examples include visual programming languages such as Scratch [224] that offers a building-block style graphical user interface to implement logic. In specific domains such as user interface design and prototyping, recent advances in deep learning models also enable developers to sketch the user interface visually and then automatically generates user interface code with the sketch [30], or using existing screenshots [252].

Besides visual no-code or low-code programming interfaces, there has also been much progress on program synthesis [27, 83, 86, 328], which uses input-output examples, logic sketches, etc. to automatically generate functions, with some recent advances that use machine learning models [25, 57, 79, 321]. Some work also generate programs from easier-to-write pseudo-code [180, 402].

There are other work in the area. Barman et al. [26], Chasins et al. [52, 53] make web automation accessible to non-coders through programming by demonstration, while Li et al. [199, 200, 201] automates mobile applications with multimodal inputs including demonstration and natural language intents. Head et al. [122] combines teacher expertise with data-driven program synthesis techniques to learn bug-fixing code transformations in classroom scenarios. Head et al. [123] helps users extract executable, simplified code from existing code. Ko and Myers [173, 174] provides a debugging interface for asking questions about program behavior. Myers and Stylos [242] discusses API designers should consider usability as a step towards enabling

end-user programming. Kery and Myers [162], Kery et al. [163] enable data scientists to explore data easily with exploratory programming. Our paper's plugin of using both state-of-the-art code generation and code retrieval to provide more natural programming experience to developers, with the potential future of enabling end-user programming, is related to Myers et al. [243] that envisions natural language programming.

4.9.6 Code Completion

Many developers use Integrated Development Environments (IDEs) as a convenient solution to help with many aspects during development. Most importantly, many developers actively rely on intelligent code-completion aid like IntelliSense[36] for Visual Studio [14, 277] to help developers learn more about the code, keep track of the parameters, and add calls to properties and methods with only a few keystrokes. Many of intelligent code-completion tools also consider the current code context where the developer is editing. With the recent advances in machine learning and deep learning, example tools like IntelliCode[37] for Visual Studio, Codota[38] and TabNine[39] present AI-assisted code-suggestion and code-completion based on the current source code context, learned from abundant amounts of projects over the Internet. The scope of our paper is to investigate generating or retrieving code using natural language queries, rather than based on the context of the current source code.

4.10 Conclusion

In this paper, we performed an extensive user study of in-IDE code generation and retrieval, developing an experimental harness and framework for analysis. This demonstrated challenges and limitations in the current state of both code generation and code retrieval; results were mixed with regards to the impact on the developer workflow, including time efficiency, code correctness and code quality. However, there was also promise: developers subjectively enjoyed the experience of using in-IDE developer management tools, and provided several concrete areas for improvement. We believe that these results will spur future, targeted development in productive directions for code generation and retrieval models.

[36] https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense
[37] https://visualstudio.microsoft.com/services/intellicode
[38] https://www.codota.com/
[39] https://www.tabnine.com/

4.11 Appendix

4.11.1 User Study Environment Design

To control the user study's development environment for different users as much as possible, and to enable data collection and activity recording outside the IDE (e.g. web browsing activity during the development), we design a complete virtual machine-based environment for users to access remotely and perform the user study on. We build the virtual machine based on a lot of open source software, including Ubuntu 18.04 operating system[40] with XFCE 4.1 desktop environment.[41] The virtual machine software is VirtualBox 6.1.10,[42] and we use Vagrant software[43] for automatic virtual machine provisioning.

Inside the Linux virtual machine, we install and configure a set of programs for data collection and workflow control during the user study:

1. **Python environment.** Python 3.6[44] is installed inside the VM, alongside with pip package manager and several commonly used Python packages for the user study tasks. The user is free to install any additional packages they need during the development.
2. **IDE with plugin.** PyCharm Community Edition 2020.1, with the plugin described in Section 4.3 is installed. This provides consistent Python development environment for the user study and the testing of the code generation and retrieval. The plugin also handles various data collection processes inside the IDE.
3. **Man-in-the-middle proxy.** We install `mitmproxy`[45] in the VM, along with our customized script sending logs back to our server. This infrastructure enables interception and data collection of both HTTP and secured HTTPS requests. With this we can collect users' complete web browsing activities during the user study.
4. **Web browser.** We install Firefox browser,[46] configured to use the proxy mentioned above so that all users' browsing activities could be logged for analysis.
5. **Keylogger.** We develop a program that runs in the background during the user study, and logs all the user's keystrokes along with the timestamps to our server. With the keylogger

[40] https://releases.ubuntu.com/18.04/
[41] https://www.xfce.org/
[42] https://www.virtualbox.org/wiki/Downloads
[43] https://www.vagrantup.com/
[44] https://www.python.org/
[45] https://mitmproxy.org/
[46] https://www.mozilla.org/en-US/firefox/

we can collect data outside the IDE about the users' activities. This data is useful for mining and analyzing developer activity patterns in terms of keyboard operations, for example copy and pasting shortcuts.

6. **User study control scripts.** We provide users a handful of scripts for easy and fully automatic retrieval, start and submission of the tasks. The scripts allow user to check their completion status of the whole study, as well as to pause and resume during a task for a break. All the user's task start, pause, resume, and submission events are logged so that the completion time of each task for the user could be calculated.

4.11.2 Pre-test Survey Details

For each of the prospective participants, we asked them about two parts of the information in a pre-study survey, apart from personal information for contact purposes. The first is regarding programming experience, used to determine if the participants have enough expertise in Python as well as the categories of tasks that we designed. The questions are:

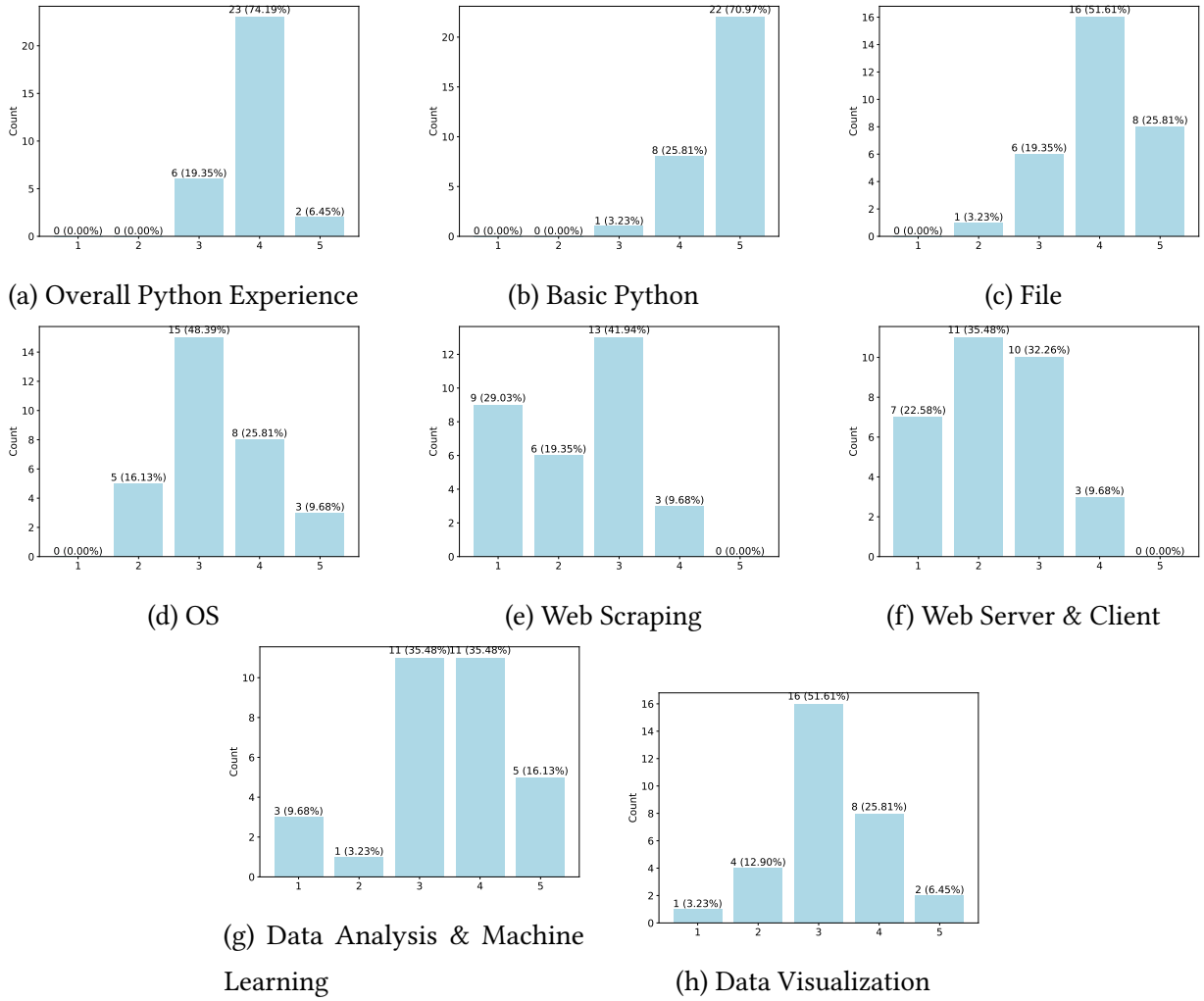
1. Which of the following best describes your current career status: Student (computer science), Student (other field), Software Engineer, Data Scientist, Researcher, Other.
2. How do you estimate your programming experience? (1: very inexperienced to 5: very experienced)
3. How experienced are you with Python? (1: very inexperienced to 5: very experienced)
4. How experienced are you with each of the following tasks in Python? (1: very inexperienced to 5: very experienced) Basic Python, File, OS, Web Scraping, Web Server & Client, Data Analysis & Machine Learning, Data Visualization.

The second part of the information is about their development preferences, used to ask for their preferences with IDE and assistive tools. The questions are:

1. What editor/IDE do you use for Python projects? Vim, Emacs, VSCode, PyCharm, Jupyter Notebook, Sublime Text, other.
2. Do you use any assistive tools or plugins to improve your coding efficiency? Some examples are code linting, type checking, snippet search tools, etc. If yes, what are they?

4.11.3 Participants Programming Experience

The detailed participants' programming experience responded in the survey is shown in Figure 4.8.



(a) Overall Python Experience

(b) Basic Python

(c) File

(d) OS

(e) Web Scraping

(f) Web Server & Client

(g) Data Analysis & Machine Learning

(h) Data Visualization

Figure 4.8: The experience and expertise for overall Python programming and 7 specific areas that we design different tasks for, from all the participants that completed the survey. 1 represents very inexperienced and 5 represents very experienced.

4.11.4 Post-study Survey Details

After each task, we ask the following questions to all users (disregarding using the plugin or not) about the task design, self-assessment, as well as the help needed during the process:

1. How difficult did you feel about the task? (1: very easy to 5: very hard)
2. How would you evaluate your performance on the task? (1: very bad to 5: very good)
3. How often did you need to look for help during the task, including web search, looking up API references, etc.? (1: not at all to 5: very often)

For users that completed the current task with plugin enabled, the following additional questions about the plugin user experience are asked:

1. How do you think the plugin impacted your efficiency timewise, if at all? (1: hindered significantly, to 3: neither hindered nor helped, to 5: helped significantly)
2. How do you think the plugin impacted your quality of life, with respect to ease of coding, concentration, etc., if at all? (1: hindered significantly, to 3: neither hindered nor helped, to 5: helped significantly)

After all assigned tasks are completed for the user, we ask them to complete a form about the overall experience with the user study and the evaluation of the plugin, as well as soliciting comments and suggestions.

1. What did you think of the tasks assigned to you in general?
2. Overall, how was your experience using this plugin? (1: very bad to 5: very good)
3. What do you think worked well, compared with your previous ways to solve problems during programming?
4. What do you think should be improved, compared with your previous ways to solve problems during programming?
5. Do you have any other suggestions/comments for the plugin?

4.11.5 Plugin Effect on Code Complexity Metrics

We also analyze the plugin's effect on code complexity metrics, following the same methods used in Section 4.5. We measure two standard proxies for code complexity of the Python programs produced by our study participants in each of their assigned tasks, *i.e.*, the number of source lines of code (SLOC) and McCabe's cyclomatic complexity (CC), a measure of the number of linearly independent paths through a program's source code [229]; in real programs, CC depends a lot on the "if"-statements, as well as conditional loops, and whether these are nested. The two measures tend to be correlated, but not strongly enough to conclude that CC is redundant with

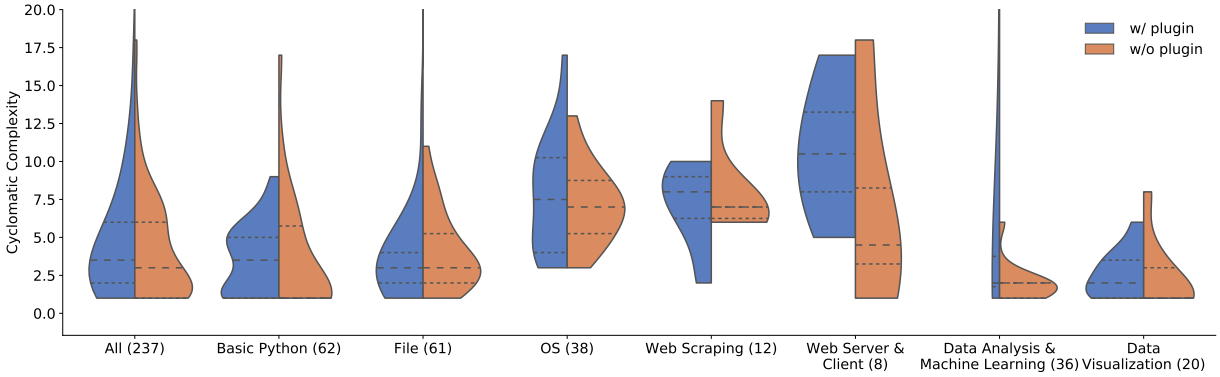


Figure 4.9: Distributions of cyclomatic complexity values across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

SLOC [185]. We use the open-source library Radon[47] to calculate CC.

One could expect that code produced by our NL2Code plugin may be more idiomatic (possibly shorter and less complex) than code written by the participants themselves.

Figure 4.9 shows the distributions of CC values across tasks and conditions. Figure 4.10 shows the distributions of SLOC values across tasks and conditions.

Table 4.8 summarizes our default specification mixed-effects regressions with CC and SLOC variables included; the models with our second specification (de-meaned task experience) are shown in Appendix 4.11.7. The models fit the data reasonably well (R^2_c = 50% for SLOC, R^2_c = 27% for CC).

Analyzing the models we make the following observations. There is no statistically significant difference between the two conditions in cyclomatic complexity values (model (4)). That is, the code written by users in the plugin condition appears statistically indistinguishably as correct and as complex from the code written by users in the control group.

We note a small effect of using the plugin on code length (model (3)). On average, the code written by users in the plugin condition is ~4 source lines of code longer than the code written by users without using the plugin. However, this effect is quite small, smaller than the standard deviation of the random `user` intercept (~6 source lines of code).

4.11.6 NL2Code Plugin Query Syntax

For the best results from the code generation model, we also instruct the users to write queries as expected by the model with the following rules:

[47] https://github.com/rubik/radon

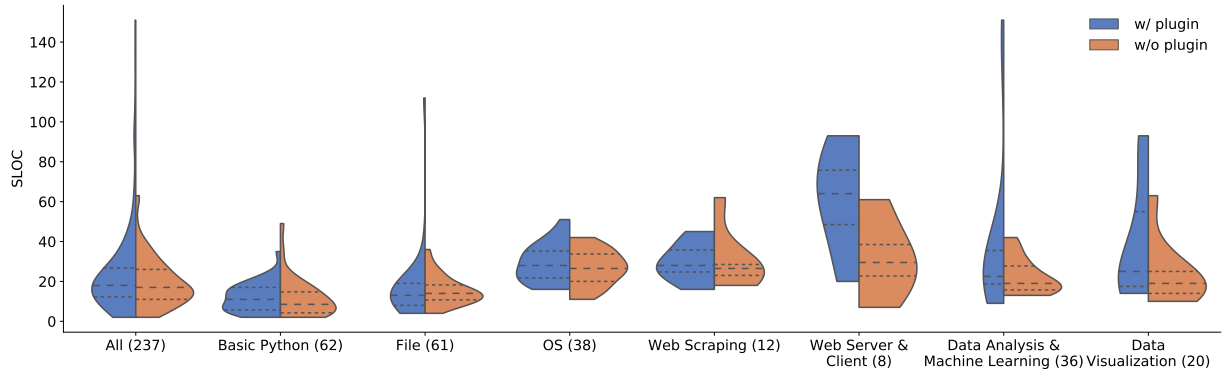


Figure 4.10: Distributions of SLOC values across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

- Quote variable names in the query with grave accent marks: ... `variable_name` ...
- Quote string literals with regular quotation marks: ... “Hello World!” ...
- Example query 1: open a file “yourfile.txt” in write mode.
- Example query 2: lowercase a string `text` and remove non-alphanumeric characters aside from space.

4.11.7 Task Performance Models (De-meaned Specification)

Table 4.9 summarizes our alternative specification (de-meaned task experience) mixed-effects regressions for two response variables in the main article, plug two response variables (CC and SLOC) introduced in Appendix 4.11.5.

4.11.8 User Queries

Table 4.10: Unique successful user queries to the NL2Code plugin, per task, for the 31 study participants. Queries for which the participant chose a snippet produced by the code generation model are shown in boldface, and in the remainder a retrieved snippet was used.

Task	Queries	
T1-1	**call `pick_with_replacement`**	**how to generate random letter**
	create a dictionary with keys `random_letters` and values `random_numbers`	import library random
	create dictionary	**list to dict**
	create empty dictionary	loop on numbers from 0 to 100
	create list "a_list"	**loop over a range of `count`**
	defaultdict	merge 2 dictionaries

Table 4.10: (continued)

Task	Queries	
	dictionary of characters and int	**pair characters in `characters` and numbers in `numbers`**
	for loop on range 100	**print `dic` keys on each line**
	generat integers 1-20	**print `dic` keys sorted**
	generate 100 integers (1-20 inclusive).	**print `dic` sorted by keys**
	generate 100 random lower-cased leters	print a to z
	generate 100 random lowercase letters	**print list**
	generate 100 random numbers	print list as string
	generate 100 random numbers from 1 to 20	print list elements
	generate a rondom lower case character	print without newline
	generate char lower case	random
	generate dict	random character between a and z
	generate list of random charachters	random characters
	generate lowercase char	**random integer between 1 and 20**
	generate random	**random number**
	generate random between 0 and 20	**random sample with replacement**
	generate random charachter	randomly generate 100 letters
	generate random int	**randomly pick an item from `seq`**
	generate random letters	**rearrange dictionary keys into alphabetic order**
	generate random lower case letters	**sort a list**
	generate random nu,ber	**sort a list into ascending order**
	generate random number	**sort a list x into ascending order**
	generate random numbers	**sort dict by key**
	generate random numbers between 1-20 inclusive	**sort key of dict**
	get a random letter	sort list
	given list `letters` and `integers`, create a dicitonary such that the values in `letters` are keys and values in `integers` are values	**sort list 'values' into ascending order**
	how to append value in dict	squence of integers from 1 to 20 inclusive
	how to check if a key is in a dictionay	zip 2 lists
	how to generate random int in range between 1 and 20	**zip `hundred_characters` with `hundred_numbers`**
T1-2	add a week to a datetime	get gmt timezone
	add days to time	get now one week from now
	assign current date and time to `now`	**get the current date in utc**
	change date format	**get the current time in utc**
	change datetime format of `week_date` to mm-dd-yyyy hh:mm	**get the date and time a week from now in gmt**
	convert `week_date` to GMT timezone and assign to `GMT_week_date`	get time and date
	convert date timezone	**get time and date in gmt in `date`**
	date from 7 days	**get time and date one week from now**
	date gmt	**get time now**
	date now	gmt
	datetime	gmt time 24
	display `week_date` in format mm-dd-yyyy hh:mm	import datetime
	format datetime	**import time**
	format datetime 24 hour	mm-dd-yyyy
	format time	**print current date time**
	get current datetime	**print date and time in GMT in 24hr format**
	get date 7 days from today	**print datetime in mm-dd-yyyy hh:mm format**
	get date and time in gmt	**time add**
	get date and time one week from now	time and date
	get date time one week from now	**time and date in certain**
	get datetime	**timedelta**
T2-1	**copy column from "data.csv" file to another "output.csv"**	**new line**
	copy column from "data.csv" to "output.csv"	**number of columns of csv**
	create 'output.csv' csv file	open "data.csv" file
	csv write	**open a csv file `data.csv` and read the data**
	csv writer	**open csv**
	cvs	**open csv file `data.csv`**

Table 4.10: (continued)

Task	Queries	
	cvs files	**open csv file with read and write**
	delete a column in csv	**open file**
	delete column from csv	**pandas read csv**
	delete column from csv file	**pandas read csv named "data.csv"**
	delete first and last column in csv file	print csv without row numbers
	delete first and last column of `df`	python make dir
	delete first and last row from the dataframe `df`	**read "data.csv" file**
	delete first row from dataframe `df`	**read csv file "data.csv"**
	delete row in csv	**read csv file using pandas**
	delete the first column in csv file `df`	read csv pure python
	file to csv	read cvs
	get current path	remove columns from csv file and save it to another csv file
	get specific columns by index in pandas data frame	**remove first column from csv file**
	headers in a dataframe	save `df` to a file `output.csv` in a new directory `example_output`
	how to delete a column in a dataframe python	**save dataframe to csv**
	how to delete columns in dataframe	save pandas dataframe to a file
	how to save a dataframe in csv file	**save this dataframe to a csv**
	if dir exist	**write `output` to csv file**
	if directory "output" exists	**write csv `output_f` to file "output/output.csv"**
	make directory	**write output to csv file "output.csv"**
	make directory "output" if it doesn't exist	**write to csv file**
T2-2	change directory	**list files in folder**
	change directory to "data"	**list of filenames from a folder**
	check file encoding	**move file to other directory**
	check if directory exists	**normalize newlines to \n**
	convert binary decoded string to ascii	**open file**
	convert file encoding	**open text file**
	convert file to utf	read a file and iterate over its contents
	convert latin-1 to utf-8	**read all files under a folder**
	convert str to utf-8	**read file**
	convert text file encoding	read ISO-8859-15
	convert text files from encoding ISO-8859-15 to encoding UTF-8.	readline encoding
	copy a file	redirect
	copy file	**remove header**
	copy file `ddd.png`	remove heading white space
	copy file to other folder	text normalize newlines to \n
	covert file to utf	traverse a directory
	find character	**travverse list of files**
	get all files in directory	**trim heading whitespace**
	get the file extension	**trim the heading and trailing whitespaces and blank lines for all text files**
	iterating files in a folder	unkown encoding
	list all text files in the data directory	write to file
	list files in directory	
T3-1	**check if `file` is a directory**	match regex year month day
	check if string has specific pattern	**move file**
	copy a file to dist	move files from directory to directory
	copy all files and directories from one folder to another	recursive copy files and folders
	copy directory to another directory	**recursively iterate over all files in a directory**
	copy directory to directory	**regex dd-mm-yy**
	copy directory tree from source to destination	regex digit python
	copy file from `src_path` to `dest_path`	regex for date
	copy files	**regex replace capture group**
	copy files and directories under `data` directory	regexp date
	copy files creating directory	rename file
	copy files from folder	**rename file with regex**
	create file	**rename files**
	create folder	replace pattern in string

Table 4.10: (continued)

Task	Queries	
	datetime to string	**search all matches in a string**
	extract year month day from string regex	**search for pattern "%d%d-%d%d" in `file`**
	get all files and folders	walk all files in a directory
	get the files that inside the folders	walk all nested files in the directory "data"
	list all filepaths in a directory	**walke all files in a directory**
	make a folder recersively	write to file
T3-2	**add entry to json file**	**load json file**
	check if file `output_file` exists	load json from a file
	check if file ends with .json	**read a json file named `f`**
	convert dict to string	sorting a dictionary by key
	convert list to dictionary	write into txt file
	import json parsing library	write json in `ret` to file `outfile`
T4-1	find all bold text from html `soup`	parse all hyperlinks from `r` using bs4
	find all hrefs from `soup`	visit `url` and extract hrefs using bs4
	find all red colored text from html `soup`	visit the given url `url` and extract all hrefs from there
	go to a url	**visit the url `url`**
	how to get page urls beautifulsoup	
T4-2	create directory	**regex []**
	download an image request	save dict to csv
	extract imafe from html	**save table beautifulsoup**
	http reques get html	
T5-1	**add json file to a list**	check email correctness
T5-2	argparse subprogram	**print format**
	exit program	request with params
	gET request to "https://jsonplaceholder.typicode.com/posts" with argument userId	
T6-1	a list of dictionary to pandas dataframe	pandas change dataframe column name
	add a new column to a dataframe row	pandas create buckets by column value
	average by group pandas	pandas dropnan
	cast a float to two decimals	pandas get average of column
	cast a list to a dataframe	pandas group by
	column to integer pandas	**pandas join dataframes**
	create a dataframe from a list	pandas join series into dataframes
	csv	pandas output csv
	csv write	pandas print with two decimals
	delete coloumn pd	**pandas read from csv**
	df set column to 7 decimals	**pandas round value**
	filter df with two conditions	pandas save csv two decimal
	filter values in pandas df	**pandas to csv**
	find unique data from csv	pandas to csv decimal
	findall	pandas write df to csv
	floating data in csv group in digit	pandas write to csv file
	format output to 2 decimal	pandas write to file decimal
	get average of row values in pandas dataframe	**read csv**
	get average value from group of data in csv	**read csv file**
	get the head of dataframe `df`	remove repeated column in csv file
	group by range pandas	**rename column pandas**
	group of data from csv	rename pandas df columns
	how to combine 2 lists into a dictionary	**round a variable to 2dp**
	how to remove an item from a list using the index	save `compan_df` dataframe to a file
	import pandas	save `compand_df` dataframe to a file
	list to an entry in pandas dataframe	**sort dataframe `jdf` by `scores`**
	load csv file with pandas	**sort dataframe `jdf` by the values of column 'scores'**
	loop files recursive	**sort pandas dataframe**
	newline space	standard deviation from group of data in csv
	pandas add new column based on row values	two deciaml place
	pandas calculate mean	**write `final_data` to csv file "price.csv"**
T6-2	cross validation in scikit learn	multinomial logistic regression model
	cross validation mean accuracy	**numpy load from csv**

Table 4.10: (continued)

Task	Queries	
	disable warnings	run 5-fold accuracy
	how to determine cross validation mean in scikit learn	set numpy random seed to 0
	how to split dataset in scikit learn	sklearn 5 fold cross validation
	how to split dataset in scikit learn	sklearn 5-fold cross validation
	linear regressor 5 folder cross validation	sklearn cross validation x, y for 5 folds
	load wine dataset	sklearn ignore warnings
T7-1	how to choose plot size in inches	plt set x axis tick range
	how to choose plot title in matplotlib	plt set xtick font size
	how to create ascatter plot using matplotlib	**reformat date**
	how to draw scatter plot for data in csv file	**save plot as image**
	plt create figure with size	**save plt figure**
	plt date as x axis	**scatter**
	plt set x axis label	scatter plot purple
T7-2	bar graph side by side	**plot bar**
	bar plot with multiple bars per label	plot size
	get height of bars in subplot bar gaphs	plot title
	get labels above bars in subplots	**plt ax legend**
	group pandas df by two columns	plt ax xlabel
	horizontal subplot	plt create 3 subplots
	import matplotlib	plt set title for subplot figure
	matplotlib grouped bar chart	plt set x tick labels
	matplotlib multiple histograms	plt show values on bar plot
	matplotlib theme	**pyplot subplots**
	pandas dataframe from csv	select row pandas
	pandas dataframe groupby column	

4.11.9 Randomly Sampled User Queries for the Oracle Analysis

Table 4.11: Sampled user queries for the oracle analysis. Queries for which the user chose a snippet from the code generation model are shown in boldface. • denotes queries "good enough" on their own; ○ denotes queries good enough given the rest of the source file as context; the former is a strict subset of the latter.

Task	Queries	
T1-1	**call `pick_with_replacement` ○**	defaultdict
	generate lowercase char •○	for loop on range 100 •○
	generate random between 0 and 20 •○	generate char lower case
	random sample with replacement •○	generate random letters •○
	sort key of dict •○	random characters
T1-2	**change datetime format of `week_date` to mm-dd-yyyy hh:mm •○**	format datetime
	convert `week_date` to GMT timezone and assign to `GMT_week_date` •○	get gmt timezone ○
	print datetime in mm-dd-yyyy hh:mm format •○	get now one week from now •○
	date now •○	get time and date •○
T2-1	**remove first column from csv file •○**	how to delete columns in dataframe ○
	csv writer	open "data.csv" file •○
	how to delete a column in a dataframe python ○	
T2-2	traverse a directory ○	
T3-1	**copy a file to dist ○**	recursive copy files and folders ○
	match regex year month day	regexp date

Table 4.11: (continued)

Task	Queries	
T4-2	download an image request	save dict to csv
T5-2	**exit program** ●○	argparse subprogram
T6-1	**load csv file with pandas** ●○	how to remove an item from a list using the index ●○
	pandas round value ○	pandas create buckets by column value
	pandas to csv	pandas group by
	read csv file ●○	pandas output csv ○
	rename column pandas ○	pandas to csv decimal ○
	filter df with two conditions	pandas write df to csv
T6-2	load wine dataset	
T7-1	**plt create figure with size** ●○	**scatter** ○
T7-2	**plt ax legend** ○	plt create 3 subplots ●○
	bar plot with multiple bars per label ○	

Table 4.3: LMER task performance models (default specification).

	Dependent variable:	
	Completion time	Correctness score
	(1)	(2)
Experience	−195.62	0.07
	(183.11)	(0.24)
Uses plugin	15.76	0.44
	(196.11)	(0.30)
Constant	3,984.51***	5.88***
	(838.07)	(1.03)
Observations	224	237
Num users	31	31
Num tasks	14	14
sd(user)	1489.25	0.82
sd(task)	1104.7	1.14
R2m	0.004	0.008
R2c	0.642	0.289
Akaike Inf. Crit.	3,987.14	1,106.66
Bayesian Inf. Crit.	4,007.61	1,127.46
Note:		*p<0.1; **p<0.05; ***p<0.01

Table 4.4: Most important 20 features and their weights from the logistic regression modeling whether successful plugin queries result in generated or retrieved code snippets.

Generation				*Retrieval*			
Weight	Feature	Weight	Feature	Weight	Feature	Weight	Feature
0.828	open	0.352	current	0.471	letters	0.294	extract
0.742	time	0.345	delete row	0.442	copy	0.289	set
0.676	sort	0.345	random number	0.438	matplotlib	0.289	plt set
0.590	read csv	0.339	trim	0.437	datetime	0.282	read file
0.556	list	0.330	text file	0.410	python	0.282	cross validation
0.507	number	0.326	keys	0.365	column csv	0.274	scikit
0.402	search	0.310	round	0.361	bar	0.274	dataframe csv
0.399	open file	0.293	numbers	0.344	copy files	0.274	sklearn
0.385	dictionary	0.291	row dataframe	0.334	delete column	0.272	digit
0.353	read	0.290	load csv	0.302	write file	0.270	folders

Table 4.5: Contingency tables for the two oracle comparison scenarios in Section 4.6.2; see Table 4.10 in Appendix 4.11.8 for the actual queries.

Snippet	*Query*			
Generation	Good enough as is		Good enough w/ context	
	False	True	False	True
False	23	8	15	16
True	7	12	1	18

Table 4.6: Most frequently added/deleted tokens after user edits to plugin-returned code snippets.

Addition				*Deletion*			
ΔFreq.	Token	ΔFreq.	Token	ΔFreq.	Token	ΔFreq.	Token
0.0040	in	0.0016	w	-0.0071	2	-0.0016	In
0.0037	for	0.0015	with	-0.0071	1	-0.0016	11
0.0030	line	0.0015	``	-0.0043	a	-0.0015	y
0.0024	file	0.0015	days	-0.0038	0	-0.0014	Seattle
0.0023	key	0.0015	cur_v	-0.0034	3	-0.0014	12
0.0023	os.path.join	0.0015	company_info	-0.0025	plt	-0.0013	4
0.0021	dic	0.0015	n	-0.0023	50	-0.0013	iris
0.0021	filename	0.0014	output	-0.0021	id_generator	-0.0013	string.digits
0.0018	print	0.0014	codecs.open	-0.0018	Out	-0.0013	10
0.0017	if	0.0014	v	-0.0017	df	-0.0013	matplotlib.pyplot

Table 4.7: Frequency changes of different token types after user edits to plugin-returned code snippets. Sorted in descending order, positive number represents addition and negative number represents deletion.

ΔFreq.	Type	ΔFreq.	Type	ΔFreq.	Type	ΔFreq.	Type
0.0138	NAME	0.0053	DEDENT	0.0004	COMMENT	-0.0095	OP
0.0053	INDENT	0.0022	STRING	-0.0049	NEWLINE	-0.0248	NUMBER

Table 4.8: LMER task performance models (default specification, w/ code complexity metrics).

	Dependent variable:			
	Completion time	Correctness score	SLOC	CC
	(1)	(2)	(3)	(4)
Experience	−195.62	0.07	−0.62	−0.21
	(183.11)	(0.24)	(1.61)	(0.46)
Uses plugin	15.76	0.44	4.16**	0.73
	(196.11)	(0.30)	(1.91)	(0.58)
Constant	3,984.51***	5.88***	27.15***	5.64***
	(838.07)	(1.03)	(7.40)	(1.95)
Observations	224	237	237	237
Num users	31	31	31	31
Num tasks	14	14	14	14
sd(user)	1489.25	0.82	6.16	1.18
sd(task)	1104.7	1.14	12.65	2.33
R2m	0.004	0.008	0.011	0.006
R2c	0.642	0.289	0.502	0.27
Akaike Inf. Crit.	3,987.14	1,106.66	2,002.42	1,417.27
Bayesian Inf. Crit.	4,007.61	1,127.46	2,023.23	1,438.08

Note: *p<0.1; **p<0.05; ***p<0.01

Table 4.9: LMER task performance models (de-meaned experience, w/ code complexity metrics).

	Dependent variable:			
	Completion time	Correctness score	SLOC	CC
	(1)	(2)	(3)	(4)
Experience BTW	−478.55	−0.04	−1.47	0.04
	(566.62)	(0.43)	(2.98)	(0.74)
Experience WI	−166.14	0.12	−0.30	−0.35
	(191.33)	(0.29)	(1.87)	(0.56)
Uses plugin	14.47	0.44	4.15**	0.74
	(196.07)	(0.30)	(1.90)	(0.58)
Constant	5,142.42**	6.32***	30.59**	4.62
	(2,348.61)	(1.77)	(12.60)	(3.07)
Observations	224	237	237	237
Num users	31	31	31	31
Num tasks	14	14	14	14
sd(user)	1482.32	0.81	6.15	1.17
sd(task)	1107.9	1.13	12.69	2.32
R2m	0.012	0.008	0.012	0.007
R2c	0.643	0.287	0.504	0.269
Akaike Inf. Crit.	3,988.86	1,108.56	2,004.30	1,419.09
Bayesian Inf. Crit.	4,012.74	1,132.84	2,028.58	1,443.36
Note:				*p<0.1; **p<0.05; ***p<0.01

Part III

Study of Retrieval-Augmented Models

Chapter 5

Capturing Structural Locality in Non-parametric Language Models (Completed)

Structural locality is a ubiquitous feature of real-world datasets, wherein data points are organized into local hierarchies. Some examples include topical clusters in text or project hierarchies in source code repositories. In this paper, we explore utilizing this structural locality within *non-parametric language models*, which generate sequences that reference retrieved examples from an external source. We propose a simple yet effective approach for adding locality information into such models by adding learned parameters that improve the likelihood of retrieving examples from local neighborhoods. Experiments on two different domains, Java source code and Wikipedia text, demonstrate that locality features improve model efficacy over models without access to these features, with interesting differences. We also perform an analysis of how and where locality features contribute to improved performance and why the traditionally used contextual similarity metrics alone are not enough to grasp the locality structure.

5.1 Introduction

Language models (LMs) predict a probability distribution over sequences, and are most widely studied to model and generate natural languages [23, 31, 47, 233]. Advances in LMs benefit many natural language processing downstream tasks, such as machine translation [24], dialog systems [330], question answering [288, 369], and general representation learning for natural

language [73, 217]. Recently, LMs have also been adopted to model sequences other than text, such as source code written in programming language [11, 125, 131, 158], which can enable useful downstream tasks like code completion [296].

Most current neural LMs are based on *parametric* neural networks, using RNN [236] or Transformer [347] architectures. These models make predictions solely using a fixed set of neural network parameters. Recently, more and more neural LMs also incorporate *non-parametric* components [101, 110, 120, 165], which usually first select examples from an external source and then reference them during the prediction. For example, Khandelwal et al. [165] model the token-level probability by interpolating the parametric LM probability with a probability obtained from the nearest context-token pairs in an external datastore. Using such non-parametric components in LMs is beneficial because the model no longer needs to memorize everything about the language in its parameters.

For such non-parametric LMs, one important concept is a *distance* metric between the current context and other contexts in the datastore. One example of such metric is the ℓ^2 distance between context vectors calculated by the parametric model [165]. This distance can be used in both retrieval and probability calculation; items in the datastore that are less distant from the current context are more likely to be retrieved and have a higher influence on the final probability. However, given that non-parametric datastores are typically very large, containing a myriad of contexts from disparate sources, calculating a metric that accurately reflects semantic similarities is non-trivial; as we demonstrate in experiments, there is much room for improvement in current practice.

In this paper, we argue that the relevance of contexts may be correlated with not only contextual distance, but also structural characteristics of the underlying data. Specifically, we take advantage of a property we dub *structural locality*, the propensity of text to be divided into local groups sharing common hierarchical attributes. This property is ubiquitous across many kinds of texts and can provide additional information on how closely related two different examples are to each other. Throughout this paper, we will provide two case-studies of this phenomenon. First, in the domain of programs written in source code, if two source files originate from the same project, they are more likely to be related than files from other projects, and even more so if they are from the exact same package [125]. Second, in natural language, two sections of Wikipedia text may be more related if they fall within the same topical domain, are from similarly titled sections, or even are from the same article (as in Figure 5.1). Notably this locality often manifests itself at different levels, such as the levels of "project", "subdirectory", and "file" cited above for source code.

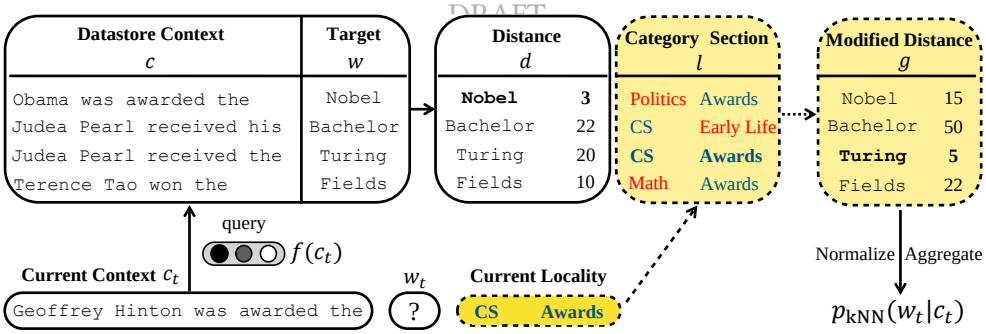


Figure 5.1: An example of incorporating structural locality in the computation flow of $p_{\text{kNN}}(w_t|c_t)$. The current context c_t is used to calculate distance d to contexts in the datastore (c, w). Dashed boxes and lines represent components proposed in our work, which leverage structural information l (non-local, local) to allow for more accurate modified distances g (lower is more similar).

In this paper, we hypothesize that by using multiple levels of structural locality, we can better calibrate the distance metrics used to retrieve examples from non-parametric datastores, thereby improving LM performance. Specifically, we propose a simple-yet-effective approach that can easily be applied to non-parametric LMs: we use different levels of structural locality to define functions that modify the contextual distance metrics used by the non-parametric module.

We evaluate our method on two drastically different domains: Java programming language source code, and natural language Wikipedia articles, achieving noticeable LM performance gains in both by adding just 5 & 7 parameters respectively. Moreover, we perform an in-depth analysis showing how the traditionally used contextual similarity metrics alone are not enough to grasp the locality structure, providing evidence for why adding the locality features is indeed useful. We also compare programming languages and natural languages to highlight several interesting differences in terms of how, and how much, the locality helps improve LM performance.

5.2 Non-parametric Language Models

Given a linguistic context consisting of a sequence of tokens $c_t = (w_1, ...w_{t-1})$, autoregressive parametric LMs estimate $p(w_t|c_t; \theta)$, the probability distribution over the next token w_t. Such parametric LMs store information regarding the language being modeled in the parameters θ. The size of θ is fixed in advance based on the hyperparameters of the model architecture, in recent years typically a neural network [23, 47, 68, 100]. In contrast, a non-parametric LM's number of parameters is not determined by just the model architecture, but also by the underlying data used to train the model. While non-parametric LMs using Bayesian statistics have existed for some

time [120, 316, 361], they have recently seen increased prevalence through the introduction of neural LMs that retrieve relevant examples from an external datastore [110, 115]. In particular, we focus on kNN-LMs [165], a variety of such models that uses a nearest neighbor retrieval mechanism to augment a pre-trained parametric LM, achieving impressive results without any additional training.

Neural network-based LMs usually map the context c to a fixed-length vector representation, with a trained function $f(c)$. In kNN-LMs, the non-parametric component consists of a collection ($\mathcal{D}$) of contexts for the kNN to retrieve from. Denoting these contexts and their corresponding next token as $(c_i, w_i) \in \mathcal{D}$, we create a datastore $(\mathcal{K}, \mathcal{V}) = \{(k_i, v_i)\}$, which contains key-value pairs:

$$(\mathcal{K}, \mathcal{V}) = \{(f(c_i), w_i) \mid (c_i, w_i) \in \mathcal{D}\} \tag{5.1}$$

During inference, the parametric component of the LM generates the output distribution over next tokens $p_{LM}(w_t|c_t;\theta)$ and the corresponding context representation $f(c_t)$, given the test input context c_t. Then the non-parametric component of the LM queries the datastore with $f(c_t)$ representation to retrieve its k-nearest neighbors $\mathcal{N}$ according to a distance function $d(\cdot,\cdot)$. We can then compute a probability distribution over these neighbors using the softmax of their negative distances. The model aggregates the probability mass for each vocabulary item across all its occurrences in the retrieved targets. This distribution is then interpolated with the parametric LM distribution p_{LM} to produce the final kNN-LM distribution:

$$p_{\text{kNN}}(w_t|c_t) \propto \sum_{(k_i,v_i)\in\mathcal{N}} \mathbf{1}_{w_t=v_i} \exp(-d(k_i, f(c_t))) \tag{5.2}$$

$$p(w_t|c_t;\theta) = \lambda p_{\text{kNN}}(w_t|c_t) + (1-\lambda) p_{\text{LM}}(w_t|c_t;\theta) \tag{5.3}$$

In our experiments, we follow Khandelwal et al. [165] in setting the interpolation factor λ to 0.25.

5.3 Defining Structural Locality

We define structural locality as a categorical feature calculated between a pair of contexts (c_i, c_j) in a collection of data, that describes whether the pair share some common, potentially hierarchical, attributes (e.g., the section title of a Wikipedia article section, or the directory path of a source code file). For each domain, a set of hierarchical attributes $\{l_0, l_1, ..., l_n\}$ can be defined based on prior knowledge of the domain. We denote $l_k(c_i, c_j) \in \{0, 1\}$ as the boolean *locality feature* value for the context pair, representing whether c_i and c_j share the same hierarchical

Table 5.1: Locality features designed for each data type according to domain knowledge.

Locality	Wikipedia text	Java projects
l_0	different article category, different section title	different project
l_1	same article category, different section title	same project, different subdirectory
l_2	same section title, different article category	same subdirectory
l_3	same section title, same article category	–

attributes l_k. Here, l_0 is reserved for "no locality", in case the pair shares none of the attributes. Without loss of generality, we set a constraint that $\sum_k l_k(c_i, c_j) = 1$, as new features can be introduced by conjunction and negation of the attributes if needed.

Specific Instantiations. We instantiate these features on our two case studies of Wikipedia text and Java source code, as summarized in Table 5.1.

In Wikipedia, for every context c_i, we define four mutually exclusive hierarchical attributes, $l_0 - l_3$. We calculate these features based on the Wikipedia article and section titles, using simple pattern matching. We then link each article to a set of categories (one article may belong to multiple categories) using the knowledge graph WikiData,[1] by aggregating all the category entities involving two properties: P31 (instance of) and P279 (subclass of). The criterion for "same section title" is exact string match [116]. If there is at least one common category between the sets of categories for two articles, the pair is assigned the "same article category".

For Java source code, we define 3 mutually exclusive attributes, $l_0 - l_2$ based on the location of the code. For each source file, we use the full file path to obtain the two attributes: project name and sub-directory path.[2] The criterion for both "same project" and "same subdirectory" is exact string match. Note that these features are strictly hierarchical, hence only two features are used to capture specific locality here.

An Aside: Connections to Domain Adaptation. Domain adaptation typically refers to reusing existing information about a given problem (e.g., data or model) to solve a task in a new domain. Domain adaptation for neural models generally focuses on fine-tuning models on in-domain data [60, 314] or making direct modifications to the model to consider domain information [44] or latent topic features [170, 235, 352]. Most of these methods do not natively support new test-time contexts that were not seen at training time. In comparison, one immediate

[1] https://www.wikidata.org/

[2] For example, full path `.../Journal.IO/src/main/java/journal/io/api/DataFile.java` has project `Journal.IO` and sub-directory `src/main/java/journal/io/api/` for package `journal.io.api`.

advantage of non-parametric LMs is the ability to adapt to different domains at test time without re-training [100, 101, 165, 232]. For example, some adaptive LMs [100, 101] make use of the previous hidden states of test documents dynamically during inference. Similarly, our proposed locality features do not require re-training on the training set.

Note that within the scope of this paper, although connected, the proposed *structural locality* is a different concept from *domain*. We consider domains as higher-level classifications describing the text where one example belongs to one domain label; e.g., a section about Kim Kardashian's early life belongs to a category of texts describing celebrities. One the other hand, with the structural locality, a user could define multiple levels of locality: to that same section, we can assign not only the domain label, but also, the section title "Early Life". The lightweight nature of our model combined with non-parametric LMs also makes adding more levels of features straightforward, as the features only need to be calculated for the top nearest neighbors, and the number parameters that need tuning in our proposed method (Section 5.5) is only about twice the number of locality features.

5.4 Structural Locality and Nearest Neighbors

In this section, we examine the relationship between distances derived from neural LM features $d(f(c_i), f(c_t))$, structural locality features $l(c_i, c_t)$, and the accuracy of the next-word prediction w_i. Specifically, the underlying assumption of the kNN-LM is that less distant contexts will be more likely to accurately predict the next word w_t. We would like to test whether this correlation between distance $d(\cdot)$ holds uniformly across different locality levels $l(\cdot)$, or if locality provides additional information indicative of whether a particular context is useful for predicting w_i beyond just that provided by the neural representations.

Data. We use two different corpora from different domains to examine this question.

WIKITEXT-103[3] is a standard language modeling benchmark [232] consisting of natural language text from English Wikipedia. It contains a 250K token, word-level vocabulary, with 103M tokens in the training set and 250K tokens in both the validation and test sets.

JAVA GITHUB[4] is a programming language corpus containing Java source code from Github [6] that is widely used in source code modeling [125, 158]. It contains 1.44B tokens from 13,362 projects in the training split, 3.83M tokens from 36 projects in the validation split and 5.33M tokens from 38 projects in the test split. The splits are separated by whole projects. The dataset

[3] https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/.
[4] https://zenodo.org/record/3628665.

is tokenized with byte-pair encoding [313] using the vocabulary from Karampatsis et al. [158] with 2,000 subtokens.

Base Model. As the neural model used to calculate context features, we follow Khandelwal et al. [165],[5] train an LM with the exact architecture and optimization described by Baevski and Auli [23]: a decoder-only Transformer [347], with 1024 dimensional hidden states for the WIKITEXT-103 dataset and 512 for JAVA GITHUB. We set the number of retrieved nearest neighbors to be analyzed to 1024, and the distance metric to ℓ^2 following the default.

Datastore. To capture the effect of our proposed locality features, the datastore should ideally be both closely related to the test examples, sufficiently large to ensure precise kNN retrieval performance for a wide range of contexts, and not too sparse in terms of the prevalence of locality features.

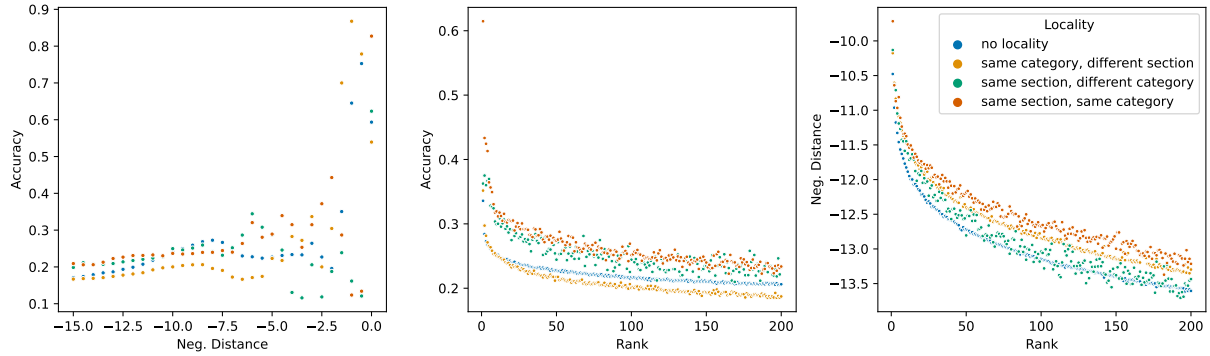
For WIKITEXT-103, we include the training set, as well as the validation/test set (excluding the text currently being evaluated) in the datastore. For the JAVA GITHUB, due to the relatively large size of the validation/test set, and the unwieldy size of the training set, we include only the validation/test set (also excluding the current file).

Analysis. Consider k nearest neighbor contexts $\mathcal{N}_t = \{c_r | r = 1...k\}$ retrieved for any test context c_t in the test set $\mathcal{C}$, ordered by the ascending distance: $\forall r : d(c_r, c_t) < d(c_{r+1}, c_t)$. We define $r \in [1, k]$ as the "rank" for the retrieved context c_r. To study the quality of the retrieved contexts, we calculate the number of correctly retrieved tokens, defined as $\#\{w_r = w_{t_{\text{gold}}}\}$ across $\mathcal{C}$.

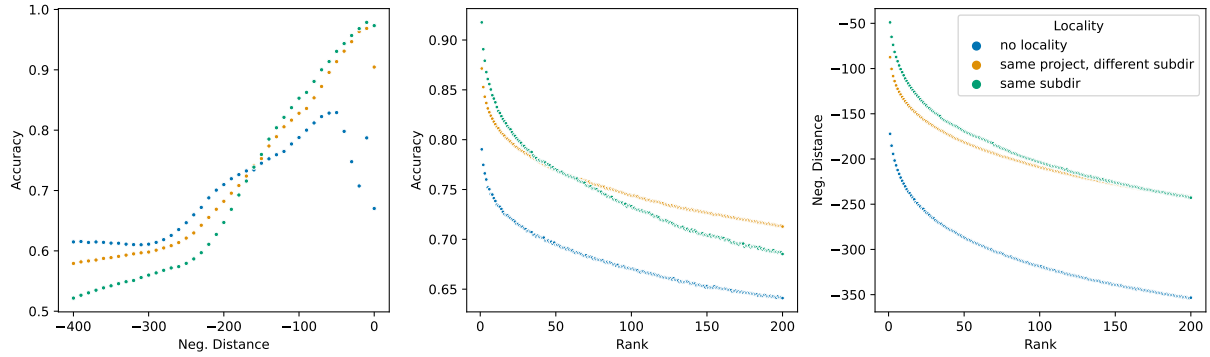
We plot in Figure 5.2, from left to right: (1) Negative distances $\{-d(c_r, c_t) | c_r \in \mathcal{N}_t, c_t \in \mathcal{C}\}$ grouped into bins, vs. the retrieval accuracy of the bin $avg(\#\{w_r = w_{t_{\text{gold}}}\})$. (2) Rank r vs. the retrieval accuracy at rank r, $avg(\#\{w_r = w_{t_{\text{gold}}}\})$. (3) Rank r vs. the average negative distance $avg(-d(c_r, c_t))$ at rank r. All of the plots are grouped by different locality levels l_0 to l_n.

Naturally, the left-most sub-figures reflect that the lower the (negative) distance, the lower the accuracy on both datasets. Yet, interesting, on the Wikipedia domain (Figure 5.2a), as the negative distance gets close to 0 (perfect match), the retrieval accuracy for the next word does not always increase; the accuracy values in this range have very high variance and all 4 levels of locality show no clear trend. This partly indicates that context-based distance is imperfect, regardless of locality. Even so, at slightly lower distances, the trends stabilize and largely show a consistent picture: more specific locality features, especially those involving the same category ($l_1 \& l_3$) yield better predictions than the locality-insensitive component for identical distances. This is especially significant at higher ranked retrievals (middle sub-figure), where contexts

[5] https://github.com/urvashik/knnlm



(a) WIKITEXT-103



(b) JAVA GITHUB

Figure 5.2: The relationship between nearest neighbor rank, negative distance, and the retrieval accuracy, grouped by different locality levels. Only top 200 nearest neighbors are shown for clarity. Negative distance on x-axis represents the upper bound of the bin.

that share the same section title and the same article category are substantially more likely to share the same completion. This suggests that the proposed locality features are not fully represented by, or correlated with the original distance metric, and thus implies that there is room for improvement by incorporating these features.

In the Java source code domain (Figure 5.2b), we generally observe that the retrieval accuracy is much higher than that in the Wikipedia domain, suggesting that the kNN is doing a better job retrieving relevant contexts. This is largely due to higher repetitiveness of source code [131]; as we show later, the base Transformer model also performs much better here than on natural language text (Section 5.6.2). We also observe a more pronounced locality effect here: at the same distances close to 0 and at the same rank, neighbors that are local to the current context have far higher accuracy, indicating usefulness of locality features in the source code domain as well. However, as we can see from the (right-most) plot of rank versus the negative distance,

the average distances of the neighbors with higher locality levels are also significantly smaller than the distance of those without locality. This suggests that the distance in the Java source code domain already correlates well with the level of locality, which may render incorporating locality features less beneficial. We study the precise benefit under one instantiation of this model next.

5.5 Incorporating Structural Locality in Non-parametric LMs

Now that we have demonstrated that locality is additionally indicative of next-word prediction accuracy beyond context distance, we propose a method to incorporate this information into the non-parametric retrieval module. In the case of kNN-LMs (Section 5.2), recall that p_{kNN} is calculated based on the softmax of the negative distance $-d(f(c_i), f(c_t))$. Assuming locality features $\{l_0, l_1, ..., l_n\}$ for each pair (c_i, c_t) consisting the retrieved nearest neighbor and the current inference context c_t, we modify the formulation of p_{kNN} (Equation 5.3) to consider these features as below:

$$p_{\text{kNN}}(w_t|c_t; \{\theta_n\}) \propto \sum_{(k_i,v_i)\in\mathcal{N}} \mathbf{1}_{w_t=v_i} \exp(-g(k_i, c_t; \{\theta_n\})) \tag{5.4}$$

$$g(k_i, c_t; \{\theta_n\}) = g_n(d(k_i, f(c_t)); \theta_n) \text{ if } l_n(c_i, c_t) = 1. \tag{5.5}$$

where $g_n(d(\cdot,\cdot); \theta_n)$ is a learnable function of the distance of the nearest neighbors, with parameter θ_n for each type of locality feature l_n. One can view function $g(\cdot)$ as a "modified" distance for nearest neighbors after taking locality information into consideration. In our experiments, we adopt a linear form of $g(\cdot)$:

$$g_n(d(\cdot,\cdot); w_n, b_n) = w_n d(\cdot,\cdot) + b_n \tag{5.6}$$

We omit the bias for $g_0(\cdot)$ by setting $b_0 = 0$ to remove one free parameter from the model and potentially make optimization easier.[6] To learn these functions, a user needs to provide just a small sample of annotated data in the same domain, as there are only $2n + 1$ parameters to optimize. In our experiments, we use the validation split for optimization. The parameters are

[6]We also experimented with an adaptive variant that conditioned the weights and biases ($\{w_n\}, \{b_n\}$) on the current context representation $f(c_t)$ parameterized by a MLP. However, this did not result in significant improvement over directly optimizing w and b (Section 5.6.3).

trained to minimize the negative log-likelihood of the kNN prediction of the gold token:

$$\arg\min_{\{\theta_n\}} -\log p_{\text{kNN}}(w_t = w_{t_{\text{gold}}} | c_t; \{\theta_n\}) \tag{5.7}$$

To optimize the parameters, we use the Adam [172] optimizer with a learning rate of 0.0001 on the validation set for 200 epochs. It converges within 20 minutes for both datasets.

5.6 How Does Structural Locality Improve Language Modeling?

5.6.1 Experimental Setup

Baselines. Since we base our model on kNN-LMs, this model will be our most directly comparable baseline. We also compare our model to the underlying parametric LM [23], without the kNN module. For the Java Github dataset, we additionally compare to the recent state-of-the-art model from Karampatsis et al. [158] on code language modeling, which uses BPE and LSTMs. In all experiments, the maximum number of tokens per input sample is 3,072.

Evaluation. We evaluate the performance of the LM with the standard perplexity metric and token prediction top-k accuracy on the held-out data.[7] The top-k accuracy is calculated by checking if the ground truth token is among the predicted top-k list. This metric, primarily for $k \in \{1, 5\}$, is commonly used to evaluate predictive models of source code [131]. In order to more easily incorporate and analyze the locality features, and also following Karampatsis et al. [158], we split the evaluation dataset into independent test examples to evaluate, where each of the example is an atomic unit in the locality hierarchy. For Java Github, each test example is a source code file, and for Wikitext-103, each example is an article section.[8]

We show additional results on top-k ($k = 10, 20$) accuracy and relative error reduction (RER) on two datasets in Table 5.3.

[7] For Java Github, the perplexity is calculated on full tokens by aggregating the likelihood of subtokens. The accuracy is calculated that *all* subtokens in a full token have to be predicted correctly.

[8] Note that because we predict Wikitext-103 section-by-section instead of article-by-article the perplexity numbers reported here are somewhat worse than other works. Article-by-article calculation is not inherently incompatible with our proposed method, but it would require additional implementation to use different locality features for different locations in the output. Hence, we used section-by-section calculation for expediency.

Table 5.2: Perplexity and top-k token prediction accuracy results on two datasets. *Uses released pre-trained model, †no stochastic training, for all others stddev < 0.01 for 5 runs.

Dataset	Model	Dev PPL	Test PPL	Rel. Gain	Top-1 Acc. (Rel. Err. Red.)	Top-5 Acc. (Rel. Err. Red.)
WIKITEXT -103	Transformer[1]	*23.31	*23.73	–	39.0% (–)	64.0% (–)
	+kNN[2]	†20.21	†19.94	16.0%	41.3% (3.79%)	66.8% (7.91%)
	+kNN + locality	**19.51**	**19.16**	3.9%	**43.2%** (3.29%)	**68.0%** (3.56%)
JAVA GITHUB	BPE LSTM[3]	–	*5.27	–	–	–
	Transformer	3.29	3.07	41.7%	75.6% (–)	87.6% (–)
	+kNN	†2.43	†2.18	29.0%	83.9% (34.0%)	96.0% (67.9%)
	+kNN + locality	**2.37**	**2.13**	2.3%	**84.7%** (4.91%)	**96.6%** (15.0%)

[1][23], [2][165], [3][158]

5.6.2 Main Results

The high-level results are shown in Table 5.2. At first glance, we can already see that the two datasets vary greatly in predictability. With a similar Transformer architecture, performance on JAVA GITHUB is much better than on the WIKITEXT-103 across the board, partly due to the rigid nature of programming language syntax. With a Transformer model, we achieved a strong state-of-the-art language model on Java code, with low perplexity and very high prediction accuracy (~75%).

By adding kNN module onto the Transformer-based LMs, perplexity and accuracy in both domains improves by a large margin. This is expected and in line with previous experiments on kNN-LMs [165]. The Wikipedia domain enjoys less relative improvement in perplexity (16%) than the Java source code domain (29%). This is particularly interesting, considering that the datastore used for WIKITEXT-103 contains both the current held-out split and the training data (~100M contexts), compared to that for JAVA GITHUB with only the current held-out split (~5M contexts). This reflects the fact that source code is known to benefit strongly from project-

Table 5.4: Learned parameters $\theta_0, \{\theta_n\}$ for each locality level and a non-local level g_0.

	WIKITEXT-103		JAVA GITHUB	
	w	b	w	b
g_0	1.233	–	0.022	–
g_1	1.246	-1.087	0.033	-3.627
g_2	1.288	-1.250	0.041	-5.920
g_3	1.285	-1.464	–	–

Table 5.3: Additional token prediction top-k ($k = 10, 20$) accuracy results and relative error reduction (RER) on two datasets.

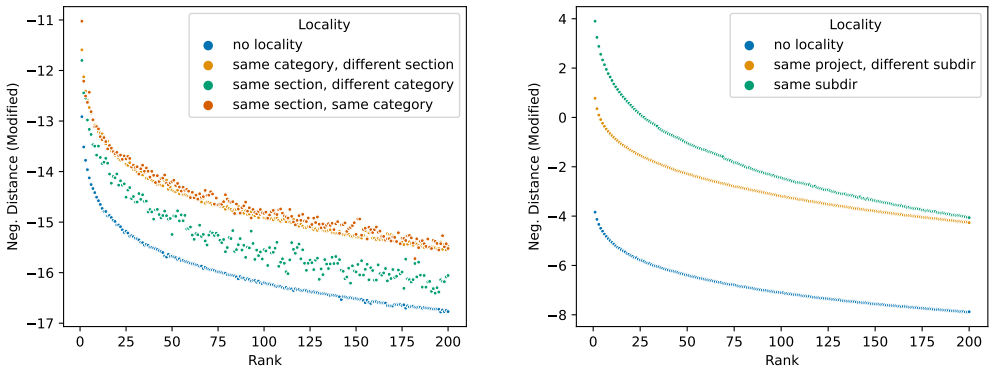
Dataset	Model	Top-10	RER	Top-20	RER
Wikitext-103	Transformer	72.0%	-	78.9%	-
	+kNN	74.6%	9.29%	81.0%	9.98%
	+kNN + locality feat.	**74.9%**	1.30%	**81.1%**	0.84%
Java Github	Transformer	89.5%	-	90.8%	-
	+kNN	97.3%	74.86%	98.2%	80.33%
	+kNN + locality feat.	**97.9%**	21.89%	**98.6%**	25.41%

and package-specific locality [125, 343].

Adding proposed locality features and finetuning the parameters on the validation set improves the performance further on both datasets, albeit with a smaller relative gain. This confirms our hypothesis that incorporating locality into the non-parametric retrieval-based LMs is beneficial. We also see that locality features in the Wikipedia domain result in fairly consistent gains, while the Java source code domain sees especially strong accuracy improvements. This echoes our analysis of the source code corpus in Section 5.4, where we found that distance was generally strongly correlated with accuracy, but that locality was particularly informative at low distances. There, it may help discriminate between top-ranked completion candidates (as also shown later in Tab. 5.5). It is notable that despite the fact that the perplexity and accuracy on Java Github are already very strong with the vanilla Transformer, we still see a noticeable relative error reduction of 4.9% by adding locality levels information.

We next study how locality features guide towards a "better" distance distribution among nearest neighbors. We plot the relationship between the nearest neighbor ranks and "modified" distance $g(k_i, c_t)$ in Figure 5.3. Table 5.4 shows the specific learned parameters for each level of $g(\cdot, \cdot)$. Evidently, the biases and weights vary accordingly with locality levels, as the model tries to "correct" the original distance by emphasizing more local contexts. Compared with the original negative distance $-d(k_i, f(c_t))$ depicted in Figure 5.2, the negative modified distance is more separated between the different locality levels on either dataset, showing the relative importance of different locality more clearly. We analyze several alternative approaches to parameterizing locality in Section 5.6.3.

For Wikitext-103, comparing Figure 5.3a with Figure 5.2a, we can see that with the original



(a) Wikitext-103 (b) Java Github

Figure 5.3: The relationship between nearest neighbor rank and the "modified" negative distance $-g$ guided by our proposed locality features, grouped by different locality levels. Similarly to Figure 5.2, only top 200 nearest neighbors are shown for clarity.

distance different localities cluster together, and the modified distance separates them much better. We can also see that if two contexts have the same article category and the same section title, then their distance on average is the closest, closely followed by those sharing article categories only. On the other hand, contexts that only share section titles are not as closely related. This is intuitively reasonable; the education section for a computer scientist and a musician can be very different in content, even if sharing some commonalities, like the phrases used to describe courses, grades, school locations, etc. This also underscores the usefulness of explicit locality features in providing interpretable insights into the relevance of domain knowledge.

For Java Github, comparing Figure 5.3b with Figure 5.2b, we can see that the original distance is already more separated between different locality levels than that of Wikitext-103, again suggesting better learned representations (in terms of locality sensitivity) for the Java domain. However, the model still benefits somewhat from the contexts that are under the same subdirectory (more so than just the same project), especially for top nearest neighbors: the gap for ranks higher than 80 is more pronounced with the modified distance. This again verifies our hypothesis about the hierarchical nature of structural locality. It also indicates potential practical applications – if this model were deployed in a code editor, one could obtain representations of the files in the same sub-directory as the current file and use them, along with the proposed locality features, to bias auto-complete results.

Table 5.5 shows a randomly sampled test context from each domain where p_{kNN} for the gold token increases after using locality features. We can see that the nearest neighbor search using

context representations performs reasonably well at capturing patterns and themed phrases, especially closer to the last token, finding two very similarly rated candidates. However, in both examples, the second retrieved candidate has a wrong target token. Informed by locality features – in the WIKITEXT-103 example, a matching section *and* category for the first candidate – the more "local" context enjoys a large boost in probability, while the non-local one's decreases slightly. We present additional examples in Table 5.6. The JAVA example demonstrates the same effect; the second retrieved example shows resemblances in variable name and template structures, but the fact that the project is focused on Google API rather than Twitter API makes the original retrieval undesirable.

5.6.3 Alternative Formulations to Learn Parameters for Locality Features

An alternative way to incorporate locality features into the model is an adaptive variant that conditions the weights and biases ($\{w_n\}, \{b_n\}$ in Equation 5.6) on the current context representation $f(c_t)$ parameterized by a MLP:

$$\begin{bmatrix} w_0 & \dots & w_n & b_1 & \dots & b_n \end{bmatrix}^T = MLP(f(c_t)) \tag{5.8}$$

In our experiments, we used a two-layer MLP with ReLU activations, with 64 hidden units and 0.3 dropout rate during training. The perplexity results compared with directly optimizing weights and biases ($\{w_n\}, \{b_n\}$) are shown in Table 5.7.

We find that contextualizing the parameters does not result in significant improvements over directly optimizing w and b, and sometimes makes the performance even worse. This is perhaps because the context vector space is very large (512-1024 dimensions) compared to the relatively few data points from the validation set used to train.

In Section 5.6.2, we discuss the effect of learned parameters for each locality level. Observing that the bias terms (b_i) and weights (w_i) vary according to the locality levels in the learned parameters and to study the weights of the non-local level w_0, we freeze all weights except for non-local weights ($w_{i>0}$) to 1 and only optimize bias terms and the weight for the non-local level (w_0). This is to exacerbate the effect of bias on different locality levels. The learned parameters are shown in Table 5.8. We see similar results where the bias terms vary aggressively to modify the "distance" with different levels of locality, and the weights for the non-local level are less than 1, lowering the importance of those non-local retrieved candidates. It's worth mentioning that for JAVA GITHUB these learned biases are much larger in amplitude than before, to compensate

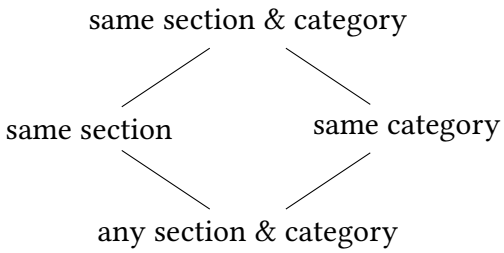
for the small scale weights learned before (only around 0.03). However, the perplexity results on both datasets are slightly worse than the full optimization setting that we use in the main experiments (19.33 vs. 19.16 in WIKITEXT-103 and 2.15 vs. 2.13 in JAVA GITHUB).

5.7 Conclusion

In this paper, we propose a novel method of incorporating structural locality into non-parametric LMs that reference retrieved examples from a datastore. We evaluate this approach in both a natural language and programming language domain, and empirically explore the similarities and differences of how structural locality affects LMs in these settings. The improvement in perplexity and prediction accuracy across both domains show the effectiveness and ubiquity of such locality information. Besides language modeling, we also envision that the method can benefit other applications that could be enhanced using user-defined prior domain knowledge such as conditional generation or representation learning using retrieved information.

Connection with related work and novelty. Previous work [125] made the observation that source code files from the same GitHub repository or sub-directory tend to be relatively similar, but did not include an empirical analysis of this effect. Rather, their observation was backed up by improved performance of their n-gram language model with multiple tiered caches. Our work gives more fine-grained insights into this phenomenon, expands the applicability to neural models and new domains, and proposes a more generalized formulation for encoding multiple localities across multiple domains. Our work improves on this in a number of ways, including: 1) directly examining the internal representations of a neural language model, 2) demonstrating that the internal representations do not sufficiently capture structural locality features, 3) providing efficient ways to compensate for this disconnect, leading to improved language modeling performance, and 4) showing that this carries over to Wikipedia, which has not been previously examined in this way.

As a result, our work both gives more fine-grained insights into this phenomenon and expands the applicability to neural models and new domains. In addition, our work proposes a more generalized formulation for encoding multiple localities across multiple domains than the one proposed in Hellendoorn and Devanbu [125], which treated locality as strictly nested (e.g. project → sub-directory → file). Our formulation in Eq. 5.5 can encode more general hierarchies, such as the *lattice* we used in the Wikipedia case:



Limitations. Our method applies to settings where locality effects are present, there is sufficient data to build a reliable datastore for each locality level, and that locality is not already meaningfully captured by the model. While this may not apply to every domain, these features are common: besides source code & Wikipedia, domains including books (features: authorship & dewey decimal system information), research papers (venue, research area), product manuals (kind, sections), and online discussions (time, topic) are all plausible candidates. The features in our studied domains were selected based on availability and prior knowledge of the domain (e.g., for Java, [125]). While they did provide measurable improvements and were natural to interpret, these may not be the optimal choice, and other options are worth investigating. It is also possible that LM improvements will eventually lead to learned context representations that almost perfectly capture the relevant locality information. However, we believe this to be unlikely: in many practical settings, there is some inherent ambiguity in partial contexts that cannot be solved with the surface text only. For instance, in Java source code files, it is common to declare a package, which will obviously match perfectly based on the first few tokens (e.g., `package org.`) with many other contexts. Yet given the scoped nature of this declaration, locally retrieved continuations are inherently far more useful.

Table 5.5: Examples from two domains where incorporating locality features (non-local, local) lead to a significant increase in the cumulative p_{kNN} for the gold token, with corresponding change in probability (normalized negative distance) for two nearest neighbors.

Test Context	**Test Target**	**Initial** $\log p_{\text{kNN}}$	Δ $\log p_{\text{kNN}}$
Section: Seasonal forecasts; Category: Pacific typhoon season *The forecast indicated the potential for 26.2 tropical storms, compared to the 10– and 30-year average of 27.8 and 26.3 storms, respectively. The following month, the group raised their ...*	forecast	-2.20	+0.89
Datastore Context	**Datastore Target**	**Orig. Log-Prob.**	**ΔLog-Prob.**
Section: Seasonal forecasts; Category: Pacific typhoon season *Their main reasons behind this is due to weaker trade force winds occurring in many parts of the basin, and there would be an enhanced cyclonic vorticity over the northwestern part of the Pacific. On April 27, the GCACIC made their first ...*	forecast	-2.91	+1.25
Section: Earthquake; Category: earthquake *Nickson Sioni from Simbo came on the (HF) radio and announced the arrival of a huge wave that had washed away several houses and come inland about 200m. This information was passed on by telephone to the Hawaii-based Pacific Tsunami Warning Center who then upgraded their ...*	warning	-3.01	-0.31
Test Context	**Test Target**	**Initial** $\log p_{\text{kNN}}$	Δ $\log p_{\text{kNN}}$

Directory: .../android/twitter/AuthConstants.java; Project: twitterdroid
`public static final String CONSUMER_SECRET = "YOUR_CONSUMER_SECRET";`
`public static final String REQUEST_URL = "http://www. ...`
	twitter	-2.03	+0.49

Datastore Context	**Datastore Target**	**Orig. Log-Prob.**	**ΔLog-Prob.**

Directory: .../jtwitter/TwitterConnection.java; Project: twitterdroid
`public static final String FRIENDS_TIMELINE_URL =`
`"http://api.twitter.com/1/statuses/friends_timeline.xml";`
`public static final String UPDATE_URL = "http://api. ...`
	twitter	-1.99	+0.17

Directory: .../impl/ActivityTemplate.java; Project: spring-social-google
`private static final String ACTIVITIES_PUBLIC = "/activities/public";`
`private static final String ACTIVITIES_URL = "https://www. ...`
	googleapis	-1.87	-0.09

Table 5.6: Examples where locality features (non-local, local) lead to a significant increase in the cumulative p_{kNN} for the gold token, with corresponding change in probability (normalized negative distance) for two nearest neighbors.

Test Context	**Test Target**	**Initial** $\log p_{\text{kNN}}$	Δ $\log p_{\text{kNN}}$
Section: Design; Category: ship class *In an effort to outmatch the American New York class, planners called for a ship armed with twelve 14-inch (36 cm) guns and faster than the 21 knots (39 km/h; 24 mph) of their rivals. Vickers files show that the Japanese had access to the designs for double- and triple-gun turrets, yet opted for six double turrets over four triple turrets. The final design—designated A-64 by the IJN—called for a ...*	displacement	-2.54	+1.09
Datastore Context	**Datastore Target**	**Orig. Log-Prob.**	**ΔLog-Prob.**
Section: Design; Category: ship class *Both ships were also given torpedo bulges to improve their underwater protection and to compensate for the weight of the additional armour. In addition, their sterns were lengthened by 7.62 metres (25 ft). These changes increased their overall length to 213.8 metres (701 ft), their beam to 31.75 metres (104 ft 2 in) and their draft to 9.45 metres (31 ft). Their ...*	displacement	-3.25	+1.23
Section: History; Category: gun mount *The British Admiralty ordered a prototype of Coles's patented design in 1859, which was installed in the ironclad floating battery, HMS Trusty, for trials in 1861, becoming the first warship to be fitted with a revolving gun turret. Coles's aim was to create a ...*	ship	-2.98	-0.24
Test Context	**Test Target**	**Initial** $\log p_{\text{kNN}}$	Δ $\log p_{\text{kNN}}$
Section: La Venta; Category: colossal statue *When discovered it was half-buried; its massive size meant that the discoverers were unable to excavate it completely. Matthew Stirling fully excavated the monument in 1940, after clearing the thick vegetation that had covered it in the intervening years. Monument 1 has been ...*	moved	-2.97	+1.22
Datastore Context	**Datastore Target**	**Orig. Log-Prob.**	**ΔLog-Prob.**
Section: San Lorenzo; Category: colossal statue *The sculpture suffered some mutilation in antiquity, with nine pits hollowed into the face and headdress. San Lorenzo Colossal Head 10 (also known as San Lorenzo Monument 89) has been ...*	moved	-4.18	+1.36
Section: San Lorenzo; Category: castle *The excavations investigated the north of the fortress, searching for an entrance postulated by architect Eugene Viollet-le-Duc, but no such entrance was found. However, the excavation did reveal was that there was an addition to the north of the castle to enable the use of guns. Typologically, the structure has been ...*	dated	-4.63	-0.11

Table 5.7: The perplexity results comparing alternative formulation using MLP to contextualize parameters for locality features on two datasets.

Dataset	Model	Dev PPL	Test PPL
Wikitext-103	Transformer	23.31	23.73
	+kNN	20.21	19.94
	+kNN + locality (MLP contextualized)	20.11	19.92
	+kNN + locality (direct)	**19.51**	**19.16**
Java Github	Transformer	3.29	3.07
	+kNN	2.43	2.18
	+kNN + locality (MLP contextualized)	2.47	2.20
	+kNN + locality (direct)	**2.37**	**2.13**

Table 5.8: Learned parameters $\theta_0, \{\theta_n\}$ for each locality level and a non-local level g_0, with fixed $w_{i>0} = 1$ during optimization.

	Wikitext-103		Java Github	
	w	b	w	b
g_0	1.127	–	0.901	–
g_1	1.000	-0.385	1.000	-28.716
g_2	1.000	-0.475	1.000	-55.428
g_3	1.000	-0.726	–	–

Chapter 6

Why do Nearest Neighbor Language Models Work? (Completed)

Language models (LMs) compute the probability of a text by sequentially computing a representation of an already-seen context and using this representation to predict the next word. Currently, most LMs calculate these representations through a neural network consuming the immediate previous context. However recently, *retrieval-augmented LMs* have shown to improve over standard neural LMs, by accessing information retrieved from a large datastore, in addition to their standard, parametric, next-word prediction. In this paper, we set out to understand *why* retrieval-augmented language models, and specifically why k-nearest neighbor language models (kNN-LMs) perform better than standard parametric LMs, even when the k-nearest neighbor component retrieves examples from the same training set that the LM was originally trained on. To this end, we perform analysis of various dimensions over which kNN-LM diverges from standard LMs, and investigate these dimensions one by one. Empirically, we identify three main reasons why kNN-LM performs better than standard LMs: using a different input representation for predicting the next tokens, *approximate* kNN search, and the importance of softmax temperature for the kNN distribution. Further, we incorporate some insights into the standard parametric LM, improving performance without the need for an explicit retrieval component. The code is available at https://github.com/frankxu2004/knnlm-why.

6.1 Introduction

Language modeling is the task of predicting the probability of a text (often conditioned on context), with broad-spanning applications across natural language processing [23, 31, 47, 233].

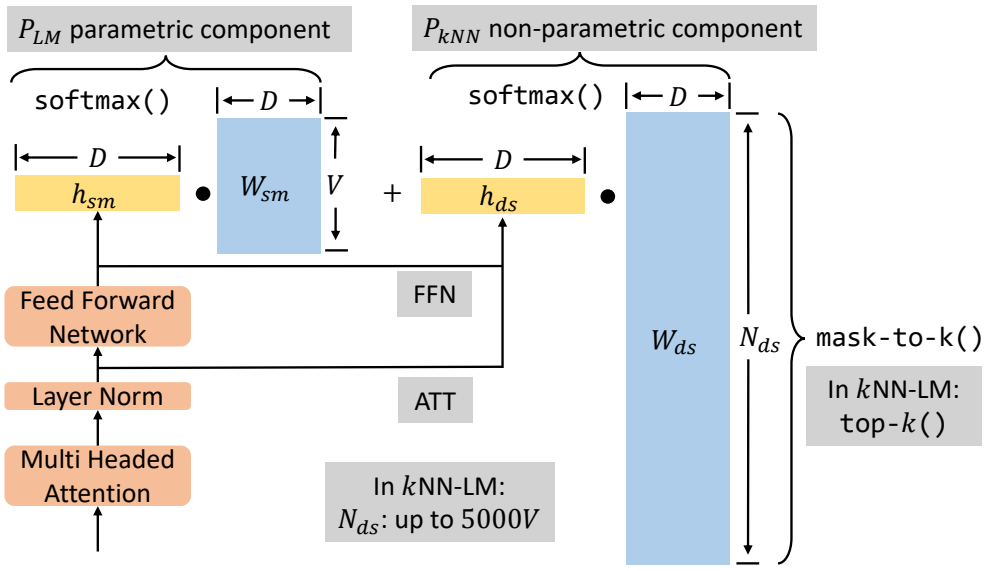


Figure 6.1: An illustration of the generalized formulation of kNN-LM in Equation 6.2.

It is usually done by sequentially encoding a context c_t using a trained neural network function f, and computing the probability of the next word w_t according to $f(c_t)$ and a vector representation of w_t.

Recently, *retrieval-augmented* LMs have shown a series of impressive results [12, 38, 101, 110, 120, 166, 408]. Retrieval-augmented LMs compute next token distribution based not only on the immediately preceding context c_t and the model parameters, but also on an external datastore, from which examples are retrieved and incorporated into the base LM's prediction. One such model that is notable for both its simplicity and efficacy is the k-nearest neighbor language model [kNN-LM; 166]. kNN-LM extends a trained base LM by linearly interpolating the output distribution with a kNN model. The nearest neighbors are retrieved according to the distances between the current context embedding of the base LM and all the context embeddings in the datastore. The datastore is created by encoding all contexts from any text, including the original LM training data.

One of the most surprising results from Khandelwal et al. [166] is that kNN-LM reduces the perplexity of the base LM *even when the kNN component is retrieving examples from the same training set that the LM was originally trained on*, indicating that kNN-LM improves the ability to model the training data and is not simply benefiting from access to more data. Intrigued by this finding, we wonder why does kNN-LM work, and how does it improve already-trained strong transformer-based models? In this paper, we set out to understand why kNN-LMs work even in this setting.

In the following sections, we first elucidate connections between the added kNN component and the standard LM component. Specifically, we note that word distributions from both components are calculated using a softmax function, based on the similarity of the current context hidden state with a set of embeddings that corresponds to different next words. With this intuition, we formalize and generalize the non-parametric distribution with the softmax layer and word embedding layer used in parametric LMs. We then show that this generalized form exposes a variety of design choices, e.g., the number of context embeddings in the datastore, the input representation used in softmax layer, different similarity functions, as well as the approximation and sparsification implementations in the kNN search. This provides a general framework for analyzing kNN-LM and similar models and allows us to perform ablation studies that test the importance of various design decisions.

We proceed to propose multiple hypotheses as to why kNN-LM works, which are testable by adjusting the various parameters exposed by our generalized formulation. Based on these hypotheses, we perform ablation experiments and analyze the nuances between different implementations of the generalized version of P_{kNN}. As the answer to our question, "why do kNN-LMs work?", we eventually show that the most probable reasons are threefold:

1. Ensembling the output of softmax using two representations from different layers of the transformer is important; in our experiments, this accounts for 55% of the performance gain of kNN-LM, or 6.5% relative perplexity improvement compared to the base LM.

2. kNN-LM uses *approximate* nearest neighbor search to handle the large number of candidates, and the lack of preciseness in the algorithm actually helps kNN-LM to generalize *better* than exact nearest neighbor search and distance calculation, possibly due to regularization effect. The relative perplexity improvement from this factor is about 2.6%.

3. Depending on the design decisions that are chosen for modeling, adding a temperature term to the kNN non-parametric component can become crucial to the success of modeling (although coincidentally, in the original settings of Khandelwal et al. [166], a temperature of 1.0 was close to optimal, which hid the importance of this term). In some settings, the relative perplexity gap between the default and optimal temperature can be as high as 8.4%.

Finally, one significant drawback to the current kNN-LM is the inefficiency of kNN search performed at each step [12, 38, 121, 350]. Because of the similarity between kNN-LM and the parametric LM's last layers and the many design choices, we also demonstrate that we are able to make kNN-LM more efficient by substituting the kNN search with another matrix operation that can fit in accelerator memory while maintaining more than half the perplexity improvement, or 6.5% relative improvement to the base LM.

6.2 Formalizing and Generalizing kNN-LM

kNN-LM [166] is a linear interpolation between a base LM and a kNN model. Given a set of contexts c_i and their corresponding next token w_i as a pair $(c_i, w_i) \in \mathcal{D}$, kNN-LMs create a datastore $(\mathcal{K}, \mathcal{V}) = \{(k_i, v_i)\}$, as a set of key-value pairs $(\mathcal{K}, \mathcal{V}) = \{(f(c_i), w_i) \mid (c_i, w_i) \in \mathcal{D}\}$, where $f(c_i)$ is typically a transformer's hidden state after reading c_i. During inference, the parametric component generates the output distribution $p_{LM}(w_t|c_t;\theta)$ over the next tokens and produces the corresponding context representation $f(c_t)$, given the test input context c_t. Then the non-parametric component queries the datastore with the $f(c_t)$ representation to retrieve its k-nearest neighbors $\mathcal{N}$ with a distance function $d(\cdot,\cdot)$. Next, kNN-LM computes a probability distribution over these neighbors using the softmax of their negative distances, and aggregates the probability mass for each vocabulary item across all of its occurrences in the retrieved targets:

$$p_{\text{kNN}}(w_t|c_t) \propto \sum_{(k_i,v_i)\in\mathcal{N}} \mathbf{1}_{w_t=v_i} \exp(-d(k_i, f(c_t))) \tag{6.1}$$

Finally, this distribution is interpolated with the parametric LM distribution p_{LM} to produce the final kNN-LM distribution $p(w_t|c_t;\theta) = (1-\lambda)p_{\text{LM}}(w_t|c_t;\theta) + \lambda p_{\text{kNN}}(w_t|c_t)$, where λ is a scalar that controls the weights of the interpolation between two components, with higher λ putting more weight on the non-parametric component.

Looking closely at Equation 6.1, we notice a similarity between the calculation of P_{kNN} and the standard P_{LM}. The kNN distribution is based on the distances between the current context and the nearest neighbors from the datastore, normalized by a softmax function. Recall that in (standard) parametric language models, the distribution over the vocabulary is also based on a measure of distance, the inner product between the current context embedding and the word embeddings of every token in the vocabulary. Because each context embedding in the datastore $(\mathcal{K}, \mathcal{V})$ corresponds to a target token, we can also view this datastore as a large word embedding matrix with multiple word embeddings for each of the vocabulary words. Theoretically, given unlimited computation, we could calculate the distribution based on the distances to every embedding in the datastore, and aggregate by vocabulary items, making it more closely resemble P_{LM}. For Equation 6.1, this will result in $k = |\mathcal{D}|$, the size of the entire datastore, and $\mathcal{N} = \mathcal{D}$, using the distances to every context in the datastore instead of a subset of nearest neighbors. In practice, we use kNN search as a way of approximation, by limiting the calculation to only k nearest neighbors to avoid the computational cost of calculating the distribution over the entire datastore.

If we re-write and generalize Equation 6.1, both the kNN-LM of Khandelwal et al. [166] and a large number of related models can be expressed through the following equation:

$$\begin{aligned} P_{\text{interp}} = (1-\lambda) \underbrace{\text{softmax}(W_{sm} \cdot h_{sm})}_{P_{\text{LM}} \text{ parametric component}} \\ + \lambda \underbrace{M \text{softmax}(\text{mask-to-k}(W_{ds} \otimes h_{ds})/\tau)}_{P_{\text{kNN}} \text{ non-parametric component}} . \end{aligned} \tag{6.2}$$

Figure 6.1 provides an illustration of Equation 6.2. The first term of the equation is the standard parametric language model, whereas the second represents a generalized version of utilizing an external datastore. The first component, the output layer of a common parametric language model, is relatively straightforward. W_{sm} of size $V \times D$ is the embedding matrix of the output token, and h_{sm} is the context vector to calculate the distribution of the output token, usually the output of the final feedforward layer in the transformer.

In the second component, W_{ds} represents the datastore, of size $N_{ds} \times D$. N_{ds} is the number of entries in the datastore, and D is the size of each context vector. h_{ds} represents the context vector used to query the datastore. As shown in Figure 6.1, h_{ds} may come from a different layer of the transformer than h_{sm}. The operator $\otimes$ represents the operation type used to calculate the similarity between context vectors and the query vector, which also has several alternatives that we discuss below. mask-to-k$(\cdot)$ represents a function to sparsify similarity scores across the datastore, setting all but k similarity scores to $-\infty$, which results in probabilities of zero for all masked similarity scores after the softmax. Practically, this is necessary for kNN-LMs because the size of the datastore N_{ds} makes it infeasible to calculate all outputs at the same time. With the masked logits, we apply a more generalized version of softmax with temperature τ. Intuitively adding the temperature can adjust the peakiness or confidence of the softmax probability distribution output. After the softmax, the matrix M of dimension $V \times N_{ds}$ sums the probability of the N_{ds} datastore entries corresponding to each of the V vocabulary entries. Each column in this matrix consists of a one-hot vector with a value of 1 and the index corresponding to the vocabulary item w_i corresponding to the datastore entry for c_i.

Within this formulation, it becomes obvious that there are many design choices for kNN-LM-like models. One important thing to note is that the right side of Equation 6.2 is actually very similar to the left side representing the standard parametric language model, with a few additional components: M, mask-to-k, and $\otimes$. More specifically, some of the design decisions that go into the kNN-LM, and parallel with standard parametric models are:

Size of W_{ds}: In standard parametric model, the size of W_{sm} is V embeddings, each with D

dimensions. In kNN-LM the size of W_{ds} is very large: N_{ds}, the size of the datastore, usually the number of tokens in the training corpus.

Input representation: In the parametric model, h_{sm} is the output from the feedforward layer in the last transformer block, which we abbreviate "ffn". In contrast, kNN-LM rather use as h_{ds} the output from the multi-headed attention layer of the last transformer block (before running the representations through the feed-forward network, and *after* the LayerNorm [21]), which we abbreviate as "att".

Similarity & Temperature: In the parametric model, the functional form of $\otimes$ is the inner product (abbreviated IP), whereas kNN-LM use negative squared L2 distance (abbreviated L2) as a similarity function between W_{ds} and h_{ds}. As the similarity scores are turned into probability distributions with the softmax function, the choice of softmax temperature (τ) can control the scaling of the similarity scores and thus the peakiness of the non-parametric distribution.

Approximation & Sparsification: In the parametric model, $k = V$, and no values are masked, but in the kNN-LM, $k \ll V$, and most of the datastore entries are pruned out. The definition of the mask-to-k($\cdot$) function, i.e. how to select the important datastore embeddings to include in the similarity calculation (in kNN-LM's case the k nearest neighbors), is a crucial open design choice.

In the following sections, we set out to better understand how each of these design decisions contributes to the improvement in accuracy due to the use of kNN-LMs.

6.3 Baseline kNN-LM Results

First, we evaluate kNN-LM on Wikitext-103 [232], and examine the importance of two design choices: the input representation h_{ds} and the similarity function $\otimes$.

In models examined in this paper, the parametric model is a transformer language model with mostly the same architecture as in Khandelwal et al. [166]. However, we make slight modifications to the original base LM [23] to accommodate our experimentation need. We use BPE tokenization [313] to train a smaller vocabulary (33K) than the original (260K) on the training corpus of Wikitext-103, as subword tokenization is ubiquitous in many state-of-the-art language models [47, 73]. Using subword tokenization also eliminates the need for adaptive softmax [154]. This makes the output layer more general, sharing more resemblance to the kNN component as described in § 6.2, and facilitates the ablation studies in this paper.[1] This base

[1]By re-training the base LM from scratch with BPE tokenization and a standard output softmax, our LM's perplexity is worse than reported by Khandelwal et al. [166]. However, we observe similar relative gains from

LM has 268M parameters. To get a perspective on how large the datastore is, it is built on the training data that contains nearly 150M BPE tokens, each paired with a context vector of size 1024. This datastore has a total memory consumption of about 300GB. Following Khandelwal et al. [166], at every retrieval step, we take the top 1024 nearest neighbors, i.e., $k = 1024$. The interpolated perplexity is computed with optimal interpolation parameter λ tuned according to the perplexity on the development set, and fixed during inference.

	h_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	0	21.750	-	-
kNN-LM-L2	att	L2	$N_{ds} \times D$	∞	19.174	14.230
kNN-LM-IP	att	IP	$N_{ds} \times D$	∞	19.095	14.077
kNN-LM-L2	ffn	L2	$N_{ds} \times D$	∞	20.734	15.594
kNN-LM-IP	ffn	IP	$N_{ds} \times D$	∞	21.101	16.254

Table 6.1: Performance of the parametric language model and several kNN-LM variants.

Results comparing multiple kNN-LM variants are shown in Table 6.1. The first row represents the base parametric language model's perplexity. The second is a formulation analogous to that of kNN-LM, and in the remaining rows, we vary the input representation h_{ds} and distance function $\otimes$ from Equation 6.2. All variants use a large datastore with size N_{ds}, approximately 5000 times the size of the vocabulary V, as also reflected in "+#params", the number of *additional* parameters other than the base LM.

We report several important quantities. "Interp." shows the interpolated perplexity. "PPL" shows the perplexity of *only* the kNN component of the model $p_{\text{kNN}}()$. This is ∞ for all kNN-LM models, as when the kNN search does not retrieve any datastore entries corresponding to the true target word w_t the probability of it will be zero. "Oracle" shows the lower bound of the interpolated perplexity by choosing the best λ for each token in the evaluation dataset, which will either be $\lambda = 0$ or $\lambda = 1$ depending on whether $P_{LM}(w_t|c_t) > P_{knn}(w_t|c_t)$. From the table, we see that:

1. Using the output of the multi-headed attention layer ("att") as h_{ds} (instead of the standard "ffn" layer) is crucial for better performance of kNN-LM.

2. In general, using negative squared L2 distance or inner product as a similarity function does not result in a large and consistent difference, although in our setting, IP provides slightly

the additional kNN component, and we argue that the base LM is orthogonal to the study of the factors behind kNN-LM's improvements.

better performance when using the "att" inputs, and slightly worse when using "ffn" inputs.

3. Interestingly, when using "ffn" and "IP", the same input and distance metric used in the parametric model, the results are the worst, indicating that kNN-LM particularly benefits from a *different view* of the data than the parametric model.

We found in preliminary experiments that kNN-LM is generalizable to other base language models as well, ranging from small models with 82M parameters to larger models with 774M parameters. The gain from kNN-LM is more significant when used with a smaller, less capable base language model (§ 6.8.1) In this paper, we mainly focus on the factors contributing to the *relative* improvements from kNN-LM, instead of the absolute performance, so we use the 268M model for the remainder of the paper. In the next sections, we perform ablation experiments on the general formulation Equation 6.2 to elucidate the key elements contributing to the performance improvements in kNN-LM.

6.4 Effect of Different W_{ds} Formulations

	h_{ds}	N_{ds}	+#params	PPL	Interp.	Oracle
Base LM	-	-	0	21.750	-	-
kNN-LM	att	Big	$N_{ds} \times D$	∞	19.095	14.077
Learned W_{ds}	att	1x	$V \times D$	22.584	20.353	16.954
kNN-LM	ffn	Big	$N_{ds} \times D$	∞	21.101	16.254
Learned W_{ds}	ffn	1x	$V \times D$	20.920	20.694	18.772

Table 6.2: Performance comparison how the choice of h_{ds}, input representation, affects kNN baselines and models with learnable embeddings as datastore alternative. h_{ds} is the attention layer output. $\otimes$ is IP.

6.4.1 Replacing Datastore with Trainable Embeddings

From the observation in § 6.3, we see that the choice of h_{ds} has a large impact on the performance of kNN-LM. This intrigues us to explore if one key to the improvements of kNN-LM lies in the combination of different input representations, namely the attention output (h_{ds} = att) and feedforward output (h_{ds} = ffn). However, based only the experiments above, it is not possible

to disentangle the effect of the choice of h_{ds} and that of other design choices and factors in Equation 6.2.

To test the effect of the choice of h_{ds} in a more controlled setting, we remove the non-parametric datastore entirely, and initialize W_{ds} in Equation 6.2 with a randomly initialized word embedding matrix of the same size ($N_{ds} = V$) as the LM's output embedding W_{sm}, and train W_{ds} with all other parameters fixed.[2] The loss function for training is the cross-entropy loss of softmax($W_{ds} \cdot h_{ds}$) with respect to the ground-truth tokens, identically to how the base LM is trained. We compare how using h_{ds} = att or h_{ds} = ffn affects the interpolated performance. The results are shown in Table 6.2, with the results of kNN-LMs using these two varieties of input representation for reference. From these experiments we find several interesting conclusions:

Effectiveness of re-training $\mathbf{W_{ds}}$: In the case of "Learned W_{ds} w/ FFN", we are essentially re-learning the weights for the softmax function separately from the underlying LM encoder. Despite this fact, the model achieves a PPL of 20.920, which is 0.83 points better than the base model. This suggests that it is beneficial to learn the parameters of W_{ds} after freezing the transformer encoder.

Effectiveness of ensembling two predictors: In both cases of W_{ds}, the interpolated perplexity is significantly better than that of using a single predictor. This is particularly the case when using the "att" representation for h_{ds}, suggesting that the utility of ensembling predictions from two views of the data is not only useful when using kNN-LM, but also in standard parametric models as well.

Parametric ensembles as an alternative to kNN-LM? Overall, by using a separate word embedding matrix with size $V \times D$ as an alternative to kNN, we can recover about 55% of the performance gain achieved by kNN-LM, with only a limited number of parameters and without the necessity for slow kNN retrieval every time a token is predicted. This suggests that the majority of the gain afforded by kNN-LM could be achieved by other more efficient means.

6.4.2 Increasing the Softmax Capacity

One premise behind kNN-LM is that the large datastore is the key reason for the kNN-LM's success: the larger the datastore's capacity, the better the performance. We wonder whether such a big datastore is warranted and whether the size and expressivity of W_{ds} leads to better performance. We test the effect of the datastore size for kNN retrieval on kNN-LM interpolated

[2]Because we previously found little difference between IP and L2 as similarity functions, we use IP in the experiments. For simplicity, we set temperature $\tau = 1$.

perplexity. If a bigger datastore is better in kNN-LM than a smaller datastore, then the hypothesis of softmax capacity is more probable. We randomly subsample the full datastore in varying percentages and the results are shown in the blue "FAISS mask, FAISS score" series in Figure 6.3. The full datastore contains more than 150M entries and storing them takes 293GB when using fp16. We see that the perplexity decreases linearly with a higher fraction of the original datastore. Even with just 5% of the datastore size (15G), kNN-LM still provides a benefit over the base LM. However, even when the subsampling percentage reaches 90%, more entrie in the datastore still provide benefits without having significant diminishing returns, suggesting that a large datastore is beneficial.

One possible reason why a larger datastore is helpful is that some words can be difficult to predict. There are several reasons: (1) They are rare, or (2) they are frequent, but they have multiple meanings and appear in different contexts. The softmax bottleneck [367] suggests that the final dot product of language model $W_{sm} \cdot h_{sm}$ is capped at D rank, limiting the expressiveness of the output probability distributions given the context; that is, a single output vector of a fixed (1024) size cannot express all the possible mappings between 100M training examples and 33K vocabulary outputs. We hypothesize that kNN-LM improves performance by alleviating the problem, since $M \exp(W_{ds} \otimes h_{ds})$ has a higher rank ($M \cdot$ sums softmax outputs of the same token) and is more expressive than just $\exp(W_{sm} \cdot h_{sm})$. kNN is a sparse approximation of the full softmax over all the embeddings in the datastore W_{ds}. To test this hypothesis, we disentangle the effect of W_{ds} size from the actual saved context embeddings in W_{ds}, by training an embedding matrix of the same size from scratch.

We explore several potential solutions for increasing the capacity of softmax, and examine if they can achieve a similar effect to kNN-LM. The first and easiest solution is to increase the embedding matrix size by adding more embedding vectors for each word type in the vocabulary. To test this, we replace W_{ds} with a much smaller matrix of size $nV \times D$, where we allocate n embedding vectors for each word type. When calculating the probability from this component, we compute the softmax over nV items and sum the probabilities for each vocabulary entry. mask-to-k($\cdot$) is no longer needed, as this formulation is small enough to fit the entire matrix in the GPU. We then finetune the new W_{ds} on the training data until convergence.

Figure 6.2 compares the base LM using the original kNN-LM with using either the attention layer output ("att") or the feedforward layer output ("ffn") as h_{ds}. We plot the number of embeddings for each word type (nV total embeddings in W_{ds}) versus the interpolated perplexity, with full details found in Table 6.3.

In both cases, comparing with the top horizontal line which represents the perplexity of

h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
-	-	-	0	21.750	-	-
att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
att	1x	IP	$V \times D$	22.584	20.353	16.954
att	2x	IP	$2V \times D$	21.903	20.529	17.432
att	3x	IP	$3V \times D$	22.434	20.395	17.132
att	4x	IP	$4V \times D$	21.936	20.521	17.423
att	5x	IP	$5V \times D$	22.025	20.643	17.560
att	6x	IP	$6V \times D$	21.972	20.519	17.422
att	9x	IP	$9V \times D$	22.084	20.696	17.631
ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
ffn	1x	IP	$V \times D$	20.920	20.694	18.772
ffn	2x	IP	$2V \times D$	20.889	20.646	18.701
ffn	3x	IP	$3V \times D$	20.829	20.603	18.717
ffn	4x	IP	$4V \times D$	20.769	20.629	18.876
ffn	5x	IP	$5V \times D$	20.720	20.594	18.878
ffn	6x	IP	$6V \times D$	20.726	20.599	18.902
ffn	9x	IP	$9V \times D$	20.687	20.567	18.887

Table 6.3: Performance comparison of kNN baselines and models with learnable embeddings of increasing size as W_{ds} datastore alternative. h_{ds} is either attention layer output (att) or feedforward layer output (ffn).

the base LM, replacing the datastore with a much smaller weight matrix (from N_{ds} to nV_{ds}) by assigning only a few more embeddings for each word helps, although only about half as effective as kNN-LM. To give a perspective, the original datastore size is about $5000V$. Surprisingly, we find that increasing n does not always bring better performance, even though a larger datastore is better than using a small datastore in kNN-LM. We see that when h_{ds} = ffn, over-parameterization provides limited improvements, while for h_{ds} = att it does not bring consistent improvements at all. Comparing the trend of increasing the embeddings in W_{ds}, with the bottom horizontal line in the plot, which represents the perplexity of the standard kNN-LM using the full datastore (W_{ds} with approx. $5000V$ embeddings), we see no clear trend that more trainable embeddings result in better perplexity, and that the gap between using trained embeddings and using full datastore is still significant. This suggests that simply over-parameterizing W_{ds} is not

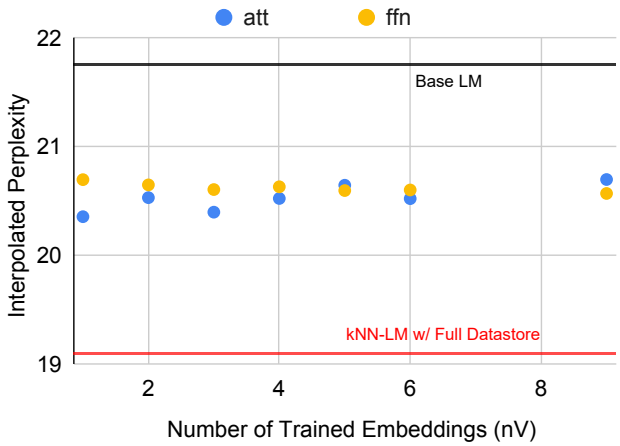


Figure 6.2: The number of embeddings per word type (nV total embeddings in W_{ds}) versus interpolated perplexity, compared with base LM and kNN-LM.

an effective method of achieving gains similar to kNN-LM.

We hypothesize that this is because by just adding more embeddings, while still using the same training procedure as the original LM, the multiple embeddings for each word type after learning could still be very close to each other, and thus do not increase the softmax capacity much. This suggests that some regularization terms may be needed during training to make the multiple embeddings not converge to the same vector, rendering over-parameterization useless.

Besides simply increasing the number of embedding vectors equally for each word type, we also propose other alternatives to increase softmax capacity. First, we hypothesize that different word types have different difficulties for the language model to predict. For those words that appear very frequently, they may appear in many different contexts. As a result, instead of adding an equal number of additional embeddings to each word type, we propose to adaptively increase the number of embeddings for word types based on word frequency, or total training loss for the word. Second, we try to break the softmax bottleneck. Yang et al. [367] proposes a solution using Mixture of Softmax (MoS) to produce more linearly independent probability distributions of words given different contexts. Last, instead of training word embeddings of increased size, we also consider compressing the datastore down to a similar-sized embedding matrix for softmax by clustering the datastore and finetuning of the matrix consisting of cluster centroids. However, none of these alternative methods provided additional benefits over the simple multi-embedding approach (§ 6.8.2).

	PPL	Interp.	Oracle
Base LM	21.750	-	-
kNN-LM w/ FAISS mask, FAISS score	∞	19.174	14.230
kNN-LM w/ FAISS mask, real score	∞	19.672	14.393
kNN-LM w/ real mask, real score	∞	19.735	14.480

Table 6.4: Performance of the parametric language model and comparison of kNN-LMs using the approximate versus ground truth kNN. $\otimes$ is L2. h_{ds} = att.

6.5 Approximate kNN & Softmax Temperature

6.5.1 Comparing Approximate kNN Search

To calculate P_{kNN} of the non-parametric component in Equation 6.2, it is usually prohibitive to use exhaustive kNN search, and thus Khandelwal et al. [164] use approximate kNN search using the FAISS library [150]. The use of FAISS (similarly to other approximate search libraries) results in two varieties of approximation.

Approximate Neighbors: Because the search for nearest neighbors is not exact, the set of nearest neighbors might not be equivalent to the actual nearest neighbors. Recall that the function mask-to-k$(\cdot)$ in Equation 6.2 is the function that selects kNN entries from the datastore W_{ds}. We denote "real mask" as the accurate nearest neighbors for mask-to-k$(\cdot)$ selection, and "FAISS mask" as the approximate nearest neighbors returned by the FAISS library.

Approximate Scores: In addition, FAISS makes some approximations in calculating the distances between the query and the retrieved neighbors for efficiency purposes. We denote "real score" as the scores calculated from ground truth distances between the embeddings, and "FAISS score" as the distances returned by FAISS approximate search.

The comparison of the different approximation settings is shown in Table 6.4. Quite surprisingly, we actually find that the interpolated perplexity with approximate search is *better* than that with exact search, both with respect to the mask and the score calculation. Intrigued by this counter-intuitive result, we explore the effect of kNN search approximation.

First, we plot the subsampled size of the datastore with the interpolated perplexity Figure 6.3, but showcasing the comparison between approximate and real masks, approximate and real scores in both the full datastore as well as a small subsampled datastore setting. We find that using an *approximate* FAISS mask to find nearest neighbors performs better than using the

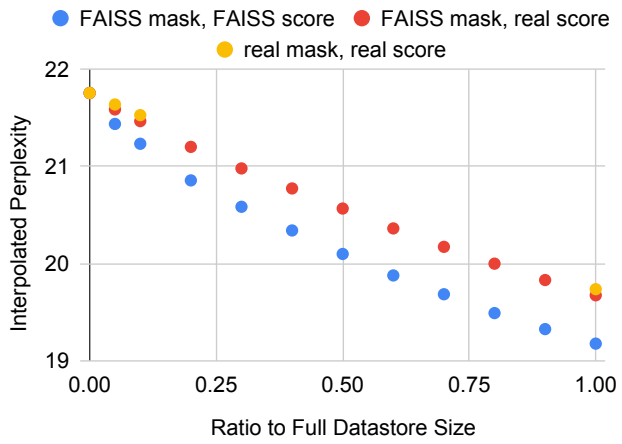


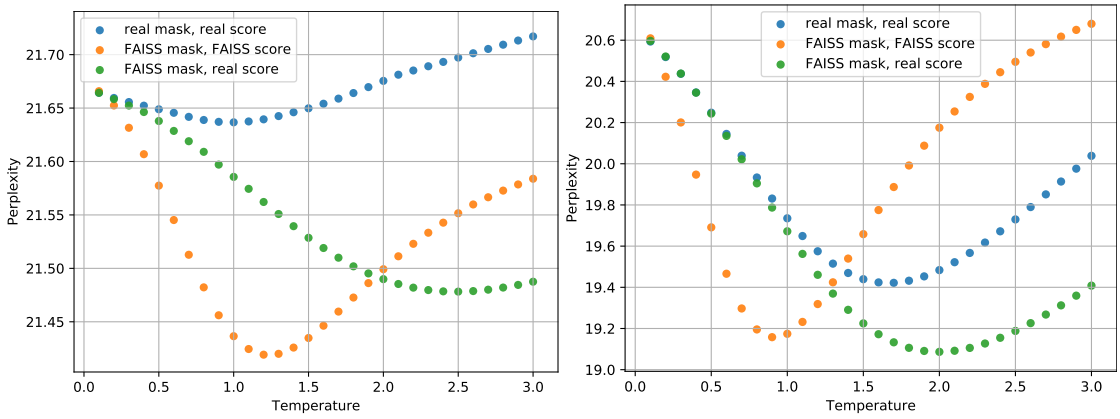
Figure 6.3: The differences between using approximate and accurate kNN search on varying sizes of the datastore.

exact nearest neighbors *both* at 5% and 100% of the datastore. However, using the approximate score returned by FAISS is better than recomputing the exact distances between embeddings for the kNN distribution *only* for the small 5% datastore scenario. Interestingly, the gap between using an approximate score or real score given the same approximate neighbors ("FAISS mask, FAISS score" vs. "FAISS mask, real score") is larger than that between using approximate or real neighbors given the same ground truth method of calculating the distance ("real mask, real score" vs. "FAISS mask, real score").

We hypothesize that this is related to regularization for preventing overfitting, and approximate search provides fuzziness that functions as a regularizer. We can think of the kNN component of kNN-LM as a model, where the datastore size is the model capacity, and the datastore is its training data. Considering that the kNN component uses the exact same training data as the base parametric LM, having ground truth, accurate kNN search may cause the kNN component to overfit the training data.

6.5.2 Adding Softmax Temperature to kNN Distribution

Because the number of retrieved nearest neighbors, k, is usually much smaller than the vocabulary size V, intuitively, the kNN distribution P_{kNN} used for interpolation tends to be more peaky than the standard LM output distribution. When $k = 1024$ and $V = 33000$, as in our experiments, P_{kNN} will only have a few vocabulary items with a non-zero probability. Furthermore, many of the retrieved neighbors share the same target token and thus make the kNN distribution even peakier. One way to control the entropy, or peakiness of the distribution is to add tempera-



(a) On 5% subsampled datastore.

(b) On full datastore.

Figure 6.4: The interpolated perplexity varies with different softmax temperature τ values.

ture to the distances that go into the softmax function [135]. We calculate the probability of non-parametric component in Equation 6.2 where τ is the softmax temperature. In general, the higher the temperature, the less "peaky" the distribution becomes. We experiment with both the 5% as well as the full datastore using different temperatures ranging from 0 to 3 at 0.1 intervals. The results are shown in Figure 6.4a and Figure 6.4b respectively.

We see that the default temperature $\tau = 1$ does not always result in the best-interpolated perplexity and tuning the softmax temperature is desirable for all sizes of datastore. The lesson learned here is that tuning the softmax temperature for the kNN distribution is crucial for getting optimal results from each setting. Only coincidentally, a temperature of 1.0 was close to optimal in the original settings of kNN-LM, which hid the importance of this hyperparameter. Even at the optimal temperature of each setting, "real mask, real score" underperforms "FAISS mask, real score". This is consistent with the counter-intuitive phenomenon in § 6.5.1. There are also differences between different datastore sizes. With the full datastore, using "real score" outperforms "FAISS score" given the same "FAISS mask". However, the opposite is true when using the 5% datastore. This suggests that as the datastore size grows, using accurate distance values are better than the approximate ones. The smaller gap between using "real score" and "FAISS score" in both datastore settings shows that the main contributor to the improvements is using approximate nearest neighbors ("FAISS mask") rather than using approximate distance values ("FAISS score").

These results emphasize the effect of approximation discussed in § 6.5.1, because comparing the small datastore with only 5% with the original datastore, we see that a small datastore means a small training set for the kNN "model" and it thus it benefits more from this regularization,

both by using the FAISS mask and FAISS score (at optimal temperature settings). Surprisingly, one of the important ingredients in kNN-LM seems to be *approximate* kNN search, which likely prevents overfitting to the datastore created from the same training set. We further analyze this unexpected result in § 6.8.3, where we find that longer words and words that appear in many different contexts have slightly better results with approximate nearest neighbors.

Consistently with our findings, He et al. [121] found that dimensionality reduction using PCA on the datastore vectors (from 1024 to 512 dimensions) improves the perplexity of the original kNN-LM from 16.46 to 16.25, which can be explained by our findings as PCA may provide another source of approximation that contributes to regularization. Notably, similar effects, where an approximation component leads to better generalization, have been reported in other NLP tasks as well, and are sometimes referred to as "beneficial search bias", when modeling errors cause the highest-scoring solution to be incorrect: for example, Meister et al. [231] suggest that "quite surprisingly, beam search often returns better results than exact inference due to beneficial search bias for NLP tasks"; Stahlberg and Byrne [332] also conclude that "vanilla NMT in its current form requires just the right amount of beam search errors, which, from a modeling perspective, is a highly unsatisfactory conclusion indeed, as the model often prefers an empty translation".

6.6 Probably Wrong Hypotheses for Why kNN-LM Works

The results in the previous sections are the result of extensive analysis and experimentation, in which we also tested a number of hypotheses that did *not* turn out to have a significant effect. Additional details of these hypotheses are detailed in the following sections, and we hope that they may provide ideas for future improvements of retrieval-based LMs.

Ensemble of Distance Metrics We hypothesized that the ensemble of two distance metrics: the standard inner product distance (which the LM uses) and the L2 distance (which the kNN component uses), is the key to the improvement. However, we found that similar gains can be achieved using the inner-product metric for the retrieved kNN (§ 6.6.1).

Ensembling of Two Models We hypothesized that the kNN component merely provides another model for ensembling. The improvement from kNN-LM is *purely* due to the ensembling effect of simply different models. However, we found that kNN-LM's improvement is orthogonal to ensembling with a different base LM (§ 6.6.5).

Sparsification The mask-to-k$(\cdot)$ used by kNN retrieval induces sparsity in the distribution over the vocabulary, due to a small k (typically 1024) compared to the size of the vocabulary

V (33K in our experiments and 260K in the original setting). We hypothesized that kNN-LM increases the probability of the top-k entries while taking "probability mass" from the long tail of unlikely word types. However, we could not gain any benefits solely from sparsifying the output probability of a standard LM and interpolating it with the original LM (§ 6.6.2).

Stolen Probabilities The *stolen probabilities* effect [70] refers to the situation where the output embeddings of an LM are learned such that some words are geometrically placed *inside* the convex hull that is formed by other word embeddings and can thus never be "selected" as the argmax word. We hypothesized that kNN-LM solves the stolen probabilities problem by allowing to assign the highest probability to *any* word, given a test context that is close enough to that word's datastore key. However, we found that *none* of the vectors in our embedding matrix and in the original embedding matrix of Khandelwal et al. [166] is located in the convex hull of the others, which is consistent with the findings of Grivas et al. [102] (§ 6.6.4).

Memorization We hypothesized that the kNN component simply provides memorization of the training set. However, we could not improve a standard LM by interpolating its probability with another standard LM that was further trained to overfit the training set (§ 6.6.6).

Soft Labels We hypothesized that kNN-LM's improvement lies in reducing the "over-correction" error when training with 1-hot labels, as hypothesized by Yang et al. [370], and that retrieving neighbors is not important. If only "soft labels" are the key, we could hypothetically improve the performance of another fresh LM with the same model architecture but trained with the soft labels from the base LM, instead of from kNN-LM. This separates the effect of "soft labeling" from the additional guidance provided by kNN. However, this did not help at all (§ 6.6.7).

Optimizing Interpolated Loss We hypothesized that the standard LM cross-entropy training loss does not emphasize the examples where base LM performs badly which could benefit from kNN, and directly optimizing the interpolated loss of standard LM and a separate trainable softmax layer could be a better alternative. However, we could not gain any benefits by training an additional softmax layer together with a base LM using the interpolated loss (§ 6.6.8).

6.6.1 Distance Metric

We hypothesize that the key to kNN-LM's performance gain is the ensemble of two distance metrics: the standard dot product distance (which the LM uses) with the L2 distance (which the kNN component uses as $\otimes$). We tried to replace the kNN component with a component that just takes the tokens retrieved by the kNN search and returns their L2 distance to the LM output

word embeddings: $W_{sm} \otimes h_{ds}$ instead of $W_{ds} \otimes h_{ds}$, where $\otimes$ represents the negative L2 distance. We tried this with both variants of h_{ds}, attention layer output, and feedforward layer output. None of these helped.

6.6.2 Sparsification

In Equation 6.2, mask-to-k$(\cdot)$ used by kNN retrieval induces sparsity in the distribution over the vocabulary, due to a small k compared to the number of vocabulary V. We hypothesize that the in kNN-LM, the kNN distribution is sparse, practically increasing the probability of the top-k entries. The kNN distribution has up to 1024 entries that are non-zero, concentrating more probability mass over the most likely tokens. This effect is similar to the redistribution of probability mass for text generation in [135]. We test this hypothesis only by taking top 32, 64, 128, 512, or 1024 tokens in the parametric LM probability and zeroing out the probabilities of the rest of the tokens. To compensate, we experiment with different softmax temperatures and then interpolate with the parametric LM probability. This isolates the effect of the datastore and retrieval at all, and this does not help at all, suggesting that sparsification of the output probability alone is not enough.

Another attempt is to hypothesize that the key in kNN-LM is that it selects "which tokens to include" in the kNN distribution, and not their distances. The intuition behind is that maybe the selection of the top tokens according to the kNN search is better than that from the dot-product distance between the language model's output vector and all the vocabulary embeddings. We perform experiments similar to the previous attempt, sparsifying the output probability with the tokens retrieved by the kNN search (but ignoring the distances provided by the kNN search) rather than the top k tokens of the LM, with and without removing duplicates. In the best case, they manage to reduce the perplexity by 0.5 (whereas kNN-LM reduces by nearly 2).

6.6.3 Location within Context Window

Supposedly, words in the beginning of the "context window" of the transformer at test time have less contextual information than words toward the end of context window.

We hypothesized that maybe the base LM performs worse in one of these (beginning vs. end of the context window), and maybe kNN-LM provides a higher improvement in one of these. We measured the per-token test perplexity with respect to the location of each token in the context window. However, we did not find any significant correlation between the performance

of the base LM and the location, and no significant correlation between the difference between kNN-LM and the base LM and the location.

We also hypothesized that maybe the beginning of every Wikipedia article is more "predictable", and the text becomes more difficult to predict as the article goes into details. However, we also did not find any correlation with the location of the word within the *document* it appears in.

6.6.4 Stolen Probabilities

The *stolen probabilities* effect [70] refers to the situation where the output embeddings of an LM are learned such that some words are geometrically placed *inside* the convex hull that is formed by other word embeddings. Since language models generate a score for every output word by computing the dot product of a hidden state with all word embeddings, Demeter et al. [70] prove that in such a case, it is impossible for words inside the convex hull to be predicted as the LM's most probable word (the "argmax").

We hypothesized that kNN-LM solves the stolen probabilities problem by allowing to assign the highest probability to *any* word, given a test hidden state that is close enough to that word's datastore key. Nevertheless, as shown by Grivas et al. [102], although this problem might happen in small RNN-based language models, in modern transformers it rarely happens in practice. Using the code of Grivas et al. [102], we checked the embeddings matrix of our model and of the checkpoint provided by Khandelwal et al. [166]. Indeed, we found that in both models – *no word is un-argmaxable*.

6.6.5 Are kNN-LM Just Ensembling?

Our hypothesis is that kNN component only provides another model for ensembling. The interpolation process is basically an ensemble model. Technically it is unsurprising that kNN-LM will have the benefit from ensembling, but we perform experiments to see how it compares to other ensembling. We trained another language model with the same architecture as the base LM we used throughout the experiments, with some variants having more than one embedding vector for each word (similar to § 6.4.2). We interpolate the models with the original base LM, and the results are shown in Table 6.5. We see that even just ensembling the base LM with another identical model, but trained with a different random seed, provides a huge performance boost, both on interpreted perplexity and on oracle perplexity.

Prev. Layers	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
same	-	-	-	0	21.750	-	-
same	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
same	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
same	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
same	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
diff	ffn	1x	IP	$F + V \times D$	21.569	18.941	14.980
diff	ffn	2x	IP	$F + 2V \times D$	21.914	18.948	14.885
diff	ffn	3x	IP	$F + 3V \times D$	22.206	18.981	14.853

Table 6.5: Performance comparison of kNN baselines and models with different size output embeddings re-trained from scratch.

However, just because ensembling two LMs of the same architecture provides better performance than interpolating the base LM with kNN does not necessarily suggest that kNN's performance improvement can be fully replaced by model ensembling. In other words, we are interested in whether the kNN performance improvements are orthogonal to that of model ensembling. To test this, we compare the performance of the ensemble of K multiple LMs versus the ensemble of $K-1$ multiple LMs plus the kNN component. The comparison is fair because we have the same number of models in the ensemble, and the only difference is whether the kNN component is included. The results are shown in Figure 6.5. For the "LM" series, each point is K LMs ensemble, and for the "kNN" series, each point is $K-1$ LMs plus kNN. We see that even at 4-ensemble, the ensemble that contain kNN as a component still have a considerable edge over the 4-ensemble that contain just LMs.

6.6.6 Are kNN-LM Just Overfitting?

Since kNN-LM improves perplexity even with the same training dataset as datastore, we are curious if kNN-LM works by only "memorizing" the training data. The hypothesis is that the datastore and the kNN search are trying to memorize the training data. In other words, the parametric LM is under-fitting some tokens. The intuition behind this is that the kNN component retrieves examples directly from the training set. What if we could retrieve the same examples using an overfitted LM? We took the trained LM, removed the dropout, and continued training until almost perfect fit (very small training loss). We then interpolated the overfitted transformer

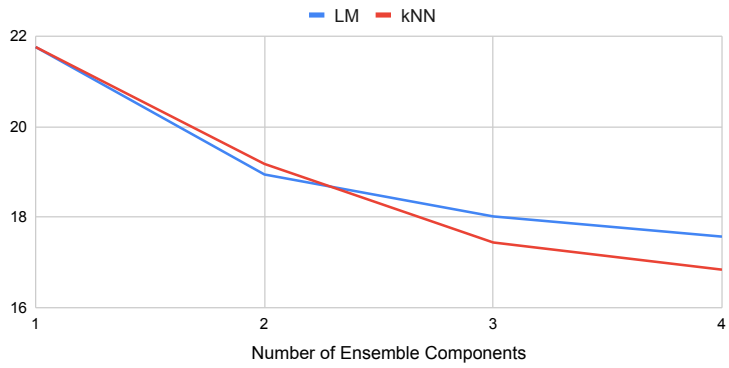


Figure 6.5: Ensembling effect comparison, between multiple base LMs and multiple base LMs plus kNN component.

	Prev. Layers	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	same	-	-	-	0	21.750	-	-
kNN-LM	same	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
kNN-LM	same	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
kNN-LM	same	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
kNN-LM	same	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
Overfit@92	diff	ffn	V	IP	$F + V \times D$	1702.806	21.732	17.764
Overfit@129	diff	ffn	V	IP	$F + V \times D$	8966.508	21.733	17.814

Table 6.6: Performance comparison of several baselines with two overfitted models, at 92 and 129 additional epochs.

with the original LM. The results are shown in Table 6.6. F represents the number of parameters in the base LM, minus the output embedding matrix. We see that overfitting can provide very little help after interpolation. Looking at the oracle performance, we think that the overfitted model memorizes some rare contexts and tokens in the training set where it could be useful during evaluation. However, the overfitting hurts the performance on other tokens too much so that even interpolation is not able to balance the performance.

6.6.7 Are kNN-LM Just Soft-Label Training?

[370] claims that using "soft labels" during training is the key to kNN's success, that interpolates the ground truth labels with kNN-LM model outputs, effectively "distilling" kNN-LM. It is based

on the hypothesis that the room for kNN-LM's improvement over base LM lies in the "over-correction" when training with a 1-hot labels. This is related to the effect from label smoothing methods [230, 270, 337]. However, we believe that this explanation is not satisfactory. If the key is training with soft-labels, why do these soft labels must be provided specifically by a kNN search? If soft labels were the key, then soft-label training where the labels come from the base LM itself should have worked as well. To separate the effect of soft labeling from the kNN's additional guidance, we train another LM with the same model architecture as the base LM, with the soft labels from the base LM. This teacher-student training is to distill the knowledge from the base LM [132]. We find that by just training with "soft labels" from the base LM to alleviate the alleged "over-correction" problem is not the key, as this does not help with the interpolated perplexity at all. This suggests that even with the same training data, kNN still provides valuable additional guidance.

6.6.8 Are kNN-LM Just Training to Optimize Interpolated Loss?

In § 6.4.2, we discover that using over-parameterization with standard LM training loss does not further close the gap towards kNN-LM. This suggests that some regularization term may be needed during training to make the multiple embeddings not converge to the same vector, rendering over-parameterization useless.

From Table 6.2, we see that a better interpolated perplexity may not require a very low perplexity when measured only with the extra input representation. However, we still use a standard LM loss to only train the additional embedding matrix, that directly minimizes the perplexity using only the extra input representation. This discrepancy between training and the evaluation with interpolation suggests that training with an alternative loss function that interpolates the base LM's output with the output using the extra input representation may be beneficial.

To test the hypothesis that standard LM training loss do not emphasize the examples where base LM performs badly, we train the extra model's parameter W_{ds}, with interpolated loss L:

$$L = \text{CrossEntropy}(\lambda \text{softmax}(W_{ds} \cdot h_{ds}) + (1 - \lambda)\text{softmax}(W_{sm} \cdot h_{sm}), y) \tag{6.3}$$

y represents the ground truth label for each context. We only learn the parameter W_{ds} while freezing all other parameters, similar to all other experiments. We choose $\lambda = 0.25$ as it is the best hyper-parameter for kNN-LM experiments and our goal for this training is to mimic the loss of kNN-LM after interpolation. This training loss effectively assigns a higher value to the

training examples where the base LM's loss is high, suggesting the need for the extra W_{ds} to help with these hard cases. However, for either "att" for "ffn" for h_{ds}, either V or $3V$ for the number of embeddings in W_{ds}, we are unable to achieve a better perplexity than just the base LM. This suggests that, while nice on paper, the interpolated loss optimization process is not trivial.

6.7 Conclusion

In this paper, we investigate why kNN-LM improves perplexity, even when retrieving examples from the same training data that the base LM was trained on. By proposing and testing various hypotheses and performing extensive ablation studies, we find that the key to kNN-LM's success is threefold: (1) Ensembling different input representations – the feedforward layer output and the attention layer output – can recover 55% of the performance, even without retrieval. (2) One of the most unexpected discoveries is that using *approximate* nearest neighbor search allows kNN-LMs to generalize better than exact nearest neighbor search, possibly due to a regularization effect. (3) Tuning the softmax temperature for the kNN distribution is crucial to adjust the standard LM output distribution with the distribution created by the retrieved neighbors' distances. These findings are orthogonal to Drozdov et al. [78] where they discovered kNN-LM works especially well when there is a large n-gram overlap between the training and the test set.

We performed extensive experiments which ruled out other hypotheses as to why kNN-LMs work, such as over-parameterization, sparsification, overfitting, ensembling of distance metrics, etc. We believe that this work unlocks a variety of exciting research directions for efficient kNN-LM alternatives in addition to existing improvement models [406]. For example, exploring methods that replace the kNN component with trainable parameters and achieve comparable results without the latency burden of kNN-LM.

6.8 Appendix

6.8.1 kNN-LM Generalization to Other LMs

To test the generalizability of kNN-LM, we follow the same experimental setup as used in § 6.3. We select several pretrained models from the GPT2 family [285] of various parameter counts, plus a distilled version of GPT2, DistillGPT2. [311] We take the pretrained model checkpoint,

	#params	Base LM PPL	kNN-LM PPL	Absolute PPL Gain
Ours	268M	21.75	19.17	2.58
Distilled-GPT2	82M	18.25	14.84	3.41
GPT2-small	117M	14.84	12.55	2.29
GPT2-medium	345M	11.55	10.37	1.18
GPT2-large	774M	10.56	9.76	0.80

Table 6.7: Performance of kNN-LM applied to other pretrained language models of different sizes.

build the datastore and evaluate on the Wikitext-103 dataset splits. The results are shown in Table 6.7. We see that kNN-LMs has good generalizability on other models. It improves the perplexity of all the base LMs tested. However, the larger the model is, and usually the better the base LM's perplexity is, the less gain can be achieved from adding kNN. Note that our model is trained from scratch on Wikitext-103 dataset and thus even with a relatively large model size, the perplexity and perplexity gain from adding kNN is still less than models with pretraining. Without loss of generalizability, we will use our own trained-from-scratch model as the base LM in the following sections for ablation study.

6.8.2 Alternative Methods for Increasing Softmax Capacity

Adaptive Increasing Embedding Size

We hypothesize that different word types have different difficulties for the language model to predict. For those words that appear very frequently, they may appear in many different contexts. As a result, instead of adding equal number of additional embeddings to each word type, we propose to adaptively increase the number of embeddings for word types based on word frequency, or total training loss for the word. Based on the intuition of Zipf's law [61], we assign $1 + \log_b f_v$ for each word type $v \in V$, based on either the frequency or the total training loss of the word, f_v. The b is a hyperparameter that could be tuned. To ensure fair comparison, we tune b so that for each experiment the total number of embeddings matches: $\sum_{v \in V} 1 + \log_b f_v = nV$. The results are shown in Table 6.8. We see that although nice on paper, given the same number of total embeddings, adaptively increasing the number of embeddings assigned for each word type does not make a significant difference in the final perplexity, when compared with the

models that use equal number of embeddings for each word type.

	h_{ds}	N_{ds}	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	-	0	21.750	-	-
KNN	att	Big	L2	$N_{ds} \times D$	∞	19.174	14.230
KNN	att	Big	IP	$N_{ds} \times D$	∞	19.095	14.077
Equal Per Word	att	3x	IP	$3V \times D$	22.434	20.395	17.132
Loss Weighted	att	3x	IP	$3V \times D$	21.948	20.440	17.303
Freq. Weighted	att	3x	IP	$3V \times D$	22.507	20.387	17.105
KNN	ffn	Big	L2	$N_{ds} \times D$	∞	20.734	15.594
KNN	ffn	Big	IP	$N_{ds} \times D$	∞	21.101	16.254
Equal Per Word	ffn	3x	IP	$3V \times D$	20.829	20.603	18.717
Loss Weighted	ffn	3x	IP	$3V \times D$	20.764	20.659	18.978
Freq. Weighted	ffn	3x	IP	$3V \times D$	20.757	20.572	18.782

Table 6.8: Performance comparison of kNN baselines and several configurations that adaptively increase the embedding size with training loss or word frequency.

Mixture of Softmaxes

[367] proposes a solution to the problem using a Mixture of Softmax (MoS) to produce more linearly independent probability distributions of words given different contexts. Suppose that there are a total of R mixture components. MoS first uses R linear layers with weight w_r to transform the current query context vector h_{ds} into $w_r h_{ds}$. With a shared word embedding matrix W_{sm}, we calculate each softmax component's probability distribution with $\text{softmax}(W_{sm} \cdot w_r h_{ds})$. The mixture distribution is then given by:

$$P_{MoS} = \sum_{r}^{R} \pi_{r,h_{ds}} \text{softmax}(W_{sm} \cdot w_r h_{ds}) \tag{6.4}$$

The prior weights are calculated using another linear layer with weight w_π, as $\pi_{r,h_{ds}} = \text{softmax}(w_\pi h_{ds})$. The softmax ensures that $\sum_r^R \pi_{r,h_{ds}} = 1$. Comparing the MoS with the first term in Equation 6.2, $M\text{softmax}(\text{mask-to-k}(W_{ds} \otimes h_{ds}))$, we see that there are some connections between the two. MoS eliminates the mask-to-k$(\cdot)$ operation, and replaces the single softmax across a very large vector (size of datastore), into multiple smaller softmaxes, each across only a vector of the size of vocabulary. As a result, the huge W_{ds} is replaced by several linear layers to project the word

embedding matrix. Now the first term becomes:

$$M(\oplus_r^R \text{softmax}(W_{sm} \cdot w_r h_{ds})) \tag{6.5}$$

$$M_{ir} = \pi_{r,h_{ds}}, \forall i \leq V \tag{6.6}$$

where $\oplus$ represents the vector concatenation operation, and the aggregation matrix M now contains the mixture weights for each softmax being concatenated. We perform experiments with a varying number of mixtures (R), different definitions h_{ds}, and whether to finetune the output word embeddings W_{sm}. We allow finetuning the word embedding when we use attention layer output as context vector, since the word embedding matrix is trained with feedforward layer output originally. The results for this formulation are shown in Table 6.9. MoS models on its own increase the performance of the language model marginally. When compared with Table 6.3, we find that these models are worse than those that simply increases the number of embeddings. This is expected because MoS has fewer added parameters compared to those, as it only requires several additional linear projection layers for the embeddings.

	h_{ds}	R	$\otimes$	+#params	PPL	Interp.	Oracle
Base LM	-	-	-	0	21.750	-	-
KNN	att	-	L2	$N_{ds} \times D$	∞	19.174	14.230
KNN	att	-	IP	$N_{ds} \times D$	∞	19.095	14.077
KNN	ffn	-	L2	$N_{ds} \times D$	∞	20.734	15.594
KNN	ffn	-	IP	$N_{ds} \times D$	∞	21.101	16.254
Ft. MoS+embed	att	2	IP	$VD + 2D^2 + 2D$	21.986	20.720	17.573
Ft. MoS+embed	att	3	IP	$VD + 3D^2 + 3D$	22.106	20.779	17.609
Ft. MoS Only	att	2	IP	$2D^2 + 2D$	22.552	21.011	17.796
Ft. MoS Only	att	3	IP	$3D^2 + 3D$	22.573	21.024	17.812
Ft. MoS Only	ffn	2	IP	$2D^2 + 2D$	21.351	21.338	20.258
Ft. MoS Only	ffn	3	IP	$3D^2 + 3D$	21.495	21.460	20.322
Ft. MoS Only	ffn	4	IP	$4D^2 + 4D$	21.321	21.321	20.396
Ft. MoS Only	ffn	5	IP	$5D^2 + 5D$	21.371	21.367	20.406

Table 6.9: Performance comparison of kNN baselines and several MoS configurations. R is the number of mixtures.

Clustering Datastore

Opposite to training the word embeddings of an increased size, we also consider ways to compress the datastore down to a similar-sized embedding matrix for softmax computation. The intuition is that the datastore contains redundant context vectors, and thus compression could make the datastore smaller without sacrificing too much performance gain. [121] shows that we can safely compress the datastore by clustering to 50% of the original size without losing performance. We test this idea further by clustering the entire datastore into a size that could fit in GPU memory (e.g. $2V$, $3V$) and thus could be easily finetuned further and use the resulting centroids to replace W_{ds}. Within each cluster, there will be a distribution of different words with contexts, and we use the frequency of words within each cluster to calculate the aggregation matrix M in Equation 6.2. This would have the added benefit of "multi-sense" embedding, which allows similar meanings to be clustered to form a new "meta word" while the same word with different meanings would form different "meta words". A notable example is bank, shore, and financial institution. However, this does not work, mostly because of the high compression loss after clustering and the imbalanced distribution of word types among each cluster.

6.8.3 Which Words Benefit from Approximation?

To further understand the unexpected results when using the different kNN approximate retrieval settings in § 6.5.1 and § 6.5.2, we analyze on a token level, based on how many times each ground truth token's probability in the evaluation set are helped by each kNN setting. It means that for each ground truth token in the evaluation, we count the times when the kNN distribution is higher than the base LM distribution P_{LM}, i.e., $P_{kNN} > P_{LM}$.

Since we found previously that approximate kNN provides an additional performance boost compared to ground truth kNN, we thus compare "real mask, real score" versus "FAISS mask, real score" in this analysis. To prevent outliers, we filter out words with less than 10 occurrences in the evaluation set. For each setting, we calculate the percentage of occurrences in the evaluation set where each token in the vocabulary where the kNN module achieves a better probability than base LM. We then plot the absolute difference between the percentages of the two settings, with respect to various possible attributes of the token that achieves better probability using each setting.

Figure 6.6 shows that the longer the token is, which usually suggests proper nouns and harder and less common words in English, are better with approximate neighbors than ground truth ones, and vice versa. We hypothesize that this is due to longer words are more prone to

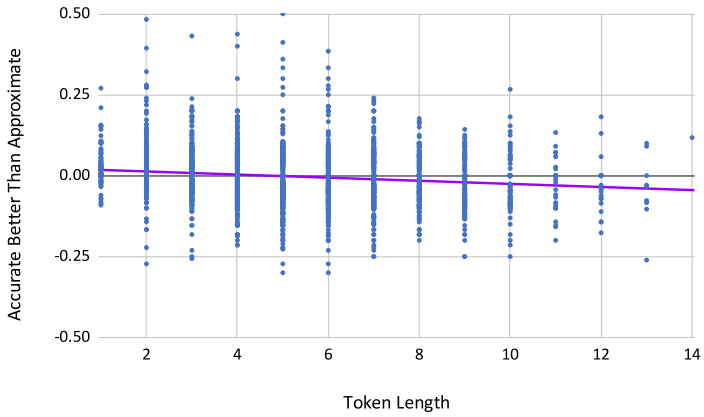


Figure 6.6: The effect of the token character length on how much accurate nearest neighbors are better than approximate FAISS neighbors. Negative values mean worse. The trend line of the scatter points is shown.

overfitting in kNN-LM and thus using approximate kNN provides an effect similar to smoothing and regularization.

We also compare words that could appear in more diverse contexts with words that co-occur with few distinct contexts. To measure how diverse the contexts of each word in the vocabulary is, we calculate both the forward and backward bigram entropy for each word in the evaluation set that has more than 10 occurrences. The bigram entropy is a simple yet good indicator of context diversity for a given word, as used in Kneser–Ney smoothing [251]. We calculate both the forward and backward bigram entropy for each word w as follows, where w_{after} and w_{before} represent the word after and before the given word w.

$$H_{\text{forward}}(w) = -\sum_{w_{\text{after}}} p(w_{\text{after}}|w) \log p(w_{\text{after}}|w) \tag{6.7}$$

$$H_{\text{backward}}(w) = -\sum_{w_{\text{before}}} p(w_{\text{before}}|w) \log p(w_{\text{before}}|w) \tag{6.8}$$

Forward and backward entropy represents how diverse the context after and before the given word is. Intuitively, bigram entropy is supposed to indicate words that can appear in lots of different contexts. The higher the entropy of a word, the more diverse its context is, and vice versa. For example, words like “Francisco” would have a low entropy because it mostly comes after “San”.

The comparison is shown in Figure 6.7. We see that the higher the entropy in both forward and backward cases, the better using approximate nearest neighbor search becomes. This suggests that words that appear in many different contexts are better off with an approximate

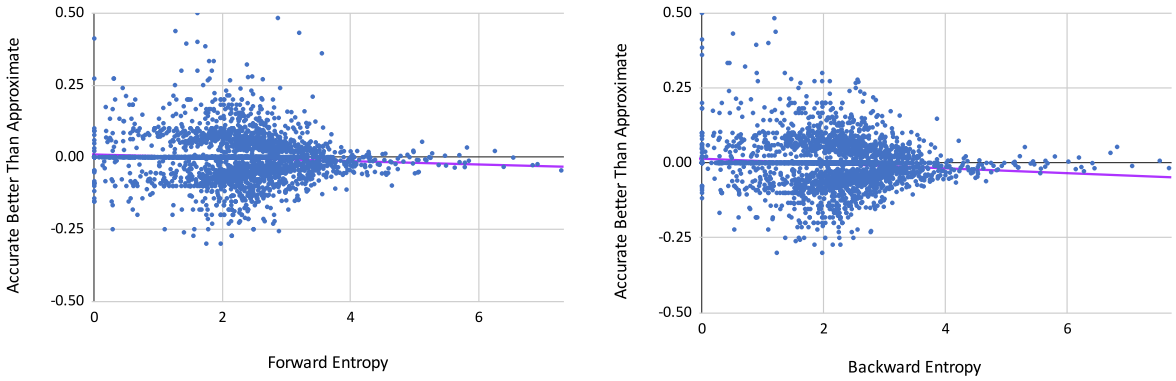


Figure 6.7: The effect of the forward and backward entropy of words on how accurate nearest neighbors are better than approximate FAISS neighbors. Negative values mean worse. The trend line of the scatter points are shown.

kNN, and "easy-to-predict" examples such as "Jersey" and "Fransisco" is better with accurate kNN, possibly because these examples are less prone to overfitting errors and thus requires less regularization from approximation.

Chapter 7

DocPrompting: Generating Code by Retrieving the Docs (Completed)

Publicly available source-code libraries are continuously growing and changing. This makes it impossible for models of code to keep current with all available APIs by simply training these models on existing code repositories. Thus, existing models *inherently cannot generalize* to using unseen functions and libraries, because these would never appear in their training data. In contrast, when human programmers use functions and libraries for the first time, they frequently refer to textual resources such as code manuals and documentation, to explore and understand the available functionality. Inspired by this observation, we introduce DocPrompting: a natural-language-to-code generation approach that explicitly leverages code documentation by (1) retrieving the relevant documentation pieces given a natural language (NL) intent, and (2) generating code based on the NL intent and the retrieved documentation. DocPrompting is general: it can be applied to any programming language, and is agnostic to the underlying neural model. We demonstrate that DocPrompting consistently improves NL-to-code models: DocPrompting improves strong base models such as CodeT5 by 2.85% in pass@1 (52% relative gain) and 4.39% in pass@10 (30% relative gain) in execution-based evaluation on the popular Python `CoNaLa` benchmark; on a new Bash dataset `tldr`, DocPrompting improves CodeT5 and GPT-Neo-1.3B by up to absolute 6.9% exact match. Data and code are available at https://github.com/shuyanzhou/docprompting.

7.1 Introduction

We address the task of natural language to code generation (NL→code): generating a code snippet, written in a general-purpose programming language such as Python or Bash, given a natural language intent. This task has seen sharply growing popularity recently due to the emergence of large language models trained on vast amounts of natural language and code [55, 90, 365]. NL→code models facilitate programming for both professional and inexperienced programmers, by allowing programmers to write code by only expressing their higher-level intent.

Many existing code generation models either learn directly from input-output pairs provided as training data [9, 11, 46, 141, 354, 364, 382], or learn the mapping between input and output implicitly from naturally occurring corpora of intertwined natural language and code [20, 253].

Nevertheless, all these works assume that *all libraries and function calls were seen in the training data*; and that at test time, the trained model will need to generate only *seen* libraries and function calls. However, new functions and libraries are introduced all the time, and even a seen function call can have unseen arguments. Thus, these existing models *inherently cannot* generalize to generate such unseen usages.

In contrast to these existing models, human programmers frequently refer to manuals and documentation when writing code [192, 257]. This allows humans to easily use functions and libraries they have never seen nor used before. Inspired by this ability, we propose DocPrompting: a code generation approach that learns to retrieve code documentation before generating the code. An overview of our approach is illustrated in Figure 7.1: First, a document *retriever* uses the NL intent ⓝ to retrieve relevant code documentation {ⓓ$_1$,ⓓ$_2$, ⓓ$_3$} from a documentation pool Ⓓ. Then, a code *generator* uses these docs in its prompt to generate the corresponding code Ⓒ. The documentation pool serves as an external data store that can be updated frequently with new contents (e.g., documentation of newly released libraries), without re-training any model component. This way, DocPrompting can leverage newly added documentation, and it can generate code containing unseen and unused functions and libraries. DocPrompting is general and applicable to any programming language and underlying base architecture. To the best of our knowledge, this is the *first* demonstration of leveraging documentation in models of code explicitly and effectively.

We demonstrate the effectiveness of DocPrompting on two NL→code benchmarks and tasks, across two programming languages, and using several base models: GPT-Neo [35], T5 [289], CodeT5 [354], Fusion-in-Decoder [143]), and Codex [55]. Further, we experiment with both

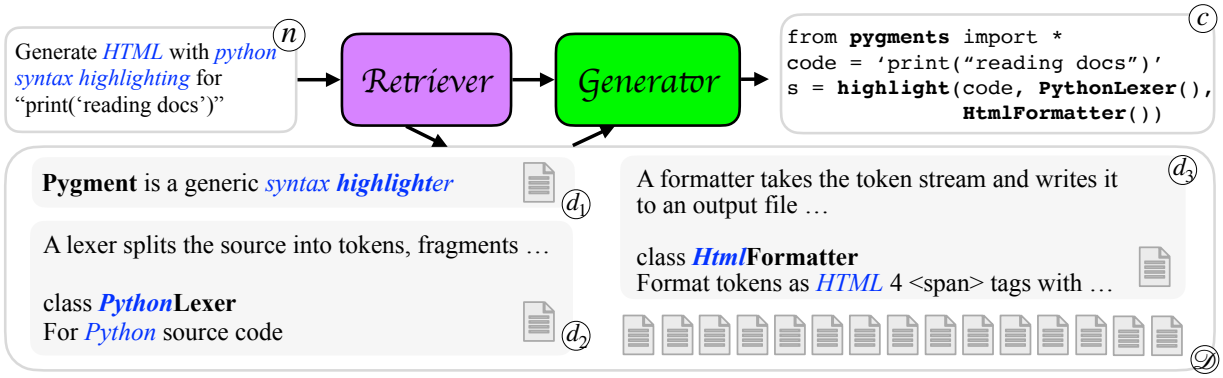


Figure 7.1: DocPrompting: given an NL intent ⓝ, the retriever retrieves a set of relevant documentation {ⓓ1,ⓓ2, ⓓ3} from a documentation pool ⓓ. Then, the generator generates the code ⓒ based on the NL and retrieved docs. DocPrompting allows the model to generalize to previously unseen usages by reading those docs. *Italic blue* highlights the shared tokens between NL and docs; **Bold** shows shared tokens between docs and the code snippet.

sparse retrievers such as BM25 [302] and dense retrieval models such as SimCSE [94]. Finally, we introduce *two new benchmarks* for retrieval-based code generation: (a) in Bash, we curate a new benchmark by crawling the `tldr` repository, and constructing the training/development/test splits without overlapping commands; (b) in Python, we re-split the popular `CoNaLa` benchmark [387] by making every test example contain at least one Python function that is not seen in the training data. Models that use DocPrompting consistently outperform their base models that generate code solely based on the NL intents. Using DocPrompting improves strong base models such as CodeT5 by 2.85% in pass@1 (52% relative gain) and 4.39% in pass@10 (30% relative gain) in execution-based evaluation in `CoNaLa`; on the new `tldr` dataset, DocPrompting improves CodeT5 and GPT-Neo-1.3B by up to absolute 6.9% exact match. We release our new benchmarks, including annotation of oracle documents for each example and pools of documentation, to serve as a test-bed for future retrieval-based code generation models.

7.2 Code Generation by Reading the Docs

Our underlying assumption is that code documentation is the most exhaustive yet succinct resource for most libraries and programming languages [305], and that documentation allows to effectively generalize to unseen libraries and functions [87]. We follow the retrieve-then-generate paradigm [111, 194], focusing on retrieving *documentation*. In this section, we describe the general approach of DocPrompting; in §7.3 and §7.6.2, we elaborate and experiment with

practical implementations of DocPrompting.

Formulation Given NL intent n, our goal is to generate a corresponding code snippet c written in some programming language (PL) such as Python. We assume that a model has access to a collection of code documentation $\mathcal{D}$. Each document $d_i \in \mathcal{D}$ describes the usage of a library, a function, or an argument in that PL. The construction of $\mathcal{D}$ is flexible: it can either be a comprehensive set of all available libraries and functions in a PL, or a customized subset for the scope of a specific project.

7.2.1 Background: Retrieval-Conditioned Generation

Although a model may use the entire collection of documents $\mathcal{D}$, only a few documents in $\mathcal{D}$ are relevant for any particular intent. Further, it is usually computationally infeasible to directly condition on the entire, unbounded, collection of documents while making predictions. Thus, we first let the model *select* a subset of documents $\mathcal{D}_n = \{d_1, d_2, .., d_k\} \subseteq \mathcal{D}$ that are potentially relevant given n, and refer to this subset while generating c.

Overall, we decompose the probability of generating c into the probability of choosing a particular subset of documents $P(\mathcal{D}_n \mid \mathcal{D}, n)$, and the probability of generating the code conditioned on the intent and the selected documents $P(c \mid \mathcal{D}_n, n)$; finally, we marginalizing over all $\mathcal{D}_n \subseteq \mathcal{D}$:

$$P(c \mid \mathcal{D}, n) = \sum_{\mathcal{D}_n \subseteq \mathcal{D}} P(c \mid \mathcal{D}_n, n) \cdot P(\mathcal{D}_n \mid \mathcal{D}, n) \tag{7.1}$$

assuming that c is independent of $\mathcal{D}$ given $\mathcal{D}_n$ (that is, $(c \perp\!\!\!\perp \mathcal{D} \mid \mathcal{D}_n)$). Since enumerating all possible subsets $\mathcal{D}_n$ is computationally infeasible, we follow the common practice and approximate the marginalization over $\mathcal{D}_n$ in Equation (7.1) by taking the most probable subset of retrieved documents $\hat{\mathcal{D}}_n$, and then conditioning the prediction of c on these most likely documents:

$$\hat{\mathcal{D}}_n := \text{argmax}_{\mathcal{D}_n \subseteq \mathcal{D}} P(\mathcal{D}_n \mid \mathcal{D}, n) \qquad P(c \mid \mathcal{D}, n) \approx P(c \mid \hat{\mathcal{D}}_n, n) \cdot P(\hat{\mathcal{D}}_n \mid \mathcal{D}, n) \tag{7.2}$$

7.2.2 DocPrompting: Generating Code by Retrieving the Docs

Equation 7.2 implies that DocPrompting relies of two main components: A *retriever* $\mathcal{R}$ retrieves relevant documents $\hat{\mathcal{D}}_n$ given the intent n; and a *generator* $\mathcal{G}$ generates the code snippet c conditioned on the retrieved documents $\hat{\mathcal{D}}_n$ and the intent n, which compose a new prompt. Specifically, $\mathcal{R}$ computes a similarity score $s(d_i, n)$ between a intent n and every document $d_i \in \mathcal{D}$. Thus, the subset $\hat{\mathcal{D}}_n \subseteq \mathcal{D}$ is the top-k documents with the highest similarity scores: $\hat{\mathcal{D}}_n = \textit{top-k}_{d_i \in \mathcal{D}}(s(d_i, n))$.

An overview of our approach is illustrated in Figure 7.1: given the intent *Generate HTML with python syntax highlighting for "print('reading docs')"*, the retriever $\mathcal{R}$ retrieves three relevant documents: d_1 describes the syntax highlighting library `pygments`, d_2 describes the class `PythonLexer`, and d_3 describes the `HtmlFormatter` class. Given these docs and the intent, the generator $\mathcal{G}$ generates the code snippet c, which uses `PythonLexer` and `HtmlFormatter` from the `pygment` library.

7.3 Practical Instantiations of DocPrompting

DocPrompting is a general approach that is not bound to any specific model choices, and it can be instantiated with any base retriever and generator. This section presents the concrete instantiations of $\mathcal{R}$ and $\mathcal{G}$ that we found to provide the best performance in our experiments.

7.3.1 Retriever Instantiation

We experiment with two main types of retrievers: *sparse* retrievers and *dense* retrievers. As our sparse retriever, we use Elasticsearch[1] with the standard BM25 [302]. This retriever represents documents using sparse features that rely on word frequencies, such as BM25 and TF-IDF.

As our dense retriever, we follow prior work [56, 94, 159]: given a triplet $(n, c, \mathcal{D}_n^*)$, where $\mathcal{D}_n^*$ are the oracle docs for n, each $d_i^+ \in \mathcal{D}_n^*$ and n form a *positive* pair (n, d_i^+), while each $d_j^- \notin \mathcal{D}_n^*$ and n form a *negative* pair $\left(n_i, d_j^-\right)$. We train the retriever in a contrastive fashion where the similarity score of a positive pair is maximized while that of in-batch negative pairs is minimized. For a pair (n_i, d_i^+), the loss function is defined as:

$$\mathcal{L}^r = -\log \frac{\exp\left(\text{sim}(\boldsymbol{h}_n, \boldsymbol{h}_{d_i^+})\right)}{\exp\left(\text{sim}(\boldsymbol{h}_n, \boldsymbol{h}_{d_i^+})\right) + \sum_{d_j^- \in \mathcal{B}/\mathcal{D}_n^*} \exp\left(\text{sim}(\boldsymbol{h}_n, \boldsymbol{h}_{d_j^-})\right)} \tag{7.3}$$

where $\boldsymbol{h}_x$ is the representation of x computed by a neural encoder, and $\mathcal{B}$ are positive docs for other examples in the batch. We define $\text{sim}(\boldsymbol{h}_x, \boldsymbol{h}_y)$ as the cosine similarity between $\boldsymbol{h}_x$ and $\boldsymbol{h}_y$.

We use all (n_i, d_i^+) in the training set as our supervised training dataset. Additionally, we use all sentences in the documentation pool for weak supervision: Following Chen et al. [56] and Gao et al. [94], representations of the same sentence with different dropout masks are treated as a positive example. Instead of using either supervised or weakly supervised training as in Gao et al. [94], we simply mix the two resulting supervision signals, and examples are randomly distributed

[1] https://github.com/elastic/elasticsearch

into batches. This mixture of tasks not only facilitates the learning process (§7.6.2), but also reduces the engineering effort required to store and reload models for separate supervised and unsupervised training phases. We initialize the retriever encoder with either the best model of Gao et al. [94] or the encoder of CodeT5-base [354]. Additional training details are provided in Appendix 7.9.3

7.3.2 Generator Instantiation

We experimented with a variety of generator models. We used GPT-Neo-125M, GPT-Neo-1.3B [35] and Codex [55], where we concatenate the retrieved documents and the NL intent as a single, long, prompt. T5-base [288] and CodeT5-base [354] have a shorter input size of 512 tokens, which is sometimes too short for the concatenation of multiple docs. Thus, for T5 and CodeT5 we apply the fusion-in-decoder approach [FiD; 143]: we first concatenate the intent n with each retrieved $d_i \in \hat{\mathcal{D}}_n$ and encode each (n, d_i) pair independently. Then, the decoder attends to *all* encoded NL-document pairs. We finetune the generator to maximize the log-likelihood of the reference code c given n and $\hat{\mathcal{D}}_n$.

With Codex [55], we performed few-shot learning rather than finetuning because the model parameters are not publicly available. We constructed the prompt with three static examples, each of which is a concatenation of retrieved documentation, an NL intent and the reference code snippet. We then appended the test example and its retrieved documentation to the few-shot examples. We used the *code-davinci-001* version because we suspect potential leakage of the test set into the training set of *code-davinci-002*. See more details in Appendix 7.9.8. Training details, hyper-parameter settings and example prompts can be found in Appendices 7.9.5 and 7.9.4.

7.4 Experimental Setup

We evaluate DocPrompting on two NL→code tasks: shell scripting (§7.4.1), in which we generate complex shell commands given an intent, and Python programming (§7.4.2), where we generate answers in Python for NL questions. In this section, we first introduce a *newly curated* benchmark `tldr`; we then describe our re-split of the popular `CoNaLa` benchmark [387]. For each benchmark, we provide a global documentation pool $\mathcal{D}$ that is shared for all examples and oracle documents $\mathcal{D}_n^*$ which we use to train the retriever. We release our newly curated benchmarks to serve as test-bed for future retrieval-based code generation models.

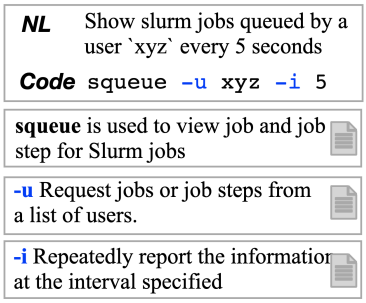


Figure 7.2: An example NL-code pair from `tldr`, along with three oracle documentation items.

7.4.1 Shell Scripting

`tldr` is a community-driven project that maintains easily-readable help pages with examples for over 2.5k Bash commands in over 25 natural languages[2]. We collected pairs of English intents and Bash command lines. The NL intents are written by human users, and the Bash commands range from popular ones like `cat` and `tar`, to uncommon commands such as `toilet` and `faketime`. Our resulting `tldr` benchmark contains 1,879 unique Bash commands and 9,187 NL→Bash pairs. We constructed the training, development and the test set with *completely disjoint commands* to test the generalizability of a code generation model. The shared documentation pool $\mathcal{D}$ is made up of the 400k paragraphs from the 1,879 Bash manuals. Each paragraph describes a single concept such as an argument flag. We further curated the oracle documents $\mathcal{D}_n^*$ for each example using simple string matching. An example from `tldr` is shown in Figure 7.2. To the best of our knowledge, this is the first work to leverage `tldr` as an NL→code benchmark. Detailed statistics and additional details are provided in Appendix 7.9.1. In `tldr`, each NL intent results in a single Bash command with a combination of argument flags. We therefore first retrieve an entire Bash manual; then, we take the top manual and retrieve the top-10 paragraphs from that manual.

Evaluation metrics We measure: (1) command name accuracy (CMD Acc) – whether the command name (*e.g.*, `cat`) is an exact match; (2) exact match (EM) – exact match between the reference and the generation; (3) token-level F1; and (4) character-level BLEU [charBLEU; 210, 318]. In all metrics, we disregard user-specific variable names in the references and the models outputs. For example, "`mycli -u [user] -h [host] [database]`" is evaluated as "`mycli -u $1 -h $2 $3`".

[2] https://github.com/tldr-pages/tldr

7.4.2 Python Programming

CoNaLa [387] is a popular benchmark for NL→Python generation. NL intents are StackOverflow questions, and code snippets are their answers. Both intents and code snippets are rewritten by human annotators. We re-split the dataset to test models' generalization to unseen Python functions. In our re-split, we verifed that every example in the development or the test set uses at least one Python function (*e.g.*, `plt.plot`) that was *not* seen in the training data. In addition, we make sure that the examples from the same StackOverflow posts are in the same set to prevent leakage. This re-split results in 2,135/201/543 examples in the training/development/test sets, respectively.

The CoNaLa documentation pool $\mathcal{D}$ contains 35,763 documents, each describing a single function, from all Python libraries available on DevDocs (https://devdocs.io). These include built-in libraries and other popular libraries such as numpy. We constructed the oracle docs $\mathcal{D}_n^*$ for each example by matching all function names in the target code c with docs. More details in Appendix 7.9.2.

Evaluation metrics We follow Yin et al. [387] and measure BLEU-4. Since we focus on generalization to unseen functions, we additionally report function name recall (*recall*) and unseen function recall ($recall_{unseen}$), which measures recall among function calls that do not appear in the training set. Finally, following Austin et al. [20], Chen et al. [55], we used the manually written unit tests from Wang et al. [356] for 100 examples from CoNaLa's test set and measure pass@k. We followed Chen et al. [55] and performed nucleus sampling [135] with $p = 0.95$. For each k, we searched for the best temperature for each model from $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. On average, each example has 2.03 tests. The concatenation of multiple Python docs often exceeded the length limit of GPT-Neo, we hence experimented in this dataset with FiD, which allows longer inputs. Additional details are provided in Appendix 7.9.2.

7.5 Results

In all following results, all models with DocPrompting use the top-10 retrieved docs from the best retriever on that dataset (Table 7.4). Every baseline uses the exact same setup as its "+DocPrompting" version, except for not using the documentation.

Table 7.1: Results on shell scripting, using a BM25 retriever with top-10 retrieved docs, on the test set of `tldr`. For the "oracle command name" experiments, we selected the best model of each type.

Model		CMD Acc (%)	EM (%)	Token F1	charBLEU
GPT-Neo-125M	-	11.96	1.94	28.75	19.99
	+DocPrompting	**25.32**	**3.56**	**31.23**	**24.43**
GPT-Neo-1.3B	-	14.55	3.12	32.46	24.70
	+DocPrompting	**27.59**	**9.05**	**37.24**	**30.57**
T5	-	10.02	0.76	19.90	25.48
	+DocPrompting	**30.28**	**9.16**	**37.58**	**31.97**
CodeT5	-	14.60	2.18	30.00	21.50
	+DocPrompting	**30.72**	**9.15**	**36.71**	**33.83**
Codex *3-shots*	-	27.48	8.94	36.04	16.94
	+DocPrompting	**31.21**	**9.29**	**36.77**	**23.72**
With the oracle command name					
T5	-	-	12.96	59.36	45.05
	+DocPrompting	-	**22.55**	**64.84**	**54.28**
Codex *3-shots*	-	-	22.44	62.26	50.29
	+DocPrompting	-	**32.43**	**69.73**	**55.21**

7.5.1 Shell Scripting Results

Results for `tldr` are shown in Table 7.1. DocPrompting consistently improves the base models. For example, T5+DocPrompting achieves more than *twice* higher accuracy in predicting the command name, more than 16 charBLEU points on the entire prediction, and almost 9% of absolute exact match gain, compared to the vanilla T5. In the few-shot learning setting with Codex, DocPrompting brings gains of 6.7 charBLEU points, and consistent improvement across all metrics over the baseline that observes only NL-code pairs in its prompt. These results show that retrieving documentation also benefits strong models such as Codex, and with only few examples in the context.

Table 7.2: Comparison to approaches that retrieve examples [267, 268]

Model		CMD Acc (%)	EM (%)	Token F1	charBLEU
GPT-Neo-125M	+ExPrompting	6.68	0.32	20.49	11.15
	+DocPrompting	**25.32**	**3.56**	**31.23**	**24.43**
GPT-Neo-1.3B	+ExPrompting	14.01	2.8	30.07	22.11
	+DocPrompting	**27.59**	**9.05**	**37.24**	**30.57**

Code generation with oracle command names In realistic settings, a human programmer may know the command name they need to use (*e.g.*, `awk`), but not know the exact usage and flags. In fact, better understanding of the usage of *known* commands is the purpose of Unix `man` pages and the `tldr` project. We conducted an oracle experiment where we provided T5 (which was the strongest model using DocPrompting) and Codex with the oracle command name (*e.g.*, `awk`). This oracle information is provided to both the baseline and the model that uses DocPrompting. The results are shown on the bottom part of Table 7.1. When the oracle command is given, DocPrompting further improves over the base models. For example, when providing Codex with the ground truth command name, DocPrompting improves its exact match from 22.44% to 32.43%.

Should we retrieve documentation or examples? All existing retrieval-based models of code retrieve NL-code pairs or code snippets, rather than documentation. To simulate this scenario, we followed Parvez et al. [267] and Pasupat et al. [268] to retrieve NL-code pairs from the training set of `tldr`, and refer to this baseline as `ExPrompting`. We finetuned the best retriever RoBERTa and two generators, and retrieved the top-30 NL-code pairs for every example. As shown in Table 7.2, *retrieving documentation* (DocPrompting) provides much higher gains than retrieving examples (`ExPrompting`). Theoretically, adding examples of unseen commands can help `ExPrompting` generalize to them as well. However, new libraries and functions may not have available examples on the web yet, while documentation often *does* becomes available when the library is released.

Table 7.3: Results on CoNaLa, using a CodeT5 retriever with top-10 retrieved docs. Function recall (Recall) measures how many functions in the reference code are correctly predicted, and unseen function recall ($\text{Recall}_{\text{unseen}}$) only considers the subset held out from the training data.

Model		BLEU	Recall	Recall_{unseen}
Codex *3-shots*	-	43.16	39.52	-
	+ DocPrompting	**43.47**	**39.87**	-
	+ DocPrompting oracle docs	50.59	57.84	-
T5	-	28.07	14.36	2.57
	+ DocPrompting	**30.04**	**21.34**	**8.24**
CodeT5	-	34.57	24.24	9.03
	+ DocPrompting	**36.22**	**27.80**	**18.30**
	+ DocPrompting oracle docs	49.04	72.20	63.91

7.5.2 Python Programming Results

Table 7.3 shows the results on CoNaLa. CodeT5+DocPrompting yields a 1.65 BLEU improvement over the state-of-the-art baseline that was initialized with CodeT5.[3] When measuring the recall of the generated function names, the benefit of DocPrompting is especially higher for *unseen* functions ($recall_{unseen}$). For example, DocPrompting achieves 18.30 compared to only 9.03 of the base CodeT5 in unseen functions. Additionally, DocPrompting improves in-context learning setting with Codex. We hypothesis that the minor gain is mainly due to the potential data leakage of Codex, which violates the split of seen and unseen functions. Another reason is that a strong generator such as Codex may require an equally strong retriever as well. We find that Codex can achieve even higher results with an oracle retriever, which shows the potential further improvement by improving the retrievers. Finally, CodeT5 performs better than T5, with and without using DocPrompting. This emphasizes the importance of using code-specific pretrained models.

Execution-based evaluation The results are shown in Figure 7.3. Using DocPrompting consistently outperforms the baseline CodeT5 for all values of pass@k. For example, DocPrompting yields 2.85% improvement on pass@1 and 4.45% improvement on pass@5, which are realistic

[3] In a separate experiment on the original split of CoNaLa, this baseline achieved a BLEU score of 39.12, which outperforms the previous state-of-the-art [28] by 4.92 BLEU points.

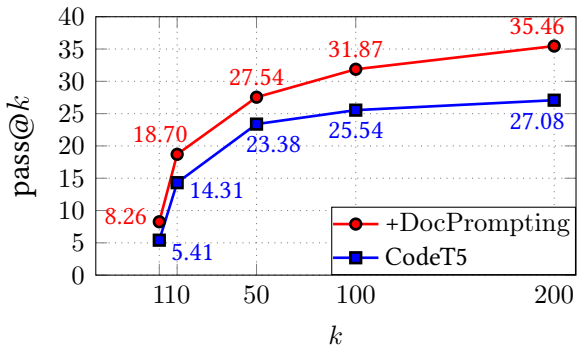


Figure 7.3: Pass@k of CodeT5 with and without DocPrompting on 100 CoNaLa examples.

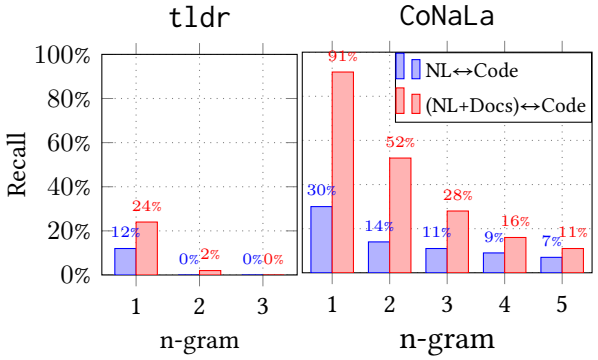


Figure 7.4: Using documentation significantly increases the n-gram overlap recall between the input and the output, in tldr and CoNaLa.

numbers of completions that can be suggested in an IDE. When k = 200, DocPrompting widens the gap to 8.38%. These results demonstrate that DocPrompting does not only improve the quality of the generated code in its surface form, but also increase its functional correctness. Additional details and results are provided in Appendix 7.9.7.

7.6 Analysis

7.6.1 Why does reading the documentation help generating more accurate code?

We believe that one of the major reasons is that *documentation eases the mapping between NL intents and code*, since the documentation contains both NL descriptions *and* function signatures. We calculated the n-gram overlap between the NL intents and their corresponding code snippets (NL↔code), and the overlap between the NL intents with their top-10 retrieved documents and their code snippets ((NL+docs)↔code). As shown in Figure 7.4, adding documentation *significantly increases* the overlap across n-grams, and increase, for example, the unigram overlap from 12% to 24% in tldr. That is, one of the reasons that retrieving documentation helps generating accurate code is that documentation bridges the gap between the "intent terminology" and the "code terminology".

Table 7.4: Retrieval performance of multiple models on the dev set of `tldr` (top) and `CoNaLa` (bottom). RoBERTa is the best model taken from from Gao et al. [94], and CodeT5 is the encoder of CodeT5-base [354]. Models with the subscript "off-shelf" are the off-the-shelf models, and the other models were finetuned with the objective in Equation 7.3. The last column is the best model (RoBERTa for `tldr` and CodeT5 for `CoNaLa`) trained without the weak supervision corpus.

	n	BM25	$\text{RoBERTa}_{\text{off-shelf}}$	RoBERTa	$\text{CodeT5}_{\text{off-shelf}}$	CodeT5	Best w/o weak sup.
`tldr`	1	**32.81**	17.53	30.03	10.45	18.10	28.30
	5	51.73	37.89	**52.50**	20.26	38.52	50.50
	10	59.86	46.80	**60.33**	25.73	51.03	59.84
	20	62.01	56.11	**64.30**	33.65	57.26	62.30
`CoNaLa`	1	3.01	4.46	13.49	4.60	**16.54**	10.51
	5	7.16	7.58	26.38	8.63	**42.35**	21.15
	10	9.73	10.93	34.86	12.25	**55.81**	29.34
	20	11.46	13.89	45.46	18.46	**66.79**	42.21

7.6.2 Ablation Study

We compared different configurations of the retriever, to gather more insights for effective DocPrompting. Table 7.4 shows a comparison between different retrievers and their setups. First, the performance of BM25 varies among datasets: In `tldr`, BM25 matches the recall of trained dense retrievers; however in `CoNaLa`, BM25 achieves only recall@10 of 9.73%, and strong dense retrievers such as the encoder of CodeT5 achieve recall@10 of 55.81. We hypothesize that this difference between datasets stems from the ways these datasets were created: `tldr` intents were written based on existing Bash commands and manuals; while `CoNaLa` examples were mined from StackOverflow posts, where users ask questions with limited or no context. Thus, NL intents in `CoNaLa` require a better semantic alignment with the documents, and thus benefit from dense retrievers. The gap resulting from different data curation processes was also observed by Rodriguez and Boyd-Graber [304] in open-domain question answering (QA).

Second, retrievers that were pretrained on the target programming language are generally stronger. For example in `CoNaLa`, CodeT5 which was pretrained on Python, is both a better off-the-shelf retriever and a better finetuned-retriever than RoBERTa, which was pretrained mainly on text. In contrast, `tldr` is based on Bash, which neither CodeT5 nor RoBERTa were explicitly pretrained on. Thus, `tldr` benefits mostly from BM25 and RoBERTa rather than

CodeT5 as retrievers.

Finally, training the retriever using weak supervision on the documentation pool (Section 7.3.1) dramatically improves the retriever. The recall of the best retrievers of each dataset without this corpus is shown in the last column of Table 7.4 ("Best w/o weak sup."). On CoNaLa, removing this corpus results in severe performance degradation. One possible explanation is that this weak supervision helps the retriever perform domain adaptation more effectively.

7.6.3 Case study

We examine the models' outputs and show two representative examples in Table 7.5. In the first example, `Image.open` was not seen in the training set, and the baseline CodeT5 incorrectly predicts `os.open`. In contrast, using DocPrompting allows to retrieve the docs and to correctly predict `Image.open`. In the second example, `df.to_csv` was not seen in training, and the baseline CodeT5 fails to correctly predict it. In contrast, DocPrompting *does* predict most of the `df.to_csv` call correctly, thanks to the retrieved docs. Nevertheless, DocPrompting generates an incorrect argument `skiprows=1`, instead of `header=False`. The reason is that along with the retrieved documentation of `df.to_csv`, the retriever also retrieved the documentation of `df.read_csv`, which has a `skiprows` argument. That is, the generator uses an argument of `df.read_csv` with the function `df.to_csv`. Further improving the retrievers and the generators, and post-filtering based on the validity of argument names, may mitigate such mistakes.

Table 7.5: Examples of predictions from CoNaLa, of the base CodeT5 compared to CodeT5+DocPrompting. Unseen functions are underscored.

NL Intent: **Open image "picture.jpg"**	
Ground truth:	`img = Image.open('picture.jpg') \n Img.show`
CodeT5:	`os.open('picture.jpg', 'r')`
CodeT5+DocPrompting:	`image = Image.open('picture.jpg', 'rb')`
NL Intent: **Exclude column names when writing dataframe 'df' to a csv file 'filename.csv'**	
Ground truth:	`df.to_csv ('filename.csv', header=False)`
CodeT5:	`df.drop(['col1', 'col2'], axis=1, inplace=True)`
CodeT5+DocPrompting:	`df.to_csv('filename.csv', skiprows=1)`

7.7 Related Work

Code generation The most common practice in NL→code generation is training a model on a dataset of NL-code pairs [9, 141, 284, 382]. Nevertheless, all these works assume that their training corpus covers *all* required libraries and functions, and their models are inherently incapable of generating libraries and functions that were not seen in the training data. On the contrary, DocPrompting allows models to generate calls to unseen function, by retrieving these functions' documentation and reading them at test time. Hashimoto et al. [115], Hayati et al. [119], Parvez et al. [267] and Lu et al. [221] learn to retrieve examples at test time; Pasupat et al. [268] also considered settings where the test data has a distribution shift from the training data. However, when new libraries are released they often come with documentation, and thus we assume that documentation for new libraries is much more likely to be available than concrete natural language intent and code snippet pairs (n, c) that use these libraries already. The models of Shrivastava et al. [325] and Wu et al. [362] retrieve code snippets from relevant files in the same project; contrarily, when predicting new libraries and functions that are *external* to the user's project, documentation is the source that is the most likely to be available.

Retrieval augmented generation The paradigm of retrieve-then-generate has gained popularity in the field of open-domain question answering [111, 159, 194], where the answer for an open-domain question exists in only few documents out of a much larger pool. Although DocPrompting takes a similar approach, documentation retrieval in code generation is even more valuable, since code libraries are updated constantly, and new libraries are introduced daily. Thus, DocPrompting allows updating the documentation pool frequently with new contents, without re-training any model components.

Documentation conditioned generation The model of Zhong et al. [405] reads documents to understand environment dynamics in a grid-world game, and Branavan et al. [40] controls situated agents in a game (Civilization II) by reading the game's manual. However, all their models were tailored to specific games; in contrast, DocPrompting is general and is applicable for a variety of programming languages and datasets.

7.8 Conclusion

We propose DocPrompting, a simple and effective approach for code generation by retrieving the relevant documentation. DocPrompting consistently improves NL→code models in two tasks, in two PLs, and across multiple strong base models. DocPrompting improves strong base

models such as CodeT5 by 2.85% in pass@1 (52% relative gain) in execution-based evaluation on the popular Python CoNaLa benchmark; on a new Bash dataset tldr, DocPrompting improves CodeT5 and GPT-Neo-1.3B by up to 6.9% exact match, and Codex by 6.78 charBLEU score.

These results open a promising direction for NL→code generation. We believe that our results can be further improved using more clever encoding of the structured nature of long documents, and using joint training of the retriever and the generator, which hopefully will avoid cascading errors. Further, we believe that the principles and the methods presented in this paper are applicable to additional code-related tasks, and other documentation-like resources such as tutorials and blog posts. To these ends, we make all our code, data, and models publicly available.

7.9 Appendix

7.9.1 tldr: A Newly Curated Shell Scripting Benchmark

NL→Bash pairs For each command (*e.g.*, cat), users contribute examples of pairs of NL descriptions and bash code (mainly one-liners), including various flags and arguments, which cover the common usages of that command. An example is shown in Figure 7.2.

We crawl NL-code pairs from the markdown files[4] in the linux and common folders. We discard Bash commands whose manual is unavailable (discussed below). The detailed statistics are shown in Table 7.6. On average, each command has 4.84 NL→Bash pairs and there is a total of 9187 NL-code pairs. To test the generalizability of a model, we construct the training, development and the test set *with completely different commands*.

Table 7.6: The statistics of the tldr shell scripting benchmark

	# Commands	NL→Bash pairs
train	1315	6414
dev	376	1845
test	188	928
total	1879	9187

[4] *e.g.*, https://github.com/tldr-pages/tldr/blob/main/pages/linux/toilet.md

Documentation pool $\mathcal{D}$ We take the bash manual of the 1897 bash commands in `tldr` to construct a documentation pool. We search each command name at `manned.org`[5], a website which archives Unix manual pages (the same as the Unix ‘ `man <command>` command), and then extract the text contents from the returned manual page. We further break each manual into multiple paragraphs by line breaks so that each paragraph delicately describes a single concept such as a command functionality or a flag usage. We make this decision due to the large volume of content each manual has, which is too long to fit the length limitation of a neural model, and too noisy and distracts the model with irrelevant information. This results in $400k$ individual entries in the pool in total.

Oracle manual $\mathcal{D}_i^*$ We find the ground truth documentation for each (n, c) pair through command name and flag matching heuristics. For instance, given a code snippet `toilet 'input_text' -f 'font_filename'`, we constrain our search to the documentation from `toilet` manual page and select documentation that starts with `-f` flag as an oracle paragraph. Along with the first paragraph that commonly summarizes a command, these paragraphs forms $\mathcal{D}_n^*$.

Evaluation metrics We use four evaluation metrics to measure the quality of the generated code: (1) command name accuracy (*CMD Acc*) – measures whether the command name (*e.g.*, `cat`) is predicted correctly; (2) token-level F1 – converts the reference code and the generated code to bag of words and measures the token-level precision, recall, and F1 overlap; (3) exact match (EM) – measures the exact match between the reference and the generation; and (4) character-level BLEU [charBLEU; 210, 318].

For token level F1, exact match, and charBLEU, we disregard all user-specific variables in the references and the system outputs. For example, "`mycli -u [user] -h [host] [database]`" is converted into "`mycli -u $1 -h $2 $3`". This is mainly because the variables are not instantiated in `tldr` and the style of the placeholder varies among contributors. For example, some contributors might write `[user]` as `[username]` or `[your_name]`. Therefore, measuring the surface form of user-specific variable names is less meaningful.

[5] https://manned.org

7.9.2 Re-splitting CoNaLa

NL→Python pairs We adapt the popular CoNaLa benchmark and re-split the dataset to test the generalization scenario. This re-split makes every example in the development and the test set have at least one Python function (*e.g.*, `plt.plot`) that was not seen in the training data. There are 2135, 201, and 543 examples in the training, development and test sets, respectively. We follow the original work [387] to evaluate the system outputs with BLEU-4. Since we focus on the generalization setting, we additionally report unseen function accuracy, which measures the percentage of correctly predicted held-out functions that do not appear in the training set.

Human-annotated unit tests Following Chen et al. [55] and Austin et al. [20], we conduct execution-based evaluation on CoNaLa to measure the functional correctness of the generated code. We randomly selected 100 examples from the test set and manually annotated unit test for each example. For example, we wrote tests such as `assert gen_code("abcds", 2) == 4` and `assert gen_code("abde", 2) == -1` to verify whether the function `gen_code` could perform "*find the index of sub string 's' in string 'str' starting from index 2*". Each example was annotated by a single annotator. The annotation was done by two authors of the paper who program with Python daily. On average, we annotate 2.03 unit tests for each example.

Documentation pool $\mathcal{D}$ Our documentation pool contains 35763 manuals. These functions are from all Python libraries that are available on DevDocs[6]. These libraries contains the Python built-in library, and popular libraries like numpy and pandas. The documentation on DevDocs are curated and further transformed and indexed to allow for quick searching of APIs. We then extract each API signature and the corresponding documentation in every library, remove any content in the documentation that is not text, and segment the documentation into multiple paragraphs based on the <p> HTML tags. The documentation pool then contains pairs of the API signature and a single paragraph in the corresponding documentation. Although the documentation pool is not comprehensive to cover all Python libraries and functions, we find it has a high coverage rate on the CoNaLa dataset. This choice reflects the flexibility of our approach upon the characteristics of a target scenario.

Oracle manual $\mathcal{D}_i^*$ To find the oracle documents for a given NL intent $\mathcal{D}_i^*$ from the original (n, c) example, we first index the function names with absolute path (*e.g.*, `plot` is indexed with

[6] https://devdocs.io

`matplotlib.pyplot.plot`) with Elasticsearch. Then we query the search engine with clean version of c where variable name are removed. The top-5 functions after de-duplication are treated as oracle manuals $\mathcal{D}_i^\star$.

Natural language and code associations during pretraining Despite our efforts, it is possible that some of the held-out functions in the test set were seen to associate with NL contexts (*e.g.*, comments) during the pretraining of a retriever and a generator. Since the generators were initialized from the same checkpoint in both the baselines and the DocPrompting models, such a possible association is expected to equally help both models. In the retriever, such a possible association did not cause the retriever to see the *exact NL intents* together with the corresponding documentation, and thus the matching between NL↔doc was not leaked. However, it is possible that there had been semantically similar intents seen along with the code snippets of the held-out functions. Nevertheless, such co-occurrence is "indirect" and "unsupervised".

7.9.3 Dense Retriever Training

We finetune the model for 10 epochs with batch size of 512 and learning rate of $1e-5$. Since CodeT5 does not use `[CLS]` token, we alternatively take the average of the hidden state of the last layer as the text representation. For `CoNaLa`, we also use the first $100k$ "mined" examples provided as part of `CoNaLa` as the supervised corpus. For `CoNaLa`, we only apply a single search step because each code snippet commonly contains more than one function. We also observed that using the first sentence that normally summarizes the usage of a function achieve the best retrieval performance than other alternatives such as using the first paragraph, or simply truncating to the maximum token length. The training takes up to 15 hours on a single A6000 GPU.

7.9.4 Generator Training

We train our single-source generators for 20 epochs with learning rate $4e-5$. We train our FiD-based generators for 10000 steps. The doc length is set to 200, any further content will be truncated. We follow [143] to set learning rate to $5e-5$ with 2000 steps warmup and linear learning rate decay. The batch size is set to 8. The best model is selected based on the token-level

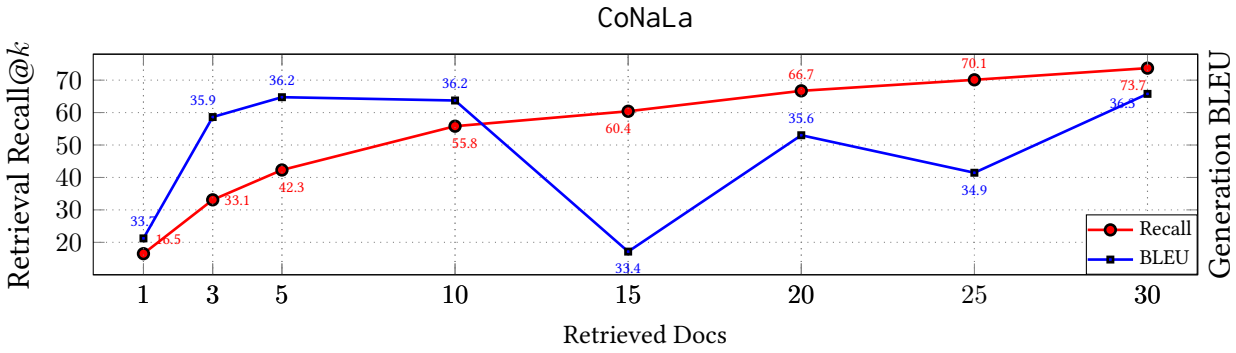


Figure 7.5: The recall@k (%) and the corresponding BLEU score by using these top-k docs on `CoNaLa` dataset (using CodeT5).

F1 score on the development set for `tldr` and BLEU score for `CoNaLa`. The training takes 8 hours on a single A6000 GPU.

7.9.5 Codex Prompts

For the baseline, we prompt Codex with three NL-code pairs and append the test query to the end. An example on `tldr` is shown on top of Table 7.7. On the bottom, we list the prompt with DocPrompting where documentation is provided along too. In the oracle command name setting, we prepend the command name before each NL intent for the baseline prompt. For DocPrompting prompt, we replace the potential docs with the retrieved docs from the oracle manual.

7.9.6 Additional Analysis

Parameter efficiency As shown in Table 7.1, under a given parameter budget, we find that DocPrompting mostly benefits from parallel encoding (FiD). For example, the parallel encoding T5+DocPrompting (220M parameters) significantly outperforms the 125M parameters joint encoding Neo-125M+DocPrompting. Only scaling up Neo+DocPrompting to 1.3B parameters manages to match the 220M parameter T5+DocPrompting. A possible explanation is that although the base Neo-1.3B (without DocPrompting) generally performs better than the base T5 (without DocPrompting), parallel encoding allows to utilize the retrieved documents better, since documents are encoded independently on the encoder side.

The impact of the number of documents Figure 7.5 shows the recall@k and the BLEU score compared to k, the number of retrieved documents. Increasing k consistently yields a higher

get the label of a fat32 partition
`fatlabel /dev/sda1`
END
display information without including the login, jcpu and pcpu columns
`w –short`
END
sort a csv file by column 9
`csvsort -c 9 data.csv`
END
search for a package in your current sources

Potential document 0: fatlabel will display or change the volume label or volume ID on the MS- DOS filesystem located on DEVICE ...
get the label of a fat32 partition
`fatlabel /dev/sda1`
END
Potential document 0: w displays information about the users currently on the machine, and their processes. The header shows, in this order ...
Potential document 1: -s, –short Use the short format. Don't print the login time, JCPU or PCPU times.
display information without including the login, jcpu and pcpu columns
`w –short`
END
Potential document 0: Sort CSV files. Like the Unix "sort" command, but for tabular data
Potential document 1: usage: csvsort [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u 0,1,2,3] [-b] [-p ESCAPECHAR]
Potential document 2: optional arguments: -h, –hel show this help message and exit -n, –names Display column names and indices from the input CSV and exit. -c COLUMNS ...
Potential document 3: csvsort -c 9 examples/realdata/FY09_EDU_Recipients_by_State.csv
Potential document 4: csvcut -c 1,9 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvsort -r -c 2 | head -n 5
sort a csv file by column 9
`csvsort -c 9 data.csv`
END
Potential document 1: ...
Potential document 2: ...
search for a package in your current sources

Table 7.7: Top: baseline Codex prompt with three NL-code pairs and a test intent. Bottom: DocPrompting prompt for Codex. In each in-context learning example, the oracle docs, the NL intent and the corresponding bash command are provided. We use up to five oracle docs for these examples. For a test example, the top-5 paragraphs from the retriever are represented with the NL intent. The documents' contents were omitted ("...") to save space.

Table 7.8: n-gram overlap between different contents (%). Using documentation significantly increases the n-gram overlap recall between the input and the output, in `tldr` and `CoNaLa`.

`tldr`	1	2	3
NL↔Code	12	0	0
(NL+retrieved docs)↔Code	24	2	0
NL↔Retrieved docs	39	8	3

`CoNaLa`	1	2	3	4	5
NL↔Code	30	14	11	9	7
(NL+retrieved docs)↔Code	91	52	28	16	11
NL↔Retrieved docs	72	14	3	1	1

recall; however, as more irrelevant documents are retrieved, the generator cannot effectively distinguish them from the relevant ones and the overall performance remain similar. For example, CodeT5 achieves the highest BLEU score using $5 \leq k \leq 10$. In contrast, when the generator is provided with the oracle docs only, its BLEU score reaches 49.04 (Table 7.3). This suggests that both precision and recall of docs are important, and the benefit of using larger values of k in open domain QA [143] does not necessarily hold in code generation.

Full n-gram overlap Table 7.8 shows that using documentation significantly increases the n-gram overlap recall between the input and the output, in `tldr` and `CoNaLa`. Since we used BM25 to retrieve docs in `tldr`, the NL↔Retrieved docs overlap is high by construction. In `CoNaLa`, the NL↔Retrieved docs unigram overlap is high as well, but since we used a *dense* retriever, the general n-gram overlap does not have to be high for DocPrompting to work well.

Retrieval latency Although retrieving docs results in additional test-time computation, the increase in latency is not prohibitive. First, encoding the input for the retrieval step "costs" a *single forward pass* through the retriever's encoder, which is significantly less expensive than generation (which requires multiple time steps of the decoder). All the documentation in the retrieval pool can be encoded in advance, and finding the top-k results can be performed quickly using libraries such as FAISS [150] on the GPU or ScaNN [108] on CPU. The cost of this top-k search is sub-linear in the size of the document pool. Second, the additional input to the generator results in an increased memory consumption, but only a small increase in latency since the tokens of a given input can be encoded in parallel. If this difference is crucial in practical settings, we can decrease the number of retrieved documents. Figure 7.5 shows that retrieving as few as five docs may be sufficient in many cases.

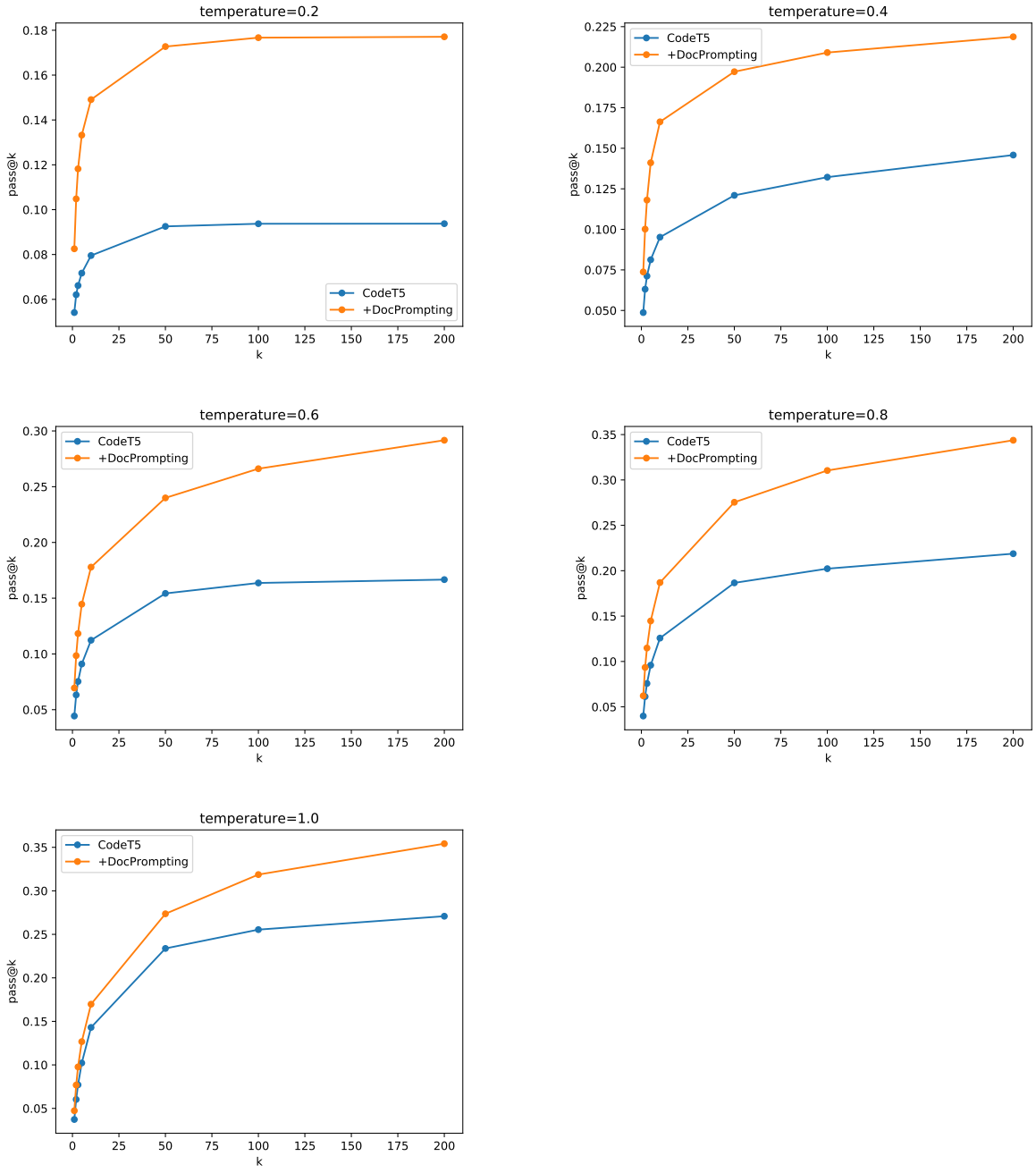


Figure 7.6: Pass@k on 100 examples on the test set with different temperatures.

7.9.7 Full Pass@k Plots

In the main execution-based evaluation, pass@k results in Section 7.5.2 and Figure 7.3, we took the best temperature for every model and value of k. Here, we show all the pass@k plots with different temperatures in Figure 7.6.

7.9.8 Experiments with *code-davinci-002*

The results with *code-davinci-002* under few-shot learning setting is shown in Table 7.9. In the non-oracle settings, Codex+DocPrompting did not improve over the base Codex; one explanation might be that the datasets are leaked into the training corpus of the Codex. For example, `CoNaLa` was extracted from StackOverflow, which is included in the large CommonCrawl corpus[7] that was used to train GPT-3, and possibly Codex. Therefore, Codex might have memorized the target code, and thus did not need the additional documentation. Although the data leakage issue might have happened in *code-davinci-001* as well, we suspect that this issue has worsened in the stronger *002* version. Regardless, we believe that the large capacity of Codex requires an equally strong retriever to improve over the base model. With an oracle retriever, DocPrompting yields significant improvement on both datasets. Thus, the non-oracle results could be further improved using a stronger non-oracle retriever.

7.9.9 Examples

7.9.10 `tldr`

Examples on `tldr` are in Table 7.10 . In the top three cases, the baseline T5 could not generate the correct bash command while T5+DocPrompting retrieves the correct bash documentation and use the correct arguments. In the bottom two cases, T5 also predicts the bash command correctly. However, in the `puppet` case, it misses concrete usages of executing the code. In the `elixir` case, it halluciate the argument flags `-example` and `-arguments` which `elixir` does not support at all. On the other hand, T5+DocPrompting does ground its generation to the retrieved docs and therefore it generates more accurate code.

7.9.11 `CoNaLa`

More examples on `CoNaLa` are shown in Table 7.11.

[7] https://commoncrawl.org/the-data/

Table 7.9: Results on `tldr` and `CoNaLa` with *code-davinci-002*.

`tldr`					
Model		CMD Acc (%)	EM (%)	Token F1	charBLEU
Codex	-	**39.01**	**14.55**	**44.89**	**33.93**
3-shots	+DocPrompting	36.10	13.97	42.55	32.93
With the oracle command name					
	-	-	20.22	59.22	38.14
	+DocPrompting	-	**33.15**	**68.59**	**44.76**

`CoNaLa`		
	BLEU	Recall
-	**48.39**	43.35
+ DocPrompting	47.21	**44.70**
+ DocPrompting oracle docs	54.67	59.68

Table 7.10: Examples of predictions from `tldr`, of the base T5 compared to T5+DocPrompting. In `tldr`, the bash commands in the test set are completely different from those in the training set.

play a midi file in a specific key (0 = c major/a minor, 1 = f major/d minor, +1 = g major/e minor, etc.)	
Ground truth:	`timidity –force-keysig='-flats|+sharps' 'path/to/file.mid'`
T5:	`midiplay 'key'`
T5+DocPrompting:	`timidity –force-keysig 'f|f|f' 'path/to/file.midi'`
clean up all temporary tex files in the current directory	
Ground truth:	`latexmk -c`
T5:	`tex clean`
T5+DocPrompting:	`latexmk -c`
specify the hash algorithm for the index file (experimental)	
Ground truth:	`git show-index –object-format='sha1|sha256' 'path/to/file'`
T5:	`hash –experimental 'path/to/index.hashes'`
T5+DocPrompting:	`git show-index –object-format= 'path/to/hash-algorithm' 'path/to/experimental'`
execute puppet code	
Ground truth:	`puppet apply –execute 'code'`
T5:	`puppet`
T5+DocPrompting:	`puppet apply –execute 'path/to/file.puppet'`
evaluate elixir code by passing it as an argument	
Ground truth:	`elixir -e 'code'`
T5:	`elixir –example –arguments 'path/to/file.elixir'`
T5+DocPrompting:	`elixir -e 'path/to/file.elixir'`

Table 7.11: Examples of predictions from CoNaLa, of the base CodeT5 compared to CodeT5+DocPrompting. Unseen functions are underscored.

set the current working directory to 'c:\Users\uname\desktop\python'	
Ground truth:	os.chdir('c:\Users\uname\desktop\python')
CodeT5:	os.system('c:\Users\uname\desktop\python')
CodeT5+DocPrompting:	os.chdir('c:\Users\uname\desktop\python')
convert dataframe 'df' to integer-type sparse object	
Ground truth:	df.to_sparse(0)
CodeT5:	np.isinstance(df, np.integer)
CodeT5+DocPrompting:	df.to_sparse('i')

Chapter 8

FLARE: Generating Code by Retrieving the Docs (Completed)

Despite the remarkable ability of large language models (LMs) to comprehend and generate language, they have a tendency to hallucinate and create factually inaccurate output. Augmenting LMs by retrieving information from external knowledge resources is one promising solution. Most existing retrieval augmented LMs employ a retrieve-and-generate setup that only retrieves information once based on the input. This is limiting, however, in more general scenarios involving generation of long texts, where continually gathering information throughout generation is essential. In this work, we provide a generalized view of *active retrieval augmented generation*, methods that actively decide when and what to retrieve across the course of the generation. We propose **F**orward-**L**ooking **A**ctive **RE**trieval augmented generation (**FLARE**), a generic method which iteratively uses a prediction of the upcoming sentence to anticipate future content, which is then utilized as a query to retrieve relevant documents to regenerate the sentence if it contains low-confidence tokens. We test FLARE along with baselines comprehensively over 4 long-form knowledge-intensive generation tasks/datasets. FLARE achieves superior or competitive performance on all tasks, demonstrating the effectiveness of our method. Code and datasets are available at https://github.com/jzbjyb/FLARE.

8.1 Introduction

Generative language models (LMs) [48, 59, 260, 262, 338, 399, 400] have become a foundational component in natural language processing (NLP) systems with their remarkable abilities. Although LMs have memorized some world knowledge during training [147, 271, 301], they still

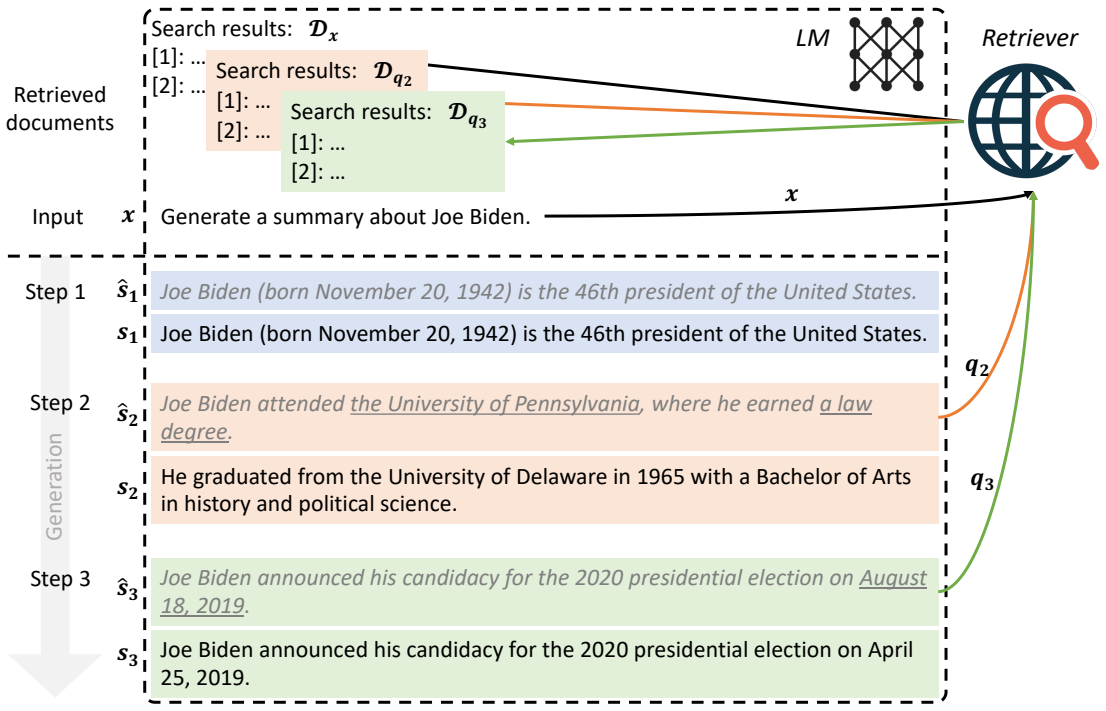


Figure 8.1: An illustration of forward-looking active retrieval augmented generation (FLARE). Starting with the user input $\boldsymbol{x}$ and initial retrieval results $\mathcal{D}_{\boldsymbol{x}}$, FLARE iteratively generates a temporary next sentence (shown in *gray italic*) and check whether it contains low-probability tokens (indicated with underline). If so (step 2 and 3), the system retrieves relevant documents and regenerates the sentence.

tend to hallucinate and create imaginary content [228, 407]. Augmenting LMs with retrieval components that look up relevant information from external knowledge resources is a promising direction to address hallucination [144, 167].

Retrieval augmented LMs commonly use a retrieve-and-generate setup where they retrieve documents based on the user's input, and then generate a complete answer conditioning on the retrieved documents [54, 112, 142, 144, 149, 186, 189, 195, 248, 280, 308, 320]. These single-time retrieval augmented LMs outperform purely parametric LMs, particularly for short-form knowledge-intensive generation tasks such as factoid question answering (QA) [153, 184], where *the information needs are clear in the user's input, and it is sufficient to retrieve relevant knowledge once solely based on the input.*

Increasingly powerful large LMs have also demonstrated abilities in more complex tasks that involve generating long-form output, such as long-form QA [80, 333], open-domain summarization [63, 98, 117], and (chain-of-thought; CoT) reasoning [96, 129, 133, 357]. In contrast

to short-form generation, long-form generation presents complex information needs that are *not always evident from the input alone*. Similar to how humans gradually gather information as we create content such as papers, essays, or books, long-form generation with LMs would *require gathering multiple pieces of knowledge throughout the generation process*. For example, to generate a summary about a particular topic, the initial retrieval based on the topic name (e.g., Joe Biden) may not cover all aspects and details. It is crucial to retrieve extra information as needed during generation, such as when generating a certain aspect (e.g., Joe Biden's education history) or a specific detail (e.g., the date of Joe Biden's presidential campaign announcement).

Several attempts have been made to retrieve multiple times throughout generation. These attempts include methods that passively use the past context to retrieve additional information at a fixed interval [37, 167, 295, 342] which might not accurately reflect what LMs intend to generate in the future or retrieve at inappropriate points. Some works in multihop QA decompose the full question into sub-questions, each of which is used to retrieve extra information [168, 169, 275, 372].

We ask the following question: can we create a simple and generic retrieval augmented LM that *actively decides when and what to retrieve* throughout the generation process, and are applicable to a variety of long-form generation tasks? We provide a generalized view of active retrieval augmented generation. Our hypothesis regarding *when to retrieve* is that LMs should retrieve information only when they lack the required knowledge to avoid unnecessary or inappropriate retrieval that occurs in passive retrieval augmented LMs [37, 167, 295, 342]. Given the observation that large LMs tend to be well-calibrated and low probability/confidence often indicates a lack of knowledge [155], we adopt an active retrieval strategy that only retrieves when LMs generate low-probability tokens. When deciding *what to retrieve*, it is important to consider what LMs intend to generate in the future, as the goal of active retrieval is to benefit future generations. Therefore, we propose anticipating the future by generating a temporary next sentence, using it as a query to retrieve relevant documents, and then regenerating the next sentence conditioning on the retrieved documents. Combining the two aspects, we propose **F**orward-**L**ooking **A**ctive **RE**trieval augmented generation (**FLARE**), as illustrated in Figure 8.1. FLARE iteratively generates *a temporary next sentence*, use it as the query to retrieve relevant documents *if it contains low-probability tokens* and regenerate the next sentence until reaches the end.

FLARE is applicable to any existing LMs at inference time without additional training. Considering the impressive performance achieved by GPT-3.5 [262] on a variety of tasks, we examine the effectiveness of our methods on `text-davinci-003`. We evaluate FLARE on 4

diverse tasks/datasets involving generating long outputs, including multihop QA (2WikiMultihopQA), commonsense reasoning (StrategyQA), long-form QA (ASQA), and open-domain summarization (WikiAsp) [96, 117, 133, 333]. Over all tasks, FLARE achieves superior or competitive performance compared to single-time and multi-time retrieval baselines, demonstrating the effectiveness and generalizability of our method.

8.2 Retrieval Augmented Generation

We formally define single-time retrieval augmented generation and propose the framework of active retrieval augmented generation.

8.2.1 Notations and Definitions

Given a user input $\boldsymbol{x}$ and a document corpus $\mathcal{D} = \{\boldsymbol{d}_i\}_{i=1}^{|\mathcal{D}|}$ (such as all Wikipedia articles), the goal of retrieval augmented LMs is to generate the answer $\boldsymbol{y} = [\boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_m] = [w_1, w_2, ..., w_n]$ containing m sentences or n tokens leveraging information retrieved from the corpus.

In retrieval augmented LM, the LM typically pairs with a retriever that can retrieve a list of documents $\mathcal{D}_{\boldsymbol{q}} = \text{ret}(\boldsymbol{q})$ for a query $\boldsymbol{q}$; the LM conditions on both the user input $\boldsymbol{x}$ and retrieved documents $\mathcal{D}_{\boldsymbol{q}}$ to generate the answer. Since we focus on examining various methods of determining when and what to retrieve, we follow existing methods [295, 342] to prepend the retrieved documents before the user input to aid future generation for both baselines and our method for fair comparisons: $\boldsymbol{y} = \text{LM}([\mathcal{D}_{\boldsymbol{q}}, \boldsymbol{x}])$, where $[\cdot, \cdot]$ is concatenation following the specified order.

8.2.2 Single-time Retrieval Augmented Generation

The most common choice is to directly use the user input as the query for retrieval and generate the complete answer at once $\boldsymbol{y} = \text{LM}([\mathcal{D}_{\boldsymbol{x}}, \boldsymbol{x}])$.

8.2.3 Active Retrieval Augmented Generation

To aid long-form generation with retrieval, we propose active retrieval augmented generation. It is a generic framework that actively decides when and what to retrieve through the generation process, resulting in the interleaving of retrieval and generation. Formally, at step $t (t \geq 1)$, the

retrieval query $\boldsymbol{q}_t$ is formulated based on both the user input $\boldsymbol{x}$ and previously generated output $\boldsymbol{y}_{<t} = [\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{t-1}]$:

$$\boldsymbol{q}_t = \text{qry}(\boldsymbol{x}, \boldsymbol{y}_{<t}),$$

where $\text{qry}(\cdot)$ is the query formulation function. At the beginning ($t = 1$), the previous generation is empty ($\boldsymbol{y}_{<1} = \varnothing$), and the user input is used as the initial query ($\boldsymbol{q}_1 = \boldsymbol{x}$). Given retrieved documents $\mathcal{D}_{\boldsymbol{q}_t}$, LMs continually generate the answer until the next retrieval is triggered or reaches the end:

$$\boldsymbol{y}_t = \text{LM}([\mathcal{D}_{\boldsymbol{q}_t}, \boldsymbol{x}, \boldsymbol{y}_{<t}]),$$

where $\boldsymbol{y}_t$ represents the generated tokens at the current step t, and the input to LMs is the concatenation of the retrieved documents $\mathcal{D}_{\boldsymbol{q}_t}$, the user input $\boldsymbol{x}$, and the previous generation $\boldsymbol{y}_{<t}$. We discard previously retrieved documents $\cup_{t'<t}\mathcal{D}_{\boldsymbol{q}_{t'}}$ and only use the retrieved documents from the current step to condition the next generation to prevent reaching the input length limit of LMs.

8.3 FLARE: Forward-Looking Active REtrieval Augmented Generation

Our intuition is that (1) LMs should only retrieve information when they do not have the necessary knowledge to avoid unnecessary or inappropriate retrieval, and (2) the retrieval queries should reflect the intents of future generations. We propose two forward-looking active retrieval augmented generation (FLARE) methods to implement the active retrieval augmented generation framework. The first method prompts the LM to generate retrieval queries when necessary while generating the answer using retrieval-encouraging instructions, denoted as $\text{FLARE}_{\text{instruct}}$. The second method directly uses the LM's generation as search queries, denoted as $\text{FLARE}_{\text{direct}}$, which iteratively generates the next sentence to gain insight into the future topic, and if uncertain tokens are present, retrieves relevant documents to regenerate the next sentence.

8.3.1 FLARE with Retrieval Instructions

Inspired by Toolformer [312], a straightforward way of expressing information needs for retrieval is to generate "[Search(query)]" when additional information is needed [312], e.g., "The colors on the flag of Ghana have the following meanings. Red is for [Search(Ghana flag red meaning)]

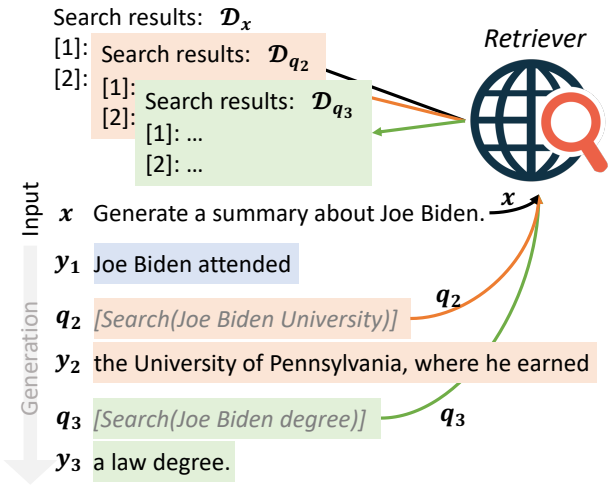


Figure 8.2: An illustration of forward-looking active retrieval augmented generation with retrieval instructions ($\text{FLARE}_{\text{instruct}}$). It iteratively generates search queries (shown in *gray italic*) to retrieve relevant information to aid future generations.

the blood of martyrs, ...” When working with GPT-3.5 models that offer only API access, we elicit such behavior by few-shot prompting [48].

Specifically, for a downstream task, we place the search-related instruction and exemplars at the beginning as skill 1, followed by the instruction and exemplars of the downstream task as skill 2. Given a test case, we ask LMs to combine skills 1 and 2 to generate search queries while performing the task. The structure of the prompt is shown in Prompt 8.3.1.

Prompt 8.3.1: retrieval instructions

Skill 1. An instruction to guide LMs to generate search queries.
Several search-related exemplars.

Skill 2. An instruction to guide LMs to perform a specific downstream task (e.g., multihop QA).
Several task-related exemplars.

An instruction to guide LMs to combine skills 1 and 2 for the test case.
The input of the test case.

As shown in Figure 8.2, when the LM generates “[Search(query)]” (shown in *gray italic*), we stop the generation and use the query terms to retrieve relevant documents, which are prepended before the user input to aid future generation until the next search query is generated or reaches the end. Additional implementation details are included in § 8.10.1.

8.3.2 Direct FLARE

Since we cannot fine-tune black-box LMs, we found queries generated by $\text{FLARE}_{\text{instruct}}$ through retrieval instructions might not be reliable. Therefore, we propose a more direct way of forward-looking active retrieval that uses the next sentence to decide when and what to retrieve.

Confidence-based Active Retrieval

As shown in Figure 8.1, at step t, we first generate a temporary next sentence $\hat{\boldsymbol{s}}_t = \text{LM}([\boldsymbol{x}, \boldsymbol{y}_{<t}])$ without conditioning on retrieved documents. Then we decide whether to trigger retrieval and formulate queries based on $\hat{\boldsymbol{s}}_t$. If the LM is confident about $\hat{\boldsymbol{s}}_t$, we accept it without retrieving additional information; if not, we use $\hat{\boldsymbol{s}}_t$ to formulate search queries $\boldsymbol{q}_t$ to retrieve relevant documents, and then regenerate the next sentence $\boldsymbol{s}_t$. The reason we utilize sentences as the basis of our iteration is due to their significance as semantic units that are neither too short nor too lengthy like phrases and paragraphs. However, our approach can also utilize phrases or paragraphs as the basis.

Since LMs tend to be well-calibrated that low probability/confidence often indicates a lack of knowledge [148, 155, 346], we actively trigger retrieval if any token of $\hat{\boldsymbol{s}}_t$ has a probability lower than a threshold $\theta \in [0, 1]$. $\theta = 0$ means retrieval is never triggered, while $\theta = 1$ triggers retrieval every sentence.

$$\boldsymbol{y}_t = \begin{cases} \hat{\boldsymbol{s}}_t & \text{if all tokens of } \hat{\boldsymbol{s}}_t \text{ have probs} \geq \theta \\ \boldsymbol{s}_t = \text{LM}([\mathcal{D}_{\boldsymbol{q}_t}, \boldsymbol{x}, \boldsymbol{y}_{<t}]) & \text{otherwise} \end{cases}$$

where the query $\boldsymbol{q}_t$ is formulated based on $\hat{\boldsymbol{s}}_t$.

Confidence-based Query Formulation

One way to perform retrieval is to directly use the next sentence $\hat{\boldsymbol{s}}_t$ as the query $\boldsymbol{q}_t$. This shares a similar spirit with methods that use generated hypothetical titles or paragraphs from LMs as retrieval queries or evidences [92, 227, 336, 392]. We generalize such techniques to long-form generation where active information access is essential.

We found retrieving with the next sentence achieves significantly better results than with the previous context, as shown later in § 8.6.2. However, it has a risk of perpetuating errors contained in it. For example, if the LM produces the sentence "Joe Biden attended the University of Pennsylvania" instead of the correct fact that he attended the University of Delaware, using

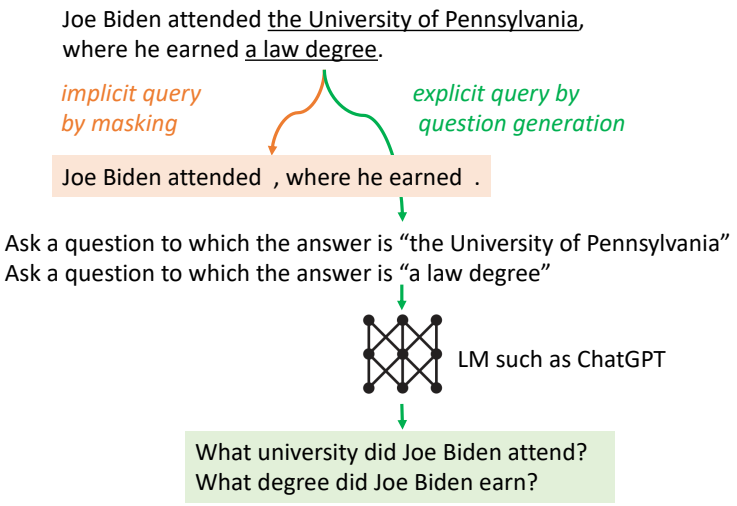


Figure 8.3: Implicit and explicit query formulation. Tokens with low probabilities are marked with underlines.

this erroneous sentence as a query might retrieve misleading information. We propose two simple methods to overcome this issue as illustrated in Figure 8.3.

Masked sentences as implicit queries. The first method masks out low-confidence tokens in $\hat{\boldsymbol{s}}_t$ with probabilities below a threshold $\beta \in [0, 1]$, where a higher β results in more aggressive masking. This removes potential distractions from the sentence to improve retrieval accuracy.

Generated questions as explicit queries. Another method is to generate explicit questions that target the low-confident span in $\hat{\boldsymbol{s}}_t$. For example, if the LM is uncertain about "the University of Pennsylvania", a question like "Which university did Joe Biden attend?" can help retrieve relevant information. Self-ask [275] achieved this by manually inserting follow-up questions into downstream task exemplars, which requires task-specific annotation efforts. Instead, we developed a universal approach that generates questions for low-confidence spans without additional annotation. Specifically, We first extract all spans from $\hat{\boldsymbol{s}}_t$ with probabilities below β. For each extracted span $\boldsymbol{z}$, we prompt `gpt-3.5-turbo` to generate a question $\boldsymbol{q}_{t,\boldsymbol{z}}$ that can be answered with the span:

Prompt 8.3.2: zero-shot question generation

User input $\boldsymbol{x}$.
Generated output so far $\boldsymbol{y}_{<t}$.

Given the above passage, ask a question to which the answer is the term/entity/phrase "$\boldsymbol{z}$".

We retrieve using each generated question and interleave the returned documents into a single ranking list to aid future generations. In summary, queries $\boldsymbol{q}_t$ are formulated based on $\hat{\boldsymbol{s}}_t$ as follows:

$$\boldsymbol{q}_t = \begin{cases} \varnothing & \text{if all tokens of } \hat{\boldsymbol{s}}_t \text{ have probs} \geq \theta \\ \text{mask}(\hat{\boldsymbol{s}}_t) \text{ or qgen}(\hat{\boldsymbol{s}}_t) & \text{otherwise} \end{cases}$$

8.3.3 Implementation Details

Base LM We validate our method on one of the most advanced GPT-3.5 LMs `text-davinci-003` by iteratively querying their API.[1]

Document corpus and retrievers. Since we focus on the integration of retrieval and generation, we use off-the-shelf retrievers that take queries as inputs and return a list of relevant documents. For datasets that mainly rely on knowledge from Wikipedia, we use the Wikipedia dump from Karpukhin et al. [160] and employ BM25 [303] as the retriever. For datasets that rely on knowledge from the open web, we use the Bing search engine as our retriever.[2]

Retrieved document formatting. Multiple retrieved documents are linearized according to their ranking and then added to the beginning of the user input.

Other implementation details such as sentence tokenization and efficiency are included § 8.10.1.

8.4 Multi-time Retrieval Baselines

Existing passive multi-time retrieval augmented LMs can also be formulated using our framework (§ 8.2.3). In this section, we formally introduce three baseline categories based on when and what to retrieve. These baselines are not exact reproductions of the corresponding paper because

[1] `https://api.openai.com/v1/completions` April 23.
[2] `https://www.microsoft.com/en-us/bing/apis/bing-web-search-api`

many design choices differ which makes direct comparisons impossible. We implemented them using the same settings, with the only variation being when and what to retrieve.

Previous-window approaches trigger retrieval every l tokens, where l represents the window size. Generated tokens from the previous window are used as the query:

$$\boldsymbol{q}_t = \boldsymbol{y}_{t-1} \quad (t \geq 2),$$
$$\boldsymbol{y}_t = [w_{(t-1)l+1}, ..., w_{tl}].$$

Some existing methods in this category are RETRO [37], IC-RALM [295], which retrieve every few tokens, and KNN-LM [167], which retrieves every token.[3] We follow Ram et al. [295] to use a window size of $l = 16$.

Previous-sentence approaches trigger retrieval every sentence and use the previous sentence as the query, and IRCoT [342] belongs to this category:

$$\boldsymbol{q}_t = \boldsymbol{y}_{t-1} \quad (t \geq 2),$$
$$\boldsymbol{y}_t = \boldsymbol{s}_t.$$

Question decomposition approaches manually annotated task-specific exemplars to guide LMs to generate decomposed sub-questions while producing outputs. For example, self-ask [275], a method in this category, manually inserts sub-questions in exemplars. For the test case, retrieval is triggered dynamically whenever the model generates a sub-question.

The aforementioned approaches can retrieve additional information while generating. However, they have notable drawbacks: (1) Using previously generated tokens as queries might not reflect what LMs intend to generate in the future. (2) Retrieving information at a fixed interval can be inefficient because it might occur at inappropriate points. (3) Question decomposition approaches require task-specific prompt engineering, which restricts their generalizability in new tasks.

8.5 Experimental Setup

We evaluate the effectiveness of FLARE on 4 diverse knowledge-intensive tasks using few-shot in-context learning [48, 215, 286]. We follow previous works [342] to sub-sample at most 500

[3] Since KNN-LM uses the contextualized representation corresponding to the current decoding position to retrieve relevant information which encodes all previous tokens. Strictly speaking, $\boldsymbol{q}_t$ should be $\boldsymbol{y}_{<t}$.

examples from each dataset due to the cost of running experiments. Datasets, metrics, and settings are summarized in Table 8.7 of § 8.10.2. The hyperparameters of FLARE are selected based on the development set. FLARE refers to $\text{FLARE}_{\text{direct}}$ if not specifically stated.

Multihop QA The goal of multihop QA is to answer complex questions through information retrieval and reasoning. We use 2WikiMultihopQA [133] which contains 2-hop complex questions sourced from Wikipedia articles that require composition, comparison, or inference, e.g., "Why did the founder of Versus die?" We follow Wang et al. [353] to generate both the chain-of-thought and the final answer. Experimental setting details are included in § 8.10.2.

We use regular expressions to extract the final answer from the output and compare it with the reference answer using exact match (EM), and token-level F_1, precision, and recall.

Commonsense reasoning Commonsense reasoning requires world and commonsense knowledge to generate answers. We use StrategyQA [96] which is a collection of crowdsourced yes/no questions, e.g., "Would a pear sink in water?" We follow Wei et al. [357] to generate both the chain-of-thought and the final yes/no answer. Details are included in § 8.10.2.

We extract the final answer and match it against the gold answer using exact match.

Long-form QA Long-form QA aims to generate comprehensive answers to questions seeking complex information [80, 333]. We use ASQA [333] as our testbed where inputs are ambiguous questions with multiple interpretations, and outputs should cover all of them. For example, "Where do the Philadelphia Eagles play their home games?" could be asking about the city, sports complex, or stadium. We found in many cases it is challenging even for humans to identify which aspect of the question is ambiguous. Therefore, we created another setting (ASQA-hint) where we provide a brief hint to guide LMs to stay on track when generating answers. The hint for the above case is "This question is ambiguous in terms of which specific location or venue is being referred to." Experimental setting details are included in § 8.10.2.

We use metrics from Stelmakh et al. [333], including EM, RoBERTa-based QA score (Disambig-F_1), ROUGE [208], and an overall score combining Disambig-F_1 and ROUGE (DR).

Open-domain summarization The goal of open-domain summarization is to generate a comprehensive summary about a topic by gathering information from open web [98]. We use WikiAsp [117] which aims to generate aspect-based summaries about entities from 20 domains

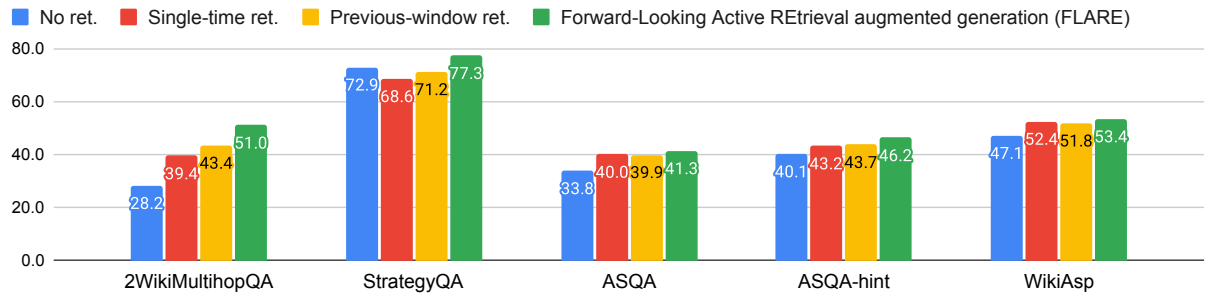


Figure 8.4: Comparision between FLARE and baselines across all tasks/datasets. We report the primary metric for each dataset: EM for 2WikiMultihopQA, StrategyQA, and ASQA, and UniEval for WikiAsp.

in Wikipedia, e.g., "Generate a summary about Echo School (Oregon) including the following aspects: academics, history." Experimental setting details are included in § 8.10.2.

Metrics include ROUGE, named entity-based F_1, and UniEval [401] which measures factual consistency.

8.6 Experimental Results

We first report overall results across 4 tasks/datasets and compare the performance of FLARE with all the baselines introduced in § 8.4. We then run ablation experiments to study the efficacy of various design choices of our method.

8.6.1 Comparison with Baselines

Overall results. The overall performance of FLARE and baseline across all tasks/datasets are reported in Figure 8.4. FLARE outperforms all baseline on all tasks/datasets, indicating that FLARE is a generic method that can effectively retrieve additional information throughout the generation.

Among various tasks, multihop QA shows the most significant improvement. This is largely due to the task's clear definition and specific objective of producing the final answer through a 2-hop reasoning process, which makes it easier for LMs to generate on-topic output. In contrast, ASQA and WikiAsp are more open-ended, which increases the difficulty of both generation and evaluation. The improvement on ASQA-hint is larger than that of ASQA because identifying ambiguous aspects is challenging even for humans in many cases, and providing a generic hint

Methods	EM	F_1	Prec.	Rec.
No retrieval	28.2	36.8	36.5	38.6
Single-time retrieval	39.4	48.8	48.6	51.5
Multi-time retrieval				
Previous-window	43.2	52.3	51.7	54.5
Previous-sentence	39.0	49.2	48.9	51.8
Question decomposition	47.8	56.4	56.1	58.6
$\text{FLARE}_{\text{instruct}}$ (ours)	42.4	49.8	49.1	52.5
$\text{FLARE}_{\text{direct}}$ (ours)	**51.0**	**59.7**	**59.1**	**62.6**

Table 8.1: FLARE and baselines on 2WikiMultihopQA. Previous-window [37, 295], previous-sentence [342], and question decomposition [275, 372] methods are reimplemented for fair comparisons.

helps LMs to stay on topic.

Thorough comparisons with baselines. The performance of all baselines on 2WikiMultihopQA are reported in Table 8.1. FLARE outperforms all baselines by a large margin, which confirms that forward-looking active retrieval is highly effective. Most multi-time retrieval augmented approaches outperform single-time retrieval but with different margins. The improvement of retrieving using the previous sentence is relatively small which we hypothesize is mainly because the previous sentence often describes entities or relations different from those in the next sentence in 2WikiMultihopQA. While the previous-window approach might use the first half of a sentence to retrieve information potentially helpful for generating the second half. Among all baselines, the question decomposition approach [275] achieves the best performance. which is not surprising since the in-context exemplars manually annotated with decomposed sub-questions guide LMs to generate sub-questions that align with the topic/intent of future generations. FLARE outperforms this baseline, indicating that manual exemplar annotation is not necessary for effective future-aware retrieval. The gap between $\text{FLARE}_{\text{instruct}}$ and question decomposition is large, indicating that teaching LMs to generate search queries using task-generic retrieval instructions and exemplars is challenging.

We report all metrics for the other datasets in Table 8.2. FLARE outperforms baselines with respect to all metrics. Retrieval using the previous window underperforms single-time retrieval on ASQA, which we hypothesize is because the previous window does not accurately reflect

Datasets	**StrategyQA**	**ASQA**				**ASQA-hint**				**WikiAsp**		
Metrics	**EM**	**EM**	**D-F$_1$**	**R-L**	**DR**	**EM**	**D-F$_1$**	**R-L**	**DR**	**UniEval**	**E-F$_1$**	**R-L**
No retrieval	72.9	33.8	24.2	33.3	28.4	40.1	32.5	36.4	34.4	47.1	14.1	26.4
Single-time retrieval	68.6	40.0	27.1	34.0	30.4	43.2	34.8	37.4	36.0	52.4	17.4	26.9
Multi-time retrieval												
Previous-window	71.2	39.9	27.0	**34.3**	30.4	43.7	35.7	37.5	36.6	51.8	18.1	27.3
Previous-sentence	71.0	39.9	27.9	**34.3**	30.9	44.7	35.9	37.5	36.7	52.6	17.8	27.2
FLARE (ours)	**77.3**	**41.3**	**28.2**	**34.3**	**31.1**	**46.2**	**36.7**	**37.7**	**37.2**	**53.4**	**18.9**	**27.6**

Table 8.2: Comparison between FLARE and baselines on StrategyQA, ASQA, ASQA-hint, and WikiAsp. D-F$_1$ is Disambig-F$_1$, R-L is ROUGE-L, and E-F$_1$ is named entity-based F$_1$.

	2WikiMultihopQA				**ASQA-hint**			
	EM	**F$_1$**	**Prec.**	**Rec.**	**EM**	**D-F$_1$**	**R-L**	**DR**
Previous	39.0	49.2	48.9	51.8	42.5	34.1	36.9	35.5
Next	48.8	57.6	57.1	60.5	45.9	35.7	37.5	36.6

Table 8.3: A head-to-head comparison between using the previous sentence and the next sentence for retrieval.

future intent. Since we focus on evaluating factuality, metrics with an emphasis on factual content (such as EM, Disambig-F$_1$, UniEval) are more reliable than metrics computed over all tokens (ROUGE-L).

8.6.2 Ablation Study

Importance of forward-looking retrieval. We first validate that forward-looking retrieval is more effective than past-context-based retrieval. We run ablation experiments on 2WikiMultihopQA and ASQA-hint comparing retrieval using the previous versus the next sentence. Specifically, both methods retrieve every sentence and directly use the complete previous/next sentence as queries. As shown in Table 8.3, using the next sentence to retrieve is clearly better than using the previous sentence, confirming our hypothesis.

We also run previous-window approaches using different numbers of past tokens as queries. As shown in Table 8.4, using too many tokens (> 32) in the past hurts the performance, further

#Tokens	EM	F_1	Prec.	Rec.
16	43.2	52.3	51.7	54.5
32	43.6	52.4	52.0	55.0
48	40.0	49.3	49.0	52.0
All	39.0	48.5	48.2	51.1

Table 8.4: Previous-window approaches using different numbers of tokens as queries.

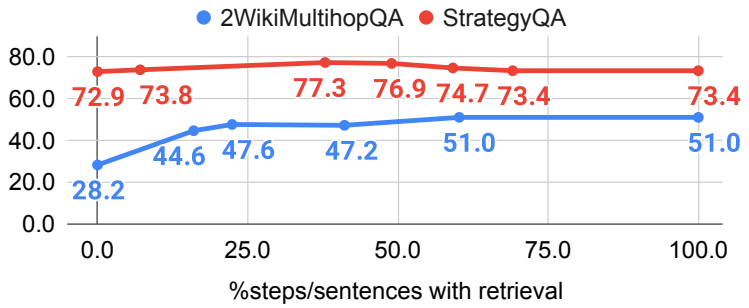


Figure 8.5: Performance (EM) of FLARE with respect to the percentage of steps/sentences with retrieval on 2WikiMultihopQA and StrategyQA.

confirming our hypothesis that previous context might not be relevant to intent of future generations.

Importance of active retrieval. Next, we investigate how active retrieval threshold θ affects performance. To alter our method from not retrieving to retrieving every sentence, we adjust the confidence threshold θ that determines when to trigger retrieval from 0 to 1. We then calculate the proportion of steps/sentences where retrieval is activated, and present the performance based on it. As shown in Figure 8.5, on 2WikiMultihopQA, the performance plateaus when the retrieval percentage exceeds 60%, indicating that retrieval when LMs are confident is not necessary. On StrategyQA, the performance drops when the retrieval percentage exceeds 50%, indicating that unnecessary retrieval can introduce noise and impede the original generation process. We found triggering retrieval for 40%-80% of sentences usually leads to a good performance across tasks/datasets.

Effectiveness of different query formulation methods We study implicit query formation by masking and explicit query formulation through question generation. In Table 8.5, we compare the performance of FLARE with different masking thresholds β. Retrieving directly with the

β	EM	F_1	Prec.	Rec.
0.0	0.488	0.576	0.571	0.605
0.2	0.498	0.588	0.582	0.616
0.4	0.510	0.597	0.591	0.627
0.6	0.506	0.593	0.586	0.622

Table 8.5: Performance of FLARE with respect to the masking threshold β on 2WikiMultihopQA.

	ASQA-hint				WikiAsp		
	EM	D-F_1	R-L	DR	UniEval	E-F_1	R-L
Implicit	45.7	36.9	37.7	37.3	53.4	18.8	27.7
Explicit	46.2	36.7	37.7	37.2	53.4	18.9	27.6

Table 8.6: A comparison between implicit and explicit query formulation methods in FLARE.

complete sentence ($\beta = 0$) is worse than masking tokens with low probabilities, confirming our hypothesis that low-confidence erroneous tokens can distract retrievers. We compare implicit and explicit query formulation methods in Table 8.6. Performances of both methods are similar, indicating that both methods can effectively reflect information needs.

8.7 Related Work

We refer to § 8.2.2 and § 8.4 for extensively discussion on single-time and multi-time retrieval augmented LMs, which is the most relevant area to this paper.

Iterative and adaptive retrieval Iterative retrieval and refinement has been studied in both text and code generation tasks [269, 393, 396, 398]. FLARE differs from these methods in the granularity of generation and retrieval strategies. Adaptive retrieval has been studied in single-time retrieval scenarios based on either question popularity or generation probabilities [197, 223], while we focus on long-form generation requiring active information access.

Browser-enhanced LMs WebGPT [248] and WebCPM [281] train LMs to interact with browser to enhance factuality using reinforcement learning or supervised training where multiple queries can be triggered before generation. FLARE is built on text-based retrievers but can be

combined with a browser to potentially improve retrieval quality.

8.8 Conclusion

To aid long-form generation with retrieval augmentation, we propose an active retrieval augmented generation framework that decides when and what to retrieve during generation. We implement this framework with forward-looking active retrieval that iteratively uses the upcoming sentence to retrieve relevant information if it contains low-confidence tokens and regenerates the next sentence. Experimental results on 4 tasks/datasets demonstrate the effectiveness of our methods. Future directions include better strategies for active retrieval and developing efficient LM architectures for active information integration.

8.9 Limitations

We also conduct experiments on Wizard of Wikipedia [75] and ELI5 [80], and found that FLARE did not provide significant gains. Wizard of Wikipedia is a knowledge-intensive dialogue generation dataset where the output is relatively short (~20 tokens on average) so retrieving multiple disparate pieces of information might not be necessary. ELI5 [80] is a long-form QA dataset requiring in-depth answers to open-ended questions. Due to issues mentioned in Krishna et al. [178] such as difficulties of grounding generation in retrieval and evaluation, both single-time retrieval and FLARE did not provide significant gains over not using retrieval. From an engineering perspective, interleaving generation and retrieval with a naive implementation increases both overheads and the cost of generation. LMs need to be activated multiple times (once for each retrieval) and a caching-free implementation also requires recomputing the previous activation each time after retrieval. This issue can be potentially alleviated with special architectural designs that encode the retrieved documents $\mathcal{D}_{\boldsymbol{q}_t}$ and the input/generation ($\boldsymbol{x}/\boldsymbol{y}_{<t}$) independently.

8.10 Appendix

8.10.1 FLARE Implementation Details

FLARE$_{\text{instruct}}$ implementation details We found that LMs can effectively combine retrieval and downstream task-related skills and generate meaningful search queries while performing

the task. However, there are two issues: (1) LMs tend to generate fewer search queries than necessary. (2) Generating excessive search queries can disrupt answer generation and adversely affect performance. We address these issues using two methods respectively. First, we increase the logit of the token "[" by 2.0 to improve the chances of LMs generating "[Search(query)]". Second, whenever LMs generate a search query, we use it to retrieve relevant information, promptly remove it from the generation, and generate the next few tokens while forbidding "[" by adding a large negative value to the logit of "[".

The initial query of FLARE. FLARE starts with the user input $\boldsymbol{x}$ as the initial query to retrieve documents to generate the first sentence $\hat{\boldsymbol{s}}_1 = \text{LM}([\mathcal{D}_{\boldsymbol{x}}, \boldsymbol{x}])$ to bootstrap the iterative generation process. For the following steps, the temporary forward-looking sentence is generated without retrieved documents.

Sentence tokenization. For each step t, we generate 64 tokens which are longer than most sentences, and use NLTK sentence tokenizer[4] to extract the first sentence and discard the rest.

Efficiency As shown in § 8.6.2, on average retrieval is triggered for $30\% \sim 60\%$ of sentences depending on downstream tasks. In comparision, KNN-LM [167] retrieves every token, RETRO or IC-RALM [37, 295] retrievers every 4~32 tokens, and IRCoT [342] retrieves every sentence. Compared to single-time retrieval, however, interleaving retrieval and generation with a naive implementation indeed increases overheads, which we discuss in the limitation section (§ 8.9).

8.10.2 Datasets and Settings

Datasets, metrics, and experimental settings are summarized in Table 8.7.

Multihop QA For "Why did the founder of Versus die?", the output we aim to generate is "The founder of Versus was Gianni Versace. Gianni Versace was shot and killed on the steps of his Miami Beach mansion on July 15, 1997. So the answer is shot." We use 8 exemplars from Trivedi et al. [342] for in-context learning, BM25 as the retriever, and Wikipedia articles as the retrieval corpus. Similar to the observation in Trivedi et al. [342], we found incorporating retrieval results for exemplars improves the performance, we use the input $\boldsymbol{x}$ of each exemplar to retrieve several documents and then add them using the prompting format. We found increasing the number

[4] https://www.nltk.org/api/nltk.tokenize.PunktSentenceTokenizer.html

of retrieval documents often increases performance. Therefore, we use the maximum number of documents that can fit within the input length limit of `text-davinci-003`, which is 2 for 2WikiMultihopQA.

Commonsense Reasoning For "Would a pear sink in water?", the output we aim to generate is "The density of a pear is about 0.6g/cm^3, which is less than water. Objects less dense than water float. Thus, a pear would float. So the final answer is no." We use 6 exemplars from Wei et al. [357], BM25 on the Wikipedia corpus, and 3 retrieved documents to run experiments.

Long-form QA For "Where do the Philadelphia Eagles play their home games?", the output we aim to generate is "We need to consider the different possible locations or venues that could be considered the home field of the Philadelphia Eagles. These include the city, the sports complex, or the stadium. Therefore, this question has 3 interpretations and the answers are: (1) The city is Philadelphia. (2) The sports complex is the South Philadelphia Sports Complex. (3) The stadium is the Lincoln Financial Field stadium." For both the original setting (ASQA) and the setting with hints (ASQA-hint), we manually annotate 8 exemplars, use BM25 on the Wikipedia corpus, and 3 retrieved documents to run experiments.

Open-domain Summarization The original WikiAsp dataset is designed for multi-document summarization and provides a list of references to systems. We converted it into the open-domain setting by removing the associated references and instead gathering information from the open web. For "Generate a summary about Echo School (Oregon) including the following aspects: academics, history.", the output we aim to generate is "# Academics. In 2008, 91% of the school's seniors received their high school diploma... # History. The class of 2008 was the 100th class in the school's history." where # is used to indicate aspects. We manually annotate 4 exemplars, and use the Bing search engine to retrieve 5 documents from the open web. To avoid leaking, we exclude several Wikipedia-related domains listed in Table 8.8 from Bing's search results.

Settings	**2WikiMultihopQA** [133]	**StrategyQA** [96]	**ASQA** [333]	**WikiAsp** [117]
		Dataset statistics		
Task	multihop QA	commonsense QA	long-form QA	open-domain summarizatio
#Examples	500	229	500	500
		Evaluation settings		
Metrics	EM, F_1, Prec., Rec.	EM	EM, Disambig-F_1, ROUGE, DR	UniEval, entity-F_1, ROUGE
		Retrieval settings		
Corpus	Wikipedia	Wikipedia	Wikipedia	open web
Retriever	BM25	BM25	BM25	Bing
Top-k	2	3	3	5
		Prompt format		
#Exemplars	8	6	8	4
Ret. for exemplars	♣	♠	♠	♠

Table 8.7: Dataset statistics and experimental settings of different tasks.

wikipedia.org, wikiwand.com, wiki2.org, wikimedia.org

Table 8.8: Wikipedia-related domains excluded from Bing's search results.

Part IV

Iterative Use of LLMs as Agents

Chapter 9

WebArena: A Realistic Web Environment for Building Autonomous Agents (Completed)

With advances in generative AI, there is now potential for autonomous agents to manage daily tasks via natural language commands. However, current agents are primarily created and tested in simplified synthetic environments, leading to a disconnect with real-world scenarios. In this chapter, we build an environment for language-guided agents that is *highly realistic* and *reproducible*. Specifically, we focus on agents that perform tasks on the web, and create an environment with fully functional websites from four common domains: e-commerce, social forum discussions, collaborative software development, and content management. Our environment is enriched with tools (*e.g.*, a map) and external knowledge bases (*e.g.*, user manuals) to encourage human-like task-solving. Building upon our environment, we release a set of benchmark tasks focusing on evaluating the *functional correctness* of task completions. The tasks in our benchmark are diverse, long-horizon, and designed to emulate tasks that humans routinely perform on the internet. We experiment with several baseline agents, integrating recent techniques such as reasoning before acting. The results demonstrate that solving complex tasks is challenging: our best GPT-4-based agent only achieves an end-to-end task success rate of 14.41%, significantly lower than the human performance of 78.24%. These results highlight the need for further development of robust agents, that current state-of-the-art large language models are far from perfect performance in these real-life tasks, and that `WebArena` can be used to measure such progress.

Our code, data, environment reproduction resources, and video demonstrations are publicly available at https://webarena.dev/.

9.1 Introduction

Autonomous agents that perform everyday tasks via human natural language commands could significantly augment human capabilities, improve efficiency, and increase accessibility. Nonetheless, to fully leverage the power of autonomous agents, it is crucial to understand their behavior within an environment that is both *authentic* and *reproducible*. This will allow measurement of the ability of agents on tasks that human users care about in a fair and consistent manner.

Current environments for evaluate agents tend to *over-simplify* real-world situations. As a result, the functionality of many environments is a limited version of their real-world counterparts, leading to a lack of task diversity [15, 99, 237, 319, 323, 324, 371]. In addition, these simplifications often lower the complexity of tasks as compared to their execution in the real world [279, 323, 371]. Finally, some environments are presented as a static resource [71, 319] where agents are confined to accessing only those states that were previously cached during data collection, thus limiting the breadth and diversity of exploration. For evaluation, many environments focus on comparing the textual *surface form* of the predicted action sequences with reference action sequences, disregarding the *functional correctness* of the executions and possible alternative solutions [71, 146, 204, 279, 366]. These limitations often result in a discrepancy between simulated environments and the real world, and can potentially impact the generalizability of AI agents to successfully understand, adapt, and operate within complex real-world situations.

We introduce WebArena, a *realistic* and *reproducible* web environment designed to facilitate the development of autonomous agents capable of executing tasks (§9.2). An overview of WebArena is in Figure 9.1. Our environment comprises four fully operational, self-hosted web applications, each representing a distinct domain prevalent on the internet: online shopping, discussion forums, collaborative development, and business content management. Furthermore, WebArena incorporates several utility tools, such as map, calculator, and scratchpad, to best support possible human-like task executions. Lastly, WebArena is complemented by an extensive collection of documentation and knowledge bases that vary from general resources like English Wikipedia to more domain-specific references, such as manuals for using the integrated development tool [81]. The content populating these websites is extracted from their real-world counterparts, preserving the authenticity of the content served on each platform. We deliver the

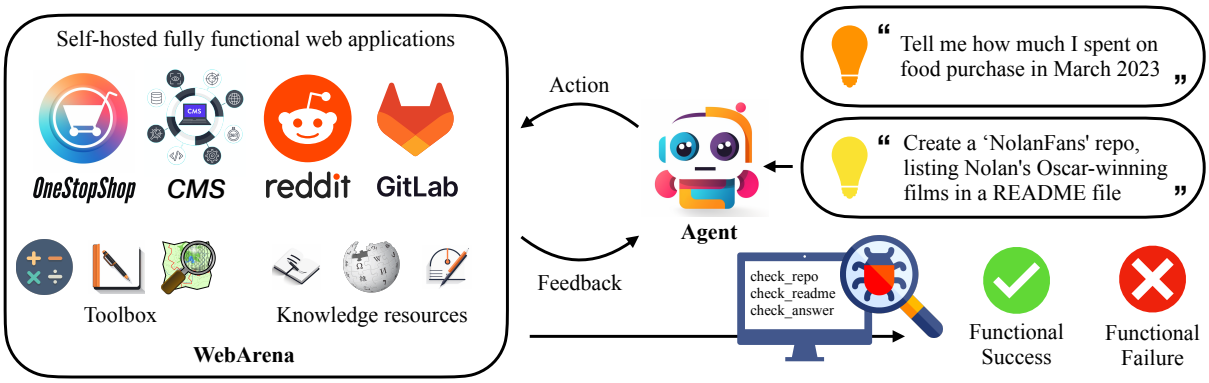


Figure 9.1: WebArena is a standalone, self-hostable web environment for building autonomous agents. WebArena creates websites from four popular categories with functionality and data mimicking their real-world equivalents. To emulate human problem-solving, WebArena also embeds tools and knowledge resources as independent websites. WebArena introduces a benchmark on interpreting *high-level realistic* natural language command to concrete web-based interactions. We provide annotated programs designed to programmatically validate the functional correctness of each task.

hosting services using Docker containers with gym-APIs [45], ensuring both the usability and the reproducibility of WebArena.

Along with WebArena, we release a ready-to-use benchmark with 812 long-horizon web-based tasks (§9.3). Each task is described as a high-level natural language intent, emulating the abstract language usage patterns typically employed by humans [34]. Two example intents are shown in the upper left of Figure 9.1. We focus on evaluating the *functional correctness* of these tasks, *i.e.*, does the result of the execution actually achieve the desired goal (§9.3.2). For instance, to evaluate the example in Figure 9.2, our evaluation method verifies the concrete contents in the designated repository. This evaluation is not only more reliable [55, 356, 403] than comparing the textual surface-form action sequences [71, 279] but also accommodate a range of potential valid paths to achieve the same goal, which is a ubiquitous phenomenon in sufficiently complex tasks.

We use this benchmark to evaluate several agents that can follow NL command and perform web-based tasks (§9.4). These agents are implemented in a few-shot in-context learning fashion with powerful large language models (LLMs) such as GPT-4 and PALM-2. Experiment results show that the best GPT-4 agent performance is somewhat limited, with an end-to-end task success rate of only 14.41%, while the human performance is 78.24%. We hypothesize that the

limited performance of current LLMs stems from a lack of crucial capabilities such as active exploration and failure recovery to successfully perform complex tasks (§9.5.2). These outcomes underscore the necessity for further development towards robust and effective agents [188] in WebArena.

9.2 WebArena: Websites as an Environment for Autonomous Agents

Our goal is to create a *realistic* and *reproducible* web environment. We achieve reproducibility by making the environment standalone, without relying on live websites. This circumvents technical challenges such as bots being subject to CAPTCHAs, unpredictable content modifications, and configuration changes, which obstruct a fair comparison across different systems over time. We achieve realism by using open-source libraries that underlie many in-use sites from several popular categories and importing data to our environment from their real-world counterparts.

9.2.1 Controlling Agents through High-level Natural Language

The WebArena environment is denoted as $\mathcal{E} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T} \rangle$ with state space $\mathcal{S}$, action space $\mathcal{A}$ (§9.2.4) and observation space $\mathcal{O}$ (§9.2.3). The transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{S}$ is deterministic, and it is defined by the underlying implementation of each website in the environment. Given a task described as a natural language intent $\mathbf{i}$, an agent issues an action $a_t \in \mathcal{A}$ based on intent $\mathbf{i}$, the current observation $o_t \in \mathcal{O}$, the action history $\mathbf{a}_1^{t-1}$ and the observation history $\mathbf{o}_1^{t-1}$. Consequently, the action results in a new state $s_{t+1} \in \mathcal{S}$ and its corresponding observation $o_{t+1} \in \mathcal{O}$. We propose a reward function $r(\mathbf{a}_1^T, \mathbf{s}_1^T)$ to measure the success of a task execution, where $\mathbf{a}_1^T$ represents the sequence of actions from start to the end time step T, and $\mathbf{s}_1^T$ denotes all intermediate states. This reward function assesses if state transitions align with the expectations of the intents. For example, with an intent to place an order, it verifies whether an order has been placed. Additionally, it evaluates the accuracy of the agent's actions, such as checking the correctness of the predicted answer.

9.2.2 Website Selection

To decide which categories of websites to use, we first analyzed approximately 200 examples from the authors' actual web browser histories. Each author delved into their browsing histories,

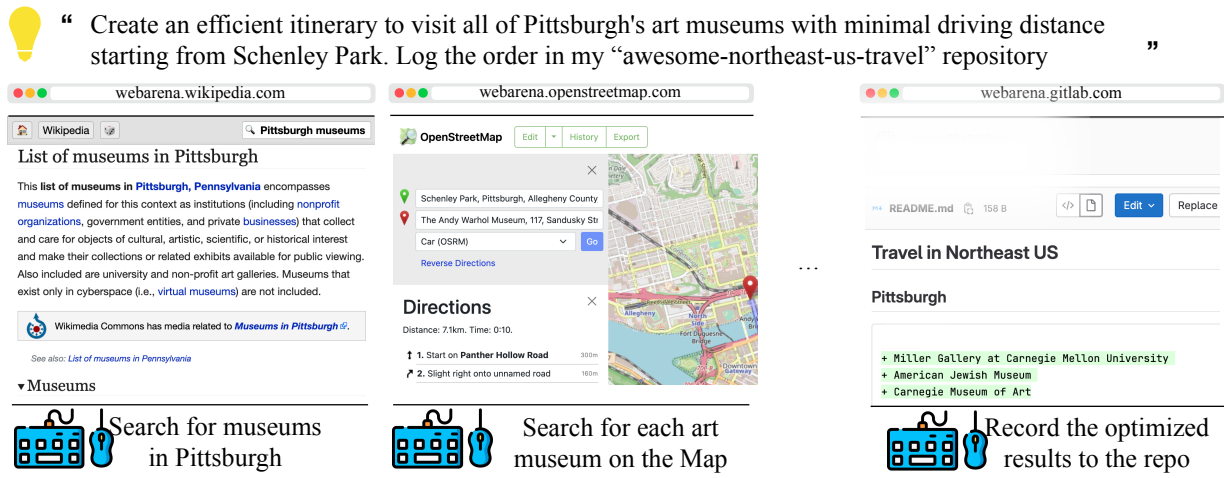


Figure 9.2: A high-level task that can be fully executed in `WebArena`. Success requires sophisticated, long-term planning and reasoning. To accomplish the goal (top), an agent needs to (1) find Pittsburgh art museums on Wikipedia, (2) identify their locations on a map (while optimizing the itinerary), and (3) update the README file in the appropriate repository with the planned route.

summarizing the goal of particular segments of their browser session. Based on this, we classified the visited websites into abstract categories. We then identified the four most salient categories and implemented one instance per category based on this analysis: (1) E-commerce platforms supporting online shopping activities (*e.g.*, Amazon, eBay), (2) social forum platforms for opinion exchanges (*e.g.*, Reddit, StackExchange), (3) collaborative development platforms for software development (*e.g.*, GitLab), and (4) content management systems (CMS) that manage the creation and revision of the digital content (*e.g.*, online store management).

In addition to these platforms, we selected three utility-style tools that are frequently used in web-based tasks: (1) a map for navigation and searching for information about points of interest (POIs) such as institutions or locations (2) a calculator, and (3) a scratchpad for taking notes. As information-seeking and knowledge acquisition are critical in web-based tasks, we also incorporated various knowledge resources into `WebArena`. These resources range from general information hubs, such as the English Wikipedia, to more specialized knowledge bases, such as the website user manuals.

Implementation We leveraged open-source libraries relevant to each category to build our own versions of an E-commerce website (OneStopShop), GitLab, Reddit, an online store content management system (CMS), a map, and an English Wikipedia. Then we imported sampled data

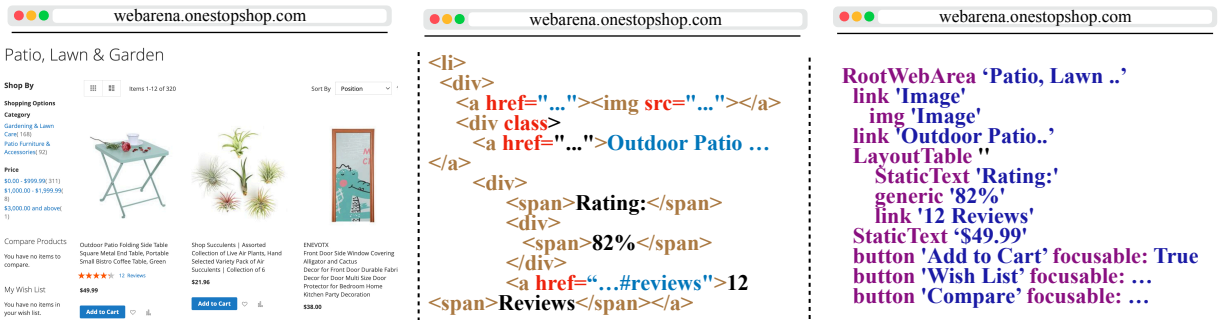


Figure 9.3: We design the observation to be the URL and the content of a web page, with options to represent the content as a screenshot (left), HTML DOM tree (middle), and accessibility tree (right). The content of the middle and right figures are trimmed to save space.

from their real-world counterparts. As an example, our version of GitLab was developed based on the actual GitLab project.[1] We carefully emulated the features of a typical code repository by including both popular projects with many issues and pull requests and smaller, personal projects. Details of all websites in WebArena can be found in Appendix 9.8.1. We deliver the environment as dockers and provide scripts to reset the environment to a deterministic initial state (See Appendix 9.8.2).

9.2.3 Observation Space

We design the observation space to roughly mimic the web browser experience: a web page URL, the opened tabs , and the web page content of the focused tab. WebArena is the first web environment to consider multi-tab web-based tasks to promote tool usage, direct comparisons and references across tabs, and other functionalities. The multi-tab functionality offers a more authentic replication of human web browsing habits compared to maintaining everything in a single tab. We provide flexible configuration to render the page content in many modes: (see Figure 9.3 for an example): (1) the raw web page HTML, composed of a Document Object Model (DOM) tree, as commonly used in past work [71, 204, 319]; (2) a screenshot, a pixel-based representation that represents the current web page as an RGB array and (3) the accessibility tree of the web page.[2] The accessibility tree is a subset of the DOM tree with elements that are *relevant* and *useful* for displaying the contents of a web page. Every element is represented as its role (*e.g.*, a link), its text content, and its properties (*e.g.*, whether it is focusable). Accessibility

[1] https://gitlab.com/gitlab-org/gitlab
[2] https://developer.mozilla.org/en-US/docs/Glossary/Accessibility_tree

trees largely retain the *structured* information of a web page while being more compact than the DOM representation.

We provide an option to limit the content to the contents within a viewport for all modes. This ensures that the observation can be input into a text-based model with limited context length or an image-based model with image size or resolution requirements.

9.2.4 Action Space

Following previous work on navigation and operation in web and embodied environments [214, 319], we design a compound action space that emulates the keyboard and mouse operations available on web pages. Figure 9.4 lists all the available actions categorized into three distinct groups. The first group includes element operations such as clicking, hovering, typing, and key combination pressing. The second comprises tab-related actions such as opening, closing, and switching between tabs. The third category consists of URL navigation actions, such as visiting a specific URL or navigating forward and backward in the browsing history.

Building on these actions, `WebArena` provides agents with the flexibility to refer to elements for operation in different ways. An element can be selected by its on-screen coordinates, (x, y), or by a unique element ID that is prepended to each element. This ID is generated when traversing the Document Object Model (DOM) or accessibility tree. With element IDs, the element selection is transformed into an n-way classification problem, thereby eliminating any disambiguation efforts required from the agent or the underlying implementation. For example, issuing the action `click [1582]` clicks the button given the observation of `[1582] Add to Cart`. This flexible element selection allows `WebArena` to support agents designed in various ways (*e.g.*, accepting input from different modalities) without compromising fair comparison metrics such as step count.

User Role Simulation Users of the same website often have disparate experiences due to their distinct *roles*, *permissions*, and *interaction histories*. We emulate this scenario by generating unique user profiles on each platform. The details can be found in Appendix 9.8.3.

9.3 Benchmark Suite of Web-based Tasks

We provide a benchmark with 812 test examples on grounding high-level natural language instructions to interactions in `WebArena`. Each example has a metric to evaluate the functional

correctness of the task execution. In this section, we first formally define the task of controlling an autonomous agent through natural language. Then we introduce the annotation process of our benchmark.

9.3.1 Intent Collection

We focus on curating *realistic* intents to carry out *complex* and *creative* tasks within `WebArena`. To start with, our annotators were guided to spend a few minutes exploring the websites to familiarize themselves with the websites' content and functionalities. As most of our websites are virtually identical to their open-web counterparts, despite having sampled data, most annotators can quickly comprehend the websites.

Next, we instructed the annotators to formulate intents based on the following criteria:

(1) The intent should be *abstract* and *high-level*, implying that the task cannot be fulfilled with merely one or two actions. As an example, instead of "*click the* `science` *subreddit*", we encouraged annotators to come up with something more complex like "*post a greeting message on* `science` *subreddit*", which involves performing multiple actions.

(2) The intent should be *creative*. Common tasks such as account creation can be easily thought of. We encouraged the annotators to add constraints (*e.g.*, "*create a Reddit account* ***identical to my GitLab one***") to make the intents more unique.

(3) The intent should be formulated as a *template* by making replaceable elements as variables. The annotators were also responsible for developing several instantiations for each variable. For example, the intent "*create a Reddit account identical to my GitLab one*" can be converted into "*create a {{site1}} account identical to my {{site2}} one*", with an instantiation like "*{site1: Reddit, site2: GitLab}*" and another like "*{site1: GitLab, site2: OneStopShopping}*". Notably, tasks derived from the same template can have distinct execution traces. The similarity resides primarily in the high-level semantics rather than the specific implementation.

We also provided a prompt for the annotators to use with ChatGPT[3] for inspiration, that contains an overview of each website and instructs the model to describe potential tasks to be performed on these sites. Furthermore, we offered a curated list of examples for annotators to reference.

Intent Analysis In total, we curated 241 templates and 812 instantiated intents. On average, each template is instantiated to 3.3 examples. The intent distribution is shown in Figure 9.6.

[3] https://chat.openai.com/

Action Type	Description
`noop`	Do nothing
`click(elem)`	Click at an element
`hover(elem)`	Hover on an element
`type(elem, text)`	Type to an element
`press(key_comb)`	Press a key comb
`tab_focus(index)`	focus on i-th tab
`new_tab`	Open a new tab
`tab_close`	Close current tab
`go_back`	Visit the last URL
`go_forward`	Undo `go_back`
`goto(URL)`	Go to URL

Figure 9.4: Action Space of `WebArena`

Category	Example
Information Seeking	When was the last time I bought shampoo
	Compare walking and driving time from AMC Waterfront to Randyland
Site Navigation	Checkout merge requests assigned to me
	Show me the ergonomic chair with the best rating
Content & Config	Post to ask "whether I need a car in NYC"
	Delete the reviews from the scammer Yoke

Figure 9.5: Example intents from three categories.

Furthermore, we classify the intents into three primary categories with examples shown in Figure 9.5:

(1) **Information-seeking** tasks expect a textual response. Importantly, these tasks in `WebArena` often require navigation across multiple pages or focus on *user-centric* content. This makes them distinct from open-domain question-answering [183, 368], which focuses on querying general knowledge with a simple retrieval step. For instance, to answer "*When was the last time I bought the shampoo*", an agent traverses the user's purchase history, checking order details to identify the most recent shampoo purchase.

(2) **Site navigation**: This category is composed of tasks that require navigating through web pages using a variety of interactive elements such as search functions and links. The objective is often to locate specific information or navigate to a particular section of a site.

(3) **Content and configuration operation**: This category encapsulates tasks that require operating in the web environment to create, revise, or configure content or settings. This includes adjusting settings, managing accounts, performing online transactions, generating new web content, and modifying existing content. Examples range from updating a social media status or README file to conducting online purchases and configuring privacy settings.

9.3.2 Evaluation Annotation

Evaluating Information Seeking Tasks To measure the correctness of information-seeking tasks where a textual answer is expected, we provide the annotated answer a^* for each intent. The a^* is further compared with the predicted answer $\hat{a}$ with one of the following scoring functions $r_{\text{info}}(\hat{a}, a^*)$.

First, we define `exact_match` where only $\hat{a}$ that is identical with a^* receives a score of one. This function is primarily applicable to intent types whose responses follow a more standardized format, similar to the evaluation on question answering literature [293, 368].

Second, we create `must_include` where any $\hat{a}$ containing a^* receives a score of one. This function is primarily used in when an unordered list of text is expected or where the emphasis of evaluation is on certain key concepts. In the second example in Table 9.1, we expect both the correct name and the email address to be presented, irrespective of the precise wording used to convey the answer.

Finally, we introduce `fuzzy_match` where we utilize a language model to assess whether $\hat{a}$ is semantically equivalent to a^*. Specifically, in this work, we use `gpt-4-0613` to perform this evaluation. The corresponding prompt details are provided in Appendix 9.8.6. The `fuzzy_match` function applies to situations where the format of the answer is diverse. For instance, in responding to "*Compare the time for walking and driving route from AMC Waterfront to Randyland*", it is essential to ensure that driving time and walking time are accurately linked with the correct terms. The `fuzzy_match` function could also flexibly match the time "2h58min" with different forms such as "2 hour 58 minutes", "2:58" and others.

Evaluating Site Navigation and Content & Config Tasks The tasks in these categories require accessing web pages that meet certain conditions or performing operations that modify the underlying data storage of the respective websites. To assess these, we establish reward functions $r_{\text{prog}}(\mathbf{s})$ that programmatically examine the intermediate states $\mathbf{s}$ within an execution trajectory to ascertain whether the outcome aligns with the intended result. These intermediate states are often the underlying databases of the websites, the status, and the content of a web page at each step of the execution.

Evaluating each instance involves two components. First, we provide a `locator`, tasked with retrieving the critical content pertinent to each intent. The implementation of this locator varies from a database query, a website-supported API call, to a JavaScript element selection on the relevant web page, depending on implementation feasibility. For example,

Function	ID	Intent	Eval Implementation
$r_{\text{info}}(a^*, \hat{a})$	1	Tell me the name of the customer who has the most cancellations in the history	`exact_match`($\hat{a}$, "Samantha Jones")
	2	Find the customer name and email with phone number 8015551212	`must_include`($\hat{a}$, "Sean Miller") `must_include`($\hat{a}$, "sean@gmail.com")
	3	Compare walking and driving time from AMC Waterfront to Randyland	`fuzzy_match`($\hat{a}$, "walking: 2h58min") `fuzzy_match`($\hat{a}$, "driving: 21min")
$r_{\text{prog}}(\mathbf{s})$	4	Checkout merge requests assigned to me	`url=locate_current_url(s)` `exact_match(URL, "gitlab.com/merge_requests?assignee_username=byteblaze")`

	5	Post to ask "whether I need a car in NYC"	`url=locate_latest_post_url(s)`
			`body=locate_latest_post_body(s)`
			`must_include(URL, "/f/nyc")`
			`must_include(body,"a car in NYC")`

Table 9.1: We introduce two evaluation approaches. r_{info} (top) measures the correctness of performing information-seeking tasks. It compares the predicted answer $\hat{a}$ with the annotated reference a^* with three implementations. r_{prog} (bottom) programmatically checks whether the intermediate states during the executions possess the anticipated properties specified by the intent.

the evaluation process for the intent of the fifth example in Table 9.1, first obtains the URL of the latest post by examining the last state in the state sequence $\mathbf{s}$. Then it navigates to the corresponding post page and obtains the post's content by running the Javascript "`document.querySelector('.submission__inner').outerText`".

Subsequently, we annotate `keywords` that need to exist within the located content. For example, the evaluation verifies if the post is correctly posted in the "nyc" subreddit by examining the URL of the post and if the post contains the requested content by examining the post content. We reuse the `exact_match` and `must_include` functions from information-seeking tasks for this purpose.

Unachievable Tasks Due to constraints such as inadequate evidence, user permissions (§9.8.3), or the absence of necessary functional support on the website, humans may ask for tasks that are not possible to complete. Inspired by previous work on evaluating question-answering models

on unanswerable questions [294], we design unachievable tasks in WebArena. For instance, fulfilling an intent like "*Tell me the contact number of OneStopShop*" is impracticable in WebArena, given that the website does not provide such contact information. We label such instances as "N/A" and expect an agent to produce an equivalent response. These examples allow us to assess an agent's ability to avoid making unfounded claims and its adherence to factual accuracy.

Annotation Process The intents were contributed by the authors following the annotation guideline in §9.3.1. Every author has extensive experience with web-based tasks. The reference answers to the information-seeking tasks were curated by the authors and an external annotator. To ensure consistency and accuracy, each question was annotated twice. If the two annotators disagreed, a third annotator finalized the annotation. The programs to evaluate the remaining examples were contributed by three of the authors who are proficient in JavaScript programming. Difficult tasks were often discussed collectively to ensure the correctness of the annotation. The annotation required the annotator to undertake the full execution and scrutinize the intermediate states.

Human Performance We sample one task from each of the 170 templates and ask five computer science graduate students to perform these tasks. The human performance is on the right. Overall, the human annotators complete 78.24% of the tasks, with lower performance on information-seeking tasks. Through examining the recorded trajectories, we found that 50% of the failures are due to misinterpreting the intent (*e.g.*, providing travel distance when asked for travel time), incomplete answers (*e.g.*, providing only name when asked for name and email), and incomplete executions (*e.g.*, partially filling the product information), while the remaining instances have more severe failures, where the executions are off-target.

Avg. Time	110s
Success Rate$_{\text{info}}$	74.68%
Success Rate$_{\text{others}}$	81.32%
Success Rate$_{\text{all}}$	78.24%

9.4 Baseline Web Agents

We experiment with three LLMs using two prompting strategies, both with two examples in the context. In the first setting, we ask the LLM to directly predict the next action given the current observation, the intent and the previously performed action. In the second setting, with the same information, the model first performs chain-of-thought reasoning steps in the text before the action prediction (CoT, Wei et al. [358], Yao et al. [373]). Before the examples, we provide a

detailed overview of the browser environment, the allowed actions, and many rules. To make the model aware of the unachievable tasks, the instruction explicitly asks the agent to stop if it believes the task is impossible to perform. We refer to this directive as Unachievable hint, or **UA hint**. This introduction is largely identical to the guidelines we presented to human annotators to ensure a fair comparison. We use an accessibility tree with element IDs as the observation space. The agent can identify which element to interact with by the ID of the element. For instance, the agent can issue `click [1582]` to click the "Add to Cart" button with the ID of 1582. The full prompts can be found in Appendix 9.8.7. The detailed configurations of each model can be found in Appendix 9.8.5.

9.5 Results

9.5.1 Main Results

The main results are shown on the top of Table 9.2. GPT-4 [261] with CoT prompting achieves a modest end-to-end task success rate of 11.70%, which is significantly lower than the human performance of 78.24%. GPT-3.5 [259] with CoT prompting is only able to successfully perform 8.75% of the tasks. The explicit reasoning procedure is somewhat helpful, it brings 2.34% improvement over the version without it. Further, TEXT-BISON-001 [17] underperforms GPT-3.5, with a success rate of 5.05%. These results underline the inherent challenges and complexities of executing tasks that span long horizons, particularly in realistic environments such as WebArena.

CoT	UA Hint	Model	SR	SR_{AC}	SR_{UA}
✓	✓	TEXT-BISON-001	5.05	4.00	27.78
✗	✓	GPT-3.5	6.41	4.90	38.89
✓	✓	GPT-3.5	8.75	6.44	58.33
✓	✓	GPT-4	11.70	8.63	**77.78**
✗	✗	GPT-3.5	5.10	4.90	8.33
✓	✗	GPT-3.5	6.16	6.06	8.33
✓	✗	GPT-4	**14.41**	**13.02**	44.44
-	✓	Human	78.24	77.30	100.00

Table 9.2: The end-to-end task success rate (SR %) on WebArena with different prompting strategies. **CoT**: the model performs step-by-step reasoning before issuing the action. **UA hint**: ask the model to stop when encountering unachievable questions.

9.5.2 Analysis

Do models know when to stop? In our error analysis of the execution trajectories, we observe a prevalent error pattern of early stopping due to the model's conclusion of unachievability. For instance, GPT-4 erroneously identifies 54.9% of feasible tasks as impossible. This issue primarily stems from the UA hint in the instruction, while this hint allows models to identify unachievable tasks, it also hinders performance on achievable tasks. To address this, we conduct an ablation study where we remove this hint. We then break down the success rate for both achievable and unachievable tasks. As shown in Table 9.2, eliminating this instruction led to a performance boost in achievable tasks, enhancing the overall task success rate of GPT-4 to 14.41%. Despite an overall decline in identifying unachievable tasks, GPT-4 retains the capacity to recognize 44.44% of such tasks. It does so by generating *reasons of non-achievability*, even without explicit instructions. On the other hand, GPT-3.5 rarely exhibits this level of reasoning. Instead, it tends to follow problematic patterns such as hallucinating incorrect answers, repeating invalid actions, or exceeding the step limits. This result suggests that even subtle differences in instruction design can significantly influence the behavior of a model in performing interactive tasks in complex environments.

Can a model maintain consistent performance across similar tasks? Tasks that originate from the same template usually follow similar reasoning and planning processes, even though their observations and executions will differ. We plot a histogram of per-template success rates for our models in Table 9.3. Of the 61 templates, GPT-4 manages to achieve a 100% task success rate on only four templates, while GPT-3.5 fails to achieve full task completion for any of the templates. In many cases, the models are only able to complete one task variation with a template. These observations indicate that even when tasks are derived from the same template, they can present distinct challenges. For instance, while "*Fork metaseq*" can be a straightforward task, "*Fork all repos from Facebook*" derived from the same template requires more repetitive operations, hence increasing its complexity. Therefore, `WebArena` provide a testbed to evaluate more sophisticated methods. In particular, those that incorporate memory components, enabling the *reuse* of successful strategies from past experiments [351, 409]. More error analysis

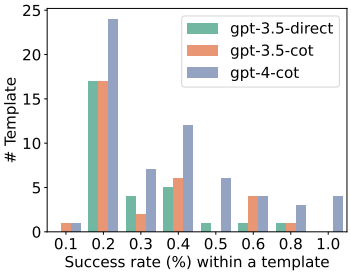


Table 9.3: Distribution of success rates on templates with ≥ 1 successful executions on GPT models (no UA hint).

with examples can be found in Appendix 9.8.8.

9.6 Related Work

Benchmark	Dynamic Interaction?	Realistic Environment?	Diverse Human Tasks?	Functional Correctness?
Mind2Web [71]	✗	✓	✓	✗
Form/QAWoB [319]	✗	✓	✓	✗
MiniWoB++ [214]	✓	✗	✗	✓
Webshop [371]	✓	✗	✗	✓
ALFRED [323]	✓	✗	✗	✓
VirtualHome [279]	✗	✗	✓	✗
AndroidEnv [340]	✓	✓	✗	✗
WebArena	✓	✓	✓	✓

Table 9.4: The comparison between our benchmark and existing benchmarks on grounding natural language instructions to concrete executions. Our benchmark is implemented in our fully interactable highly-realistic environment. It features diverse tasks humans may encounter in their daily routines. We design evaluation metrics to assess the functional correctness of task executions.

Benchmarks for Controlling Agents through Natural Language Controlling agents via natural language in the digital world have been studied in the literature [39, 71, 204, 214, 319, 340, 366]. However, the balance between *functionality*, *authenticity*, and *support for environmental dynamics* remains a challenge. Existing benchmarks often compromise these aspects, as shown in Table 9.4. Some works rely on static states, limiting agents' explorations and functional correctness evaluation [71, 319], while others simplify real-world complexities, restricting task variety [214, 371]. While AndroidEnv [340] replicates an Android setup, it does not guarantee the reproducibility since live Android applications are used. [177, 279, 323] and extends to gaming environments [81, 182], where the environment mechanisms often diverge from human objectives.

Interactive Decision-Making Agents [249] introduce WebGPT which searches the web and reads the search results to answer questions. [109] propose a web agent that decomposes tasks into more manageable sub-tasks and synthesizes Javascript code for the task executions. Adding a multi-modal dimension, [190] and [317] develop agents that predict actions based on screenshots of web pages rather than relying on the text-based DOM trees. Performing tasks in interactive environments requires the agents to exhibit several capabilities including hierarchical planning, state tracking, and error recovery. Existing works [136, 203, 222] observe LLMs could break a task into more manageable sub-tasks [410]. This process can be further refined by representing task executions as programs, a technique that aids sub-task management and skill reuse [93, 206, 351, 409]. Meanwhile, search and backtracking methods introduce a more structured approach to planning while also allowing for decision reconsideration [218, 374]. Existing works also incorporate failure recovery, self-correction [171, 322], observation summarization [331] to improve execution robustness. The complexity of `WebArena` presents a unique challenge and opportunity for further testing and improvement of these methods.

9.7 Conclusion

We present `WebArena`, a highly-realistic, standalone, and reproducible web environment designed for the development and testing of autonomous agents. `WebArena` includes fully functional web applications and genuine data from four major categories, providing a realistic platform for agent interaction. It further supports a wide range of tools and external knowledge bases, fostering a focus on human-like problem-solving. Additionally, we curate a comprehensive benchmark consisting of 812 examples that focus on translating high-level natural language intents into specific web interactions. We also offer metrics to programmatically ascertain whether tasks have been completed according to the desired objectives. Our experiments show that even GPT-4 only achieves a limited end-to-end task success rate of 14.41%, significantly lagging behind the human performance of 78.24%. These findings underscore the need for future research to focus on enhancing the robustness and efficacy of autonomous agents within `WebArena` environment.

9.8 Appendix

9.8.1 Website Implementation

Given the selected websites described in §9.2.2, we make the best attempt to reproduce the functionality of commonly used sites in a reproducible way. To achieve this, we utilized open-source frameworks for the development of the websites across various categories and imported data from their real-world counterparts. For the E-commerce category, we constructed a shopping website with approximately $90k$ products, including the prices, options, detailed product descriptions, images, and reviews, spanning over 300 product categories. This website is developed using Adobe Magento, an open-source e-commerce platform[4]. Data resources were obtained from data from actual online sites, such as that included in the Webshop data dump[371]. As for the social forum platform, we deployed an open-source software Postmill[5], the open-sourced counterpart of Reddit[6]. We sampled from the top 50 subreddits[7]. We then manually selected many subreddit for northeast US cities as well as subreddit for machine learning and deep learning-related topics. This manual selection encourages cross-website tasks such as seeking information related to the northeast US on both Reddit and the map. In total, we have 95 subreddits, 127390 posts, and 661781 users. For the collaborative software development platform, we choose GitLab[8]. We heuristically simulate the code repository characteristics by sampling at least ten repositories for every programming language: 80% of them are sampled from the set of top 90 percentile wrt stars repos using a discrete probability distribution weighted proportional to their number of stars; the remaining are sampled from the bottom ten percentile set using similar weighted distribution. This is done to ensure fair representation of repos of all kinds, from popular projects with many issues and pull requests to small personal projects. In total, we have 300 repositories and more than 1000 accounts with at least one commit to a repository. For the content management system, we adapted Adobe Magento's admin portal, deploying the sample data provided in the official guide. We employ OpenStreetMap[9] for map service implementation, confining our focus to the northeast US region due to data storage constraints. We implement a calculator and a scratchpad ourselves.

[4] https://github.com/magento/magento2
[5] https://postmill.xyz/
[6] https://www.reddit.com/
[7] https://redditlist.com/sfw.html
[8] https://gitlab.com/gitlab-org/gitlab
[9] https://www.openstreetmap.org/

Lastly, we configure the knowledge resources as individual websites, complemented with search functionality for efficient information retrieval. Specifically, we utilize Kiwix[10] to host an offline version of English Wikipedia with a knowledge cutoff of May 2023. The user manuals for GitLab and Adobe Commerce Merchant documentation are scraped from the official websites.

9.8.2 Environment Delivery and Reset

One goal for our evaluation environment is ease of use and reproducibility. As a result, we deploy our websites in separate Docker images [11], one per website. The Docker images are fully self-contained with all the code of the website, database, as well as any other software dependencies. They also do not rely on external volume mounts to function, as the data of the websites are also part of the docker image. This way, the image is easy to distribution containing all the pre-populated websites for reproducible evaluation. End users can download our packaged Docker images and run them on their systems and re-deploy the exact websites together with the data used in our benchmarks for their local benchmarking.

Since some evaluation cases may require the agent to modify the data contained in the website, *e.g.*, creating a new user, deleting a post, etc., it is crucial to be able to easily reset the website environment to its initial state. With Docker images, the users could stop and delete the currently running containers for that website and start the container from our original image again to fully reset the environment to the initial state. Depending on the website, this process may take from a few seconds to one minute. However, not all evaluation cases would require an environment reset, as many of the intents are information gathering and are read-only for the website data. Also, combined with the inference time cost for the agent LLMs, we argue that this environment reset method, through restarting Docker containers from the original images, will have a non-negligible but small impact on evaluation time.

9.8.3 User Roles Simulation

Users of the same website often have disparate experiences due to their distinct *roles*, *permissions*, and *interaction histories*. For instance, within an E-commerce CMS, a shop owner might possess full read and write permissions across all content, whereas an employee might only be granted write permissions for products but not for customer data. We aim to emulate this scenario by generating unique user profiles on each platform.

[10] https://www.kiwix.org/en/
[11] https://www.docker.com/

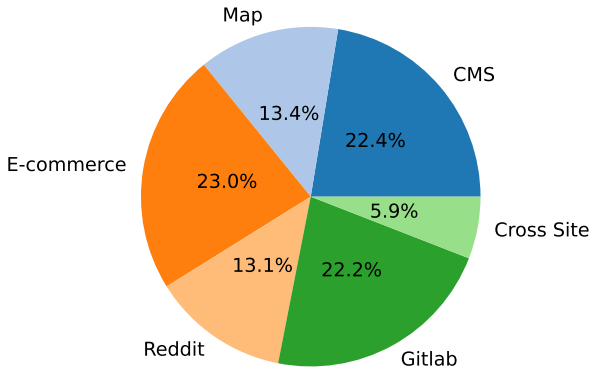


Figure 9.6: The intent distribution across different websites. Cross-site intents necessitate interacting with multiple websites. Notably, regardless of the website, all user intents require interactions with multiple web pages.

On the shopping site, we created a customer profile that has over 35 orders within a span of two years. On GitLab, we selected a user who maintains several popular open-source projects with numerous merge requests and issues. This user also manages a handful of personal projects privately. On Reddit, our chosen profile was a user who actively participates in discussions, with many posts and comments. Lastly, on our E-commerce CMS, we set up a user profile for a shop owner who has full read-and-write access to all system contents.

All users are automatically logged into their accounts using a pre-cached cookie. To our best knowledge, this is the first publicly available agent evaluation environment to implement such a characteristic. Existing literature typically operates under the assumption of universally identical user roles [71, 214, 319].

9.8.4 Intent Distribution

The distribution of intents across the websites are shown in Figure 9.6.

9.8.5 Experiment Configurations

We experiment with GPT-3.5-TURBO-16K-0613, GPT-4-0613, and TEXT-BISON-001 with a temperature of 1.0 and a top-p parameter of 0.9. The maximum number of state transitions is set to 30. We halt execution if the same action is repeated more than three times on the same observation or if the agent generates three consecutive invalid actions. These situations typically indicate a

CoT	UA Hint	Model	SR
✓	✗	GPT-3.5	6.28

Table 9.5: The task success rate (SR %) of GPT-3.5-TURBO-16K-0613 with temperature 0.0.

high likelihood of execution failure and hence warrant early termination. For TEXT-BISON-001, we additionally allow ten retries until it generates a valid action.

Primarily, we use a high temperature of 1.0 to encourage the *exploration*. To aid replicating the results, we provide the results of GPT-3.5-TURBO-16K-0613 with temperature 0.0 in Table 9.5 and the execution trajectories in our code repository.

9.8.6 Prompt for `fuzzy_match`

Help a teacher to grade the answer of a student given a question. Keep in mind that the student may use different phrasing or wording to answer the question. The goal is to evaluate whether the answer is semantically equivalent to the reference answer.
question: {{intent}}
reference answer: {{reference answer}}
all the string 'N/A' that you see is a special sequence that means 'not achievable'
student answer: {{prediction}}
Conclude the judgement by correct/incorrect/partially correct.

Predictions that are judged as "correct" will receive a score of one, while all other predictions will receive a score of zero.

9.8.7 The Prompts of the Baseline Web Agents

The system message of the reasoning agent for both GPT-3.5 and GPT-4 is in Figure 9.7, and two examples are in Figure 9.8. The system message of the direct agent for GPT-3.5 is in Figure 9.9 and the two examples are in Figure 9.10. **UA hint** refers to the instruction of " If you believe the task is impossible to complete, provide the answer as "N/A" in the bracket.". We remove this sentence in our ablation studies.

You are an autonomous intelligent agent tasked with navigating a web browser. You will be given web-based tasks. These tasks will be accomplished through the use of specific actions you can issue.

Here's the information you'll have:
The user's objective: This is the task you're trying to complete.
The current web page's accessibility tree: This is a simplified representation of the webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track your progress.

The actions you can perform fall into several categories:
Page Operation Actions
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the field with id. By default, the "Enter" key is pressed after typing unless press_enter_after is set to 0.
`hover [id]`: Hover over an element with id.
`press [key_comb]`: Simulates the pressing of a key combination on the keyboard (e.g., Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser's focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous
`go_back` action was performed).

Completion Action:
`stop [answer]`: Issue this action when you believe the task is complete. If the objective is to find a text-based answer, provide the answer in the bracket. If you believe the task is impossible to complete, provide the answer as "N/A" in the bracket.

Homepage:
If you want to visit other websites, check out the homepage at http://homepage.com. It has a list of websites you can visit.
http://homepage.com/password.html lists all the account names and passwords for the websites. You can use them to log in to the websites.

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action.
4. Generate the action in the correct format. Start with a "In summary, the next action I will perform is“ phrase,

example_user
OBSERVATION:
[1744] link 'HP CB782A#ABA 640 Inkjet Fax Machine (Renewed)'
  [1749] StaticText '$279.49'
  [1757] button 'Add to Cart'
  [1760] button 'Add to Wish List'
  [1761] button 'Add to Compare'
URL: http://onestopmarket.com/office-products/office-electronics.html
OBJECTIVE: What is the price of HP Inkjet Fax Machine
PREVIOUS ACTION: None

example_assistant
Let's think step-by-step. This page lists the information of HP Inkjet Fax Machine, which is the product identified in the objective. Its price is $279.49. I think I have achieved the objective. I will issue the stop action with the answer. In summary, the next action I will perform is ```stop [$279.49]```

example_user
OBSERVATION:
[164] textbox 'Search' focused: True required: False
[171] button 'Go'
[174] link 'Find directions between two points'
[212] heading 'Search Results'
[216] button 'Close'
URL: http://openstreetmap.org
OBJECTIVE: Show me the restaurants near ABC
PREVIOUS ACTION: None

example_assistant
Let's think step-by-step. This page has a search box whose ID is [164]. According to the nominatim rule of openstreetmap, I can search for the restaurants near a location by "restaurants near". I can submit my typing by pressing the Enter afterwards. In summary, the next action I will perform is ```type [164] [restaurants near ABC] [1]```

Figure 9.8: The two examples provided as `example_user` and `example_assistant` for the reasoning agent. Before issuing the action, the agent first perform reasoning.

You are an autonomous intelligent agent tasked with navigating a web browser. You will be given web-based tasks. These tasks will be accomplished through the use of specific actions you can issue.

Here's the information you'll have:
The user's objective: This is the task you're trying to complete.
The current web page's accessibility tree: This is a simplified representation of the webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track your progress.

The actions you can perform fall into several categories:
Page Operation Actions
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the field with id. By default, the "Enter" key is pressed after typing unless press_enter_after is set to 0.
`hover [id]`: Hover over an element with id.
`press [key_comb]`: Simulates the pressing of a key combination on the keyboard (e.g., Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser's focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous
`go_back` action was performed).

Completion Action:
`stop [answer]`: Issue this action when you believe the task is complete. If the objective is to find a text-based answer, provide the answer in the bracket. If you believe the task is impossible to complete, provide the answer as "N/A" in the bracket.

Homepage:
If you want to visit other websites, check out the homepage at http://homepage.com. It has a list of websites you can visit.
http://homepage.com/password.html lists all the account name and password for the websites. You can use them to log in to the websites.

To be successful, it is very important to follow the following rules:

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. Generate the action in the correct format. Always put the action inside a pair of ```. For example, ```click

example_user
OBSERVATION:
[1744] link 'HP CB782A#ABA 640 Inkjet Fax Machine (Renewed)'
  [1749] StaticText '$279.49'
  [1757] button 'Add to Cart'
  [1760] button 'Add to Wish List'
  [1761] button 'Add to Compare'
URL: http://onestopmarket.com/office-products/office-electronics.html
OBJECTIVE: What is the price of HP Inkjet Fax Machine
PREVIOUS ACTION: None

example_assistant
```stop [$279.49]```

example_user
OBSERVATION:
[164] textbox 'Search' focused: True required: False
[171] button 'Go'
[174] link 'Find directions between two points'
[212] heading 'Search Results'
[216] button 'Close'
URL: http://openstreetmap.org
OBJECTIVE: Show me the restaurants near ABC
PREVIOUS ACTION: None

example_assistant
```type [164] [restaurants near ABC] [1]```

Figure 9.10: The two examples provided as `example_user` and `example_assistant` for the direct agent. The agent directly emits the next action given the observation.

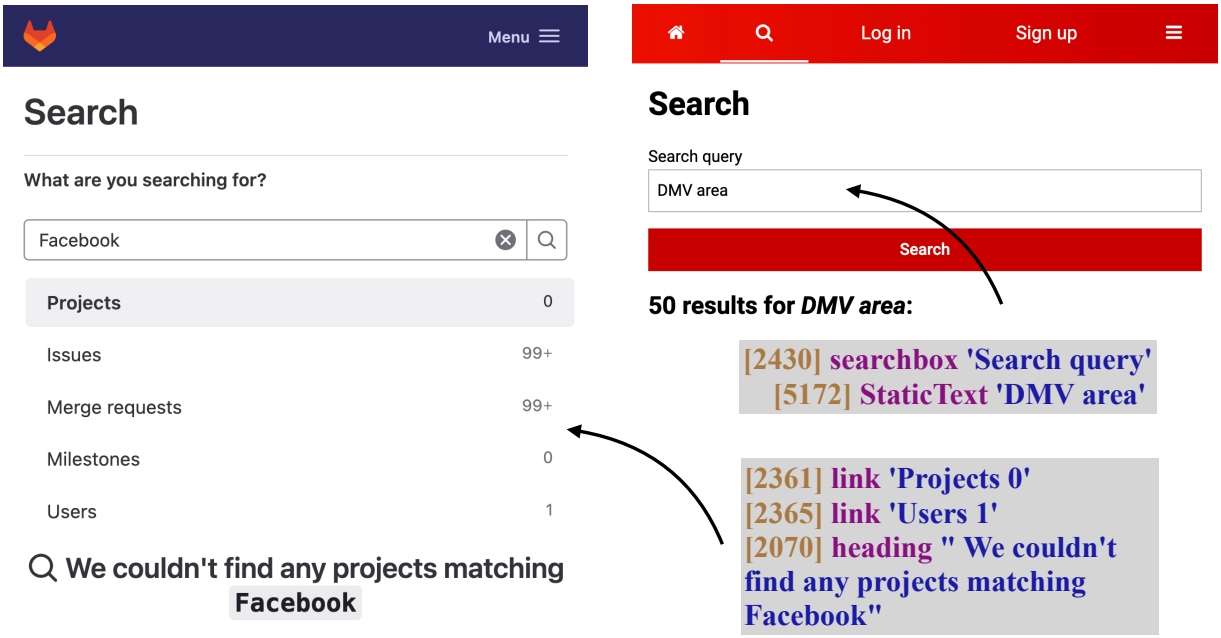


Figure 9.11: Two examples where the GPT-4 agent failed, along with their screenshot and the accessibility tree of the relevant sections (grey). On the left, the agent fails to proceed to the "Users" section to accomplish the task of "Fork all Facebook repos"; on the right, the agent repeats entering the same search query even though the observation indicates the input box is filled.

9.8.8 Additional Error Analysis

Observation Bias Realistic websites frequently present information on similar topics across various sections to ensure optimal user accessibility. However, a GPT-4 agent often demonstrates a tendency to latch onto the first related piece of information it encounters without sufficiently verifying its relevance or accuracy. For instance, the homepage of the E-Commerce CMS displays the best-selling items based on *recent purchases*, while historical best-seller data is typically accessed via a separate report. Presented with the task of "*What is the top-1 best-selling product in 2022*", the GPT-4 agent defaults to leveraging the readily available information on the homepage, bypassing the necessary step of generating the report to obtain the accurate data.

Failures in Observation Interpretation Interestingly, while GPT-4 is capable of summarizing the observations, it occasionally overlooks more granular information, such as the previously entered input. As in the right-hand example of Figure 9.11, `[5172] StaticText` indicates that the search term "DMV area" has already been entered. However, the agent disregards this detail and continuously issues the command `type [2430] [DMV area]` until it reaches the maximum step limit. Furthermore, the agent often neglects the previous action information that is provided alongside the observation.

We hypothesize that these observed failures are related to the current pretraining and supervised fine-tuning on dialogues employed in GPT models [263]. These models are primarily trained to execute instructions given *immediate* observations (*i.e.*,, the dialogue history); thereby, they may exhibit a lack of explorations. Furthermore, in dialogue scenarios, subtle differences in NL expressions often have less impact on the overall conversation. As a result, models may tend to overlook minor variations in their observations.

Chapter 10

Comparative Study of Web Navigation-based vs. API-based Agents (Proposed)

From our previous work on WebArena (Chapter 2), we find that current LLMs struggle at navigating web pages through the observations used commonly by humans. However, we argue that since the Web UI is mostly designed with a focus on real-human user experience, there might be some designs that may seem natural to human users yet pose a big challenge to AI agents. For example, a cleaner design using a lot of icons instead of descriptive text may be intuitive for humans but not for AI agents; a web page containing a lot of animations and transitions for aesthetics; or even a webpage riddled with advertisements for human consumption.

We hypothesize that current LLMs may perform better in the same tasks proposed in WebArena, but instead of using browsers as an interface, they directly call various APIs provided by the websites. For example, for a shopping website there may be APIs for searching products, ordering a specific product, querying the users' orders, etc. If we have a comprehensive enough list of APIs for websites used in WebArena, plus the corresponding detailed documentation, use cases, examples, etc. The LLM may learn how to perform the same tasks by only interacting with programmatic APIs. We argue that this will have the benefit of cleaner input/output for LLMs, and maybe more machine-readable formats, less noise, and potentially more efficient use of the limited context length by LLMs.

However, this comparison is not to showcase that we should use API-based interaction methods for LLM agents. The test benchmark using a browser environment poses a challenge

to the LLM in terms of testing human-level perception of real world websites and navigation, and is an interesting challenge by itself. However, this may not be the most efficient approach of helping users complete intents as AI agents. Using APIs also has the drawback of requiring public API access and good descriptive documents for agents to refer to. A good analogy is the camera vision-based versus radar-based approaches to self-driving cars: in some scenarios it might be sufficient to utilize camera-based vision models to recognize the surrounding road conditions, while in extreme weathers where the visibility is severely blocked, radar-based systems will provide better reliability and performance. In the future, maybe the best of both worlds can be combined.

Chapter 11

Methodology Improvements to Agents and Code Generation Models (Proposed)

Now that we have a evaluation suite for testing web agents as well as the understanding of how API calling fares against traditional web browsing agents from the previous chapters, we propose several directions to improving code generation and agent models, to move forward towards the end goal of instructing computers to perform a wide variety of tasks via natural language interface.

Improved API calling ability for LLMs. Recently, some work [329] focused on building LLM agents using APIs as an interface to the environment. Following the lessons and insights learned from the comparison between web-based and API-based WebArena intents (Chapter 10), we will propose a new API calling LLM agent. Existing work include Toolformers [312], ReAct [372], *etc.* that involve interrupting LLM generation when encountering an API call request, and putting the API response back into the generation flow to utilize the API response. Similar methods could also be tested on programming intents that complete a whole function or a whole class method given textual intent, as these tasks would usually also involve using multiple programming library APIs.

We aim to compare both retrieval-based versus pretrain/finetune based methods for learning how to call specific APIs given documentation, metadata. We could retrieve relevant APIs from the database and prompt the LLM or finetune the LLM based on synthetically created API calling examples as training sets, sampled from a large pool of APIs based on some kind of heuristics. We could also explore ways to learn from demo/instructions available online in a distant supervision fashion.

Learning from demonstration and instructions. Inspired by MineDojo[1] [81], we want to explore ways of learning from demonstrations and manuals and tutorials on the Internet for LLM agents. It could be either for WebArena tasks as well as for programming code generation tasks. The idea is that manuals and tutorials on the Internet might be a good distant supervision data source, and we are curious how the knowledge and steps presented in those manuals about popular public websites would transfer into our own versions of those websites inside WebArena.

we propose to improve existing LLMs through large amounts of instructions, manuals, tutorials and demonstrations found online about how to perform certain tasks.

Iterative code generation. Besides focusing on only agents, the "iterative", or "interactive" nature of how agents interact with outside world via actuation and feedback could be potentially applied to another key natural language interface: more complex, longer code generation. Code generation line by line, while utilizing the execution results and variable states from the previously generated lines to inform the generation of the next line. The intuition comes from WebArena tasks where the action generation process is actively informed by the previous execution results and are inherently iterative. On the other hand, when developers are writing code, they also usually print out values of the intermediate variables or inspect the state of some objects before writing the next lines of code. These may also provide the valuable contexts that were missing in our human study of code generation models (Chapter 4).

We propose to generate longer code completions or API calling sequences by iteratively generating part of it and then consider the intermediate execution responses to inform a more accurate generation later on.

[1] https://minedojo.org/

Bibliography

[1] R. Agashe, Srini Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. In *2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing (EMNLP/IJCNLP)*, 2019. 4.2, 4.9.1, 4.9.3

[2] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5435–5445, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1546. URL https://www.aclweb.org/anthology/D19-1546. 1.1, 2.1

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2021.naacl-main.211. 3.2.1

[4] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023. 1.1

[5] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019. 3.4.2

[6] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013. 5.4

[7] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 281–293, 2014. 4.8

[8] Miltiadis Allamanis, Daniel Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *The 32nd International Conference on Machine Learning (ICML)*, 2015. 4.9.2

[9] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2123–2132. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/allamanis15.html. 1.3, 7.1, 7.7

[10] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4): 1–37, 2018. 4.1, 35

[11] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 245–256. PMLR, 2020. URL http://proceedings.mlr.press/v119/alon20a.html. 1.1, 1.3, 3.1, 5.1, 7.1

[12] Uri Alon, Frank F Xu, Junxian He, Sudipta Sengupta, Dan Roth, and Graham Neubig. Neuro-symbolic language modeling with automaton-augmented retrieval. *arXiv preprint arXiv:2201.12431*, 2022. 1.3, 6.1, 6.1

[13] S. Amann, Sebastian Proksch, and S. Nadi. Feedbag: An interaction tracker for visual studio. *International Conference on Program Comprehension (ICPC)*, pages 1–3, 2016. 4.9.4

[14] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. IEEE, 2016. 4.9.6

[15] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian D. Reid, Stephen Gould, and Anton van den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA,*

June 18-22, 2018, pages 3674–3683. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00387. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Anderson_Vision-and-Language_Navigation_Interpreting_CVPR_2018_paper.html. 1.4, 9.1

[16] Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81, 1995. 1

[17] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. Palm 2 technical report, 2023. 9.5.1

[18] Yigal Arens, Craig A Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 6(2-3):99–130, 1996. 4.8

[19] Philip Arthur, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Semantic parsing of ambiguous input through paraphrasing and verification. *Transactions*

of the Association for Computational Linguistics (TACL), 3:571–584, 2015. 4.8

[20] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732. 1.1, 1.3, 3.1, 3.2.1, 7.1, 7.4.2, 7.9.2

[21] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 6.2

[22] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. Harnessing stack overflow for the ide. In *International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 26–30. IEEE, 2012. 4.3

[23] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018. 1.1, 3.1, 5.1, 5.2, 5.4, 5.6.1, 5.2, 6.1, 6.3

[24] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.0473. 5.1

[25] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *5th International Conference on Learning Representations (ICLR)*, 2017. 4.9.1, 4.9.5

[26] S. Barman, Sarah E. Chasins, Rastislav Bodík, and Sumit Gulwani. Ringer: web automation by demonstration. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016. 4.9.5

[27] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and Jørgen Fischer Nilsson. Synthesis of programs in computational logic. In *Program Development in Computational Logic*, 2004. 4.9.5

[28] Nathanaël Beau and Benoit Crabbé. The impact of lexical and grammatical processing on generating code from natural language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2204–2214, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.173. URL https://aclanthology.org/2022.findings-acl.173. 3

[29] Andrew Bell, Malcolm Fairbrother, and Kelvyn Jones. Fixed and random effects models: making an informed choice. *Quality & Quantity*, 53(2):1051–1074, 2019. 4.5

[30] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2018, Paris, France, June 19-22, 2018*, pages 3:1–3:6. ACM, 2018. doi: 10.1145/3220134.3220135. URL https://doi.org/10.1145/3220134.3220135. 4.9.5

[31] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003. URL http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf. 1.1, 3.1, 5.1, 6.1

[32] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D13-1160. 1.1, 2.1

[33] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing (EMNLP)*, pages 1533–1544, 2013. 4.9.3

[34] Yonatan Bisk, Jan Buys, Karl Pichotta, and Yejin Choi. Benchmarking hierarchical script knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4077–4085, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1412. URL https://www.aclweb.org/anthology/N19-1412. 1.4, 9.1

[35] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata. 1.1, 3.1, 3.2.1, 7.1, 7.3.2

[36] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. 2022. 1.1, 3.1, 3.2.1

[37] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 2206–2240. PMLR, 2022. URL https://proceedings.mlr.press/v162/borgeaud22a.html. 8.1, 8.4, 8.1, 8.10.1

[38] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022. 1.3, 6.1, 6.1

[39] S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90, Suntec, Singapore, 2009. Association for Computational Linguistics. URL https://aclanthology.org/P09-1010. 9.6

[40] S.R.K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a Monte-Carlo framework. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 268–277, Portland, Oregon, USA, 2011. Association for Computational Linguistics. URL https://aclanthology.org/P11-1028. 7.7

[41] J. Brandt, P. Guo, J. Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2009. 4.9.4

[42] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009. URL http://pgbovine.net/publications/

opportunistic-programming-two-studies_CHI-2009.pdf. 2.1

[43] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010. URL https://hci.stanford.edu/publications/2010/blueprint/brandt_chi10_blueprint.pdf. 2.1

[44] Denny Britz, Quoc Le, and Reid Pryzant. Effective domain mixing for neural machine translation. In *Proceedings of the Second Conference on Machine Translation*, pages 118–126, 2017. 5.3

[45] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL https://arxiv.org/abs/1606.01540. 9.1

[46] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX. 1.3, 7.1

[47] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html. 1.1, 3.1, 3.4.1, 5.1, 5.2, 6.1, 6.3

[48] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner,

Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html. 8.1, 8.3.1, 8.5

[49] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. 1.4

[50] Brock Angus Campbell and Christoph Treude. Nlp2code: Code snippet content assist via natural language tasks. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 628–632. IEEE, 2017. 4.3, 4.3, 4.8

[51] Veronica Cateté and T. Barnes. Application of the delphi method in computer science principles rubric creation. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017. 4.4.4

[52] Sarah E. Chasins, S. Barman, Rastislav Bodík, and Sumit Gulwani. Browser record and replay as a building block for end-user web automation tools. *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015. 4.9.5

[53] Sarah E. Chasins, Maria Mueller, and Rastislav Bodík. Rousillon: Scraping distributed hierarchical web data. *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2018. 4.9.5

[54] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1870–1879. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1171. URL https://doi.org/10.18653/v1/P17-1171. 1.3, 8.1

[55] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374. 1.1, 1.4, 3, 3.1, 3.2.1, 3.3, 3.4.1, 3.4.2, 7.1, 7.3.2, 7.4.2, 7.9.2,

9.1

[56] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 2020. URL http://proceedings.mlr.press/v119/chen20j.html. 7.3.1, 7.3.1

[57] X. Chen, C. Liu, and D. Song. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations (ICLR)*, 2019. 4.9.5

[58] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://www.aclweb.org/anthology/D14-1179. 2.3.1

[59] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311. 8.1

[60] Chenhui Chu, Raj Dabre, and Sadao Kurohashi. An empirical comparison of domain adaptation methods for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 385–391, 2017. 5.3

[61] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009. 6.8.2

[62] J. Cohen. *Applied multiple regression/correlation analysis for the behavioral sciences*. Lawrence Erlbaum, 2003. ISBN 0805822232. 4.5

[63] Nachshon Cohen, Oren Kalinsky, Yftah Ziser, and Alessandro Moschitti. Wikisum: Coherent summarization dataset for efficient human-evaluation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021*, pages 212–219. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-short.28. URL https://doi.org/10.18653/v1/2021.acl-short.28. 8.1

[64] Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. *Advances in Neural Information Processing Systems*, 32:7059–7069, 2019. 3.1

[65] Harald Cramér. *Mathematical methods of statistics*, volume 43. Princeton University Press, 1999. 4.6.2

[66] A. Cypher, Daniel C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. Myers, and Alan Turransky. Watch what i do: programming by demonstration. 1993. 4.9.5

[67] Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, Christine Pao David Pallett, Alexander Rudnicky, , and Elizabeth Shriber. Expanding the scope of the ATIS task: The ATIS-3 corpus. *Proceedings of the workshop on Human Language Technology*, pages 43–48, 1994. URL http://dl.acm.org/citation.cfm?id=1075823. 1.1, 2.1

[68] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of ACL*, 2019. 5.2

[69] M. Dawood, Khalid A. Buragga, Abdul Raouf Khan, and Noor Zaman. Rubric based assessment plan implementation for computer science program: A practical approach. *Proceedings of 2013 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*, pages 551–555, 2013. 4.4.4

[70] David Demeter, Gregory Kimmel, and Doug Downey. Stolen probability: A structural weakness of neural language models. In *Proceedings of the 58th Annual Meeting of the*

Association for Computational Linguistics, pages 2191–2197, 2020. 6.6, 6.6.4

[71] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023. 1.4, 9.1, 9.1, 9.2.3, **??**, 9.6, 9.8.3

[72] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016. 1.1, 3.1

[73] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423. 3.2.1, 5.1, 6.3

[74] Edsger W Dijkstra. On the foolishness of "natural language programming". In *Program Construction*, pages 51–53. Springer, 1979. 1, 4.1, 4.8, 4.9.1

[75] Emily Dinan, Stephen Roller, Kurt Shuster, Angela Fan, Michael Auli, and Jason Weston. Wizard of wikipedia: Knowledge-powered conversational agents. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=r1l73iRqKm. 8.9

[76] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1004. URL https://www.aclweb.org/anthology/P16-1004. 1.1, 2.1

[77] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1068. URL https://www.aclweb.org/anthology/P18-1068. 1.1, 2.1

[78] Andrew Drozdov, Shufan Wang, Razieh Rahimi, Andrew McCallum, Hamed Zamani, and Mohit Iyyer. You can't pick your neighbors, or can you? when and how to rely on retrieval in the kNN-LM. In *Findings of the Association for Computational Linguistics: EMNLP 2022*,

pages 2997–3007, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.218. 6.7

[79] K. Ellis, Maxwell Nye, Y. Pu, Felix Sosa, J. Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019. 4.9.5

[80] Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. ELI5: long form question answering. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3558–3567. Association for Computational Linguistics, 2019. doi: 10.18653/v1/p19-1346. URL https://doi.org/10.18653/v1/p19-1346. 8.1, 8.5, 8.9

[81] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=rc8o_j8I8PX. 1.4, 9.1, 9.6, 11

[82] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021. 4.8

[83] Y. Feng, R. Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018. 4.9.5

[84] Zhangyin Feng, Daya Guo, Duyu Tang, N. Duan, X. Feng, Ming Gong, Linjun Shou, B. Qin, Ting Liu, Daxin Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. In *2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020. 35

[85] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 3.2.1

[86] John K. Feser, S. Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *36th annual ACM SIGPLAN conference on Programming*

Language Design and Implementation (PLDI), 2015. 4.9.5

[87] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002. 7.2

[88] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. CACHECA: A cache language model based code suggestion tool. In *International Conference on Software Engineering (ICSE)*, volume 2, pages 705–708. IEEE, 2015. 1, 4.1

[89] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–49, 2015. 4.6.1

[90] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *ArXiv preprint*, abs/2204.05999, 2022. URL https://arxiv.org/abs/2204.05999. 7.1

[91] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *ArXiv preprint*, abs/2101.00027, 2021. URL https://arxiv.org/abs/2101.00027. 3.1, 3.2.2

[92] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. *CoRR*, abs/2212.10496, 2022. doi: 10.48550/arXiv.2212.10496. URL https://doi.org/10.48550/arXiv.2212.10496. 8.3.2

[93] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023. 9.6

[94] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL https://aclanthology.org/2021.emnlp-main.552. 7.1, 7.3.1, 7.3.1, 7.4

[95] Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press, 2006. 4.5, 4.5

[96] Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. *Transactions of the Association for Computational Linguistics*, 9:346–361, 2021. 8.1, 8.5, **??**

[97] J. Ginsparg. Natural language processing in an automatic programming domain. 1978. 4.9.1

[98] John M. Giorgi, Luca Soldaini, Bo Wang, Gary D. Bader, Kyle Lo, Lucy Lu Wang, and Arman Cohan. Exploring the challenges of open domain multi-document summarization. *CoRR*, abs/2212.10526, 2022. doi: 10.48550/arXiv.2212.10526. URL https://doi.org/10.48550/arXiv.2212.10526. 8.1, 8.5

[99] Daniel Gordon, Aniruddha Kembhavi, Mohammad Rastegari, Joseph Redmon, Dieter Fox, and Ali Farhadi. IQA: visual question answering in interactive environments. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4089–4098. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00430. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Gordon_IQA_Visual_Question_CVPR_2018_paper.html. 1.4, 9.1

[100] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016. 5.2, 5.3

[101] Edouard Grave, Moustapha Cissé, and Armand Joulin. Unbounded cache model for online language modeling with open vocabulary. *arXiv preprint arXiv:1711.02604*, 2017. 1.3, 5.1, 5.3, 6.1

[102] Andreas Grivas, Nikolay Bogoychev, and Adam Lopez. Low-rank softmax can have unargmaxable classes in theory but rarely in practice. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6738–6758, 2022. 6.6, 6.6.4

[103] Barbara J Grosz, Douglas E Appelt, Paul A Martin, and Fernando CN Pereira. Team: An experiment in the design of transportable natural-language interfaces. *Artificial Intelligence*, 32(2):173–243, 1987. 1

[104] Shuchi Grover, S. Basu, and Patricia K. Schank. What we can learn about student learning from open-ended programming projects in middle school computer science. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018. 4.4.4

[105] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, pages 631–642. ACM, 2016. URL https://dl.acm.org/citation.cfm?id=2950334. 2.1

[106] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018. 4.9.2, 4.9.3

[107] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46(1):317–330, 2011. 1, 4.1

[108] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3887–3896. PMLR, 2020. URL http://proceedings.mlr.press/v119/guo20h.html. 7.9.6

[109] Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2023. 9.6

[110] Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics*, 6: 437–450, 2018. 1.3, 5.1, 5.2, 6.1

[111] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: Retrieval-augmented language model pre-training. *ArXiv preprint*, abs/2002.08909, 2020. URL https://arxiv.org/abs/2002.08909. 7.2, 7.7

[112] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. REALM: retrieval-augmented language model pre-training. *CoRR*, abs/2002.08909, 2020. URL https://arxiv.org/abs/2002.08909. 1.3, 8.1

[113] Sonia Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. *2013 35th International Conference on Software Engineering (ICSE)*, pages 842–851, 2013. 4.8, 4.9.2

[114] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, et al. Pre-trained models: Past, present and future. *AI Open*, 2021. 3.2

[115] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. A retrieve-and-

edit framework for predicting structured outputs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10073–10083, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/cd17d3ce3b64f227987cd92cd701cc58-Abstract.html. 4.8, 5.2, 7.7

[116] Hiroaki Hayashi, Zecong Hu, Chenyan Xiong, and Graham Neubig. Latent relation language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7911–7918, 2020. 5.3

[117] Hiroaki Hayashi, Prashant Budania, Peng Wang, Chris Ackerson, Raj Neervannan, and Graham Neubig. Wikiasp: A dataset for multi-domain aspect-based summarization. *Trans. Assoc. Comput. Linguistics*, 9:211–225, 2021. doi: 10.1162/tacl_a_00362. URL https://doi.org/10.1162/tacl_a_00362. 8.1, 8.5, **??**

[118] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1111. URL https://www.aclweb.org/anthology/D18-1111. 4.8

[119] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1111. URL https://aclanthology.org/D18-1111. 7.7

[120] Junxian He, Taylor Berg-Kirkpatrick, and Graham Neubig. Learning sparse prototypes for text generation. *arXiv preprint arXiv:2006.16336*, 2020. 1.3, 5.1, 5.2, 6.1

[121] Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick. Efficient nearest neighbor language models. *arXiv preprint arXiv:2109.04212*, 2021. 6.1, 6.5.2, 6.8.2

[122] Andrew Head, Elena Leah Glassman, Gustavo Soares, R. Suzuki, Lucas Figueredo, L. D'Antoni, and B. Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 2017. 4.9.5

[123] Andrew Head, Elena Leah Glassman, B. Hartmann, and Marti A. Hearst. Interactive

extraction of examples from existing code. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018. 4.9.5

[124] George E. Heidorn. Automatic programming through natural language dialogue: A survey. *IBM Journal of research and development*, 20(4):302–313, 1976. 4.9.1

[125] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017. 1.1, 3.1, 5.1, 5.4, 5.6.2, 5.7

[126] Vincent J. Hellendoorn and Anand Ashok Sawant. The growing cost of deep learning for source code. *Commun. ACM*, 65(1):31–33, dec 2021. ISSN 0001-0782. doi: 10.1145/3501261. URL https://doi.org/10.1145/3501261. 3.1

[127] Gary G Hendrix. Natural-language interface. *American Journal of Computational Linguistics*, 8(2):56–61, 1982. 1

[128] Gary G Hendrix, Earl D Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3(2):105–147, 1978. 1

[129] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *CoRR*, abs/2009.03300, 2020. URL https://arxiv.org/abs/2009.03300. 8.1

[130] E. Hill, Manuel Roldan-Vega, J. Fails, and Greg Mallet. Nl-based query refinement and contextualized code search results: A user study. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 34–43, 2014. 4.8, 4.9.2

[131] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 1.1, 3.1, 5.1, 5.4, 5.6.1

[132] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015. 6.6.7

[133] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing A multi-hop QA dataset for comprehensive evaluation of reasoning steps. In Donia Scott, Núria Bel, and Chengqing Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December*

8-13, 2020, pages 6609–6625. International Committee on Computational Linguistics, 2020. doi: 10.18653/v1/2020.coling-main.580. URL https://doi.org/10.18653/v1/2020.coling-main.580. 8.1, 8.5, ??

[134] Joseph L Hodges Jr and Erich L Lehmann. Estimates of location based on rank tests. *The Annals of Mathematical Statistics*, pages 598–611, 1963. 4.6.3

[135] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rygGQyrFvH. 3.3, 6.5.2, 6.6.2, 7.4.2

[136] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR, 2022. URL https://proceedings.mlr.press/v162/huang22a.html. 9.6

[137] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. 3.2.2, 4.6.2, 4.9.2, 4.9.3

[138] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016. 35, 4.9.2

[139] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018. 4.2, 4.9.1, 4.9.3

[140] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1192. URL https://www.aclweb.org/anthology/D18-1192. 1.1, 2.1

[141] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium, 2018. Association for

Computational Linguistics. doi: 10.18653/v1/D18-1192. URL https://aclanthology.org/D18-1192. 1.3, 7.1, 7.7

[142] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. In Paola Merlo, Jörg Tiedemann, and Reut Tsarfaty, editors, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, pages 874–880. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.eacl-main.74. URL https://doi.org/10.18653/v1/2021.eacl-main.74. 1.3, 8.1

[143] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.74. URL https://aclanthology.org/2021.eacl-main.74. 7.1, 7.3.2, 7.9.4, 7.9.6

[144] Gautier Izacard, Patrick S. H. Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. *CoRR*, abs/2208.03299, 2022. doi: 10.48550/arXiv.2208.03299. URL https://doi.org/10.48550/arXiv.2208.03299. 1.3, 8.1, 8.1

[145] Jürgen M Janas. The semantics-based natural language interface to relational databases. In *Cooperative Interfaces to Information Systems*, pages 143–188. Springer, 1986. 1

[146] Yacine Jernite, Kavya Srinet, Jonathan Gray, and Arthur Szlam. CraftAssist Instruction Parsing: Semantic Parsing for a Minecraft Assistant. *ArXiv preprint*, abs/1905.01978, 2019. URL https://arxiv.org/abs/1905.01978. 1.4, 9.1

[147] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. How can we know what language models know. *Trans. Assoc. Comput. Linguistics*, 8:423–438, 2020. doi: 10.1162/tacl_a_00324. URL https://doi.org/10.1162/tacl_a_00324. 8.1

[148] Zhengbao Jiang, Jun Araki, Haibo Ding, and Graham Neubig. How can we know *When* language models know? on the calibration of language models for question answering. *Trans. Assoc. Comput. Linguistics*, 9:962–977, 2021. doi: 10.1162/tacl_a_00407. URL https://doi.org/10.1162/tacl_a_00407. 8.3.2

[149] Zhengbao Jiang, Luyu Gao, Jun Araki, Haibo Ding, Zhiruo Wang, Jamie Callan, and Graham Neubig. Retrieval as attention: End-to-end learning of retrieval and reading

within a single transformer. *CoRR*, abs/2212.02027, 2022. doi: 10.48550/arXiv.2212.02027. URL https://doi.org/10.48550/arXiv.2212.02027. 1.3, 8.1

[150] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. 6.5.1, 7.9.6

[151] Paul CD Johnson. Extension of nakagawa & schielzeth's r^2_{GLMM} to random slopes models. *Methods in Ecology and Evolution*, 5(9):944–946, 2014. 4.5

[152] K. Sparck Jones, S. Walker, and S.E. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 1. *Information Processing & Management*, 36(6):779 – 808, 2000. ISSN 0306-4573. doi: https://doi.org/10.1016/S0306-4573(00)00015-7. URL http://www.sciencedirect.com/science/article/pii/S0306457300000157. 2.2.4

[153] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1601–1611. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1147. URL https://doi.org/10.18653/v1/P17-1147. 8.1

[154] Armand Joulin, Moustapha Cissé, David Grangier, Hervé Jégou, et al. Efficient softmax approximation for gpus. In *International conference on machine learning*, pages 1302–1310. PMLR, 2017. 6.3

[155] Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, Deep Ganguli, Danny Hernandez, Josh Jacobson, Jackson Kernion, Shauna Kravec, Liane Lovitt, Kamal Ndousse, Catherine Olsson, Sam Ringer, Dario Amodei, Tom Brown, Jack Clark, Nicholas Joseph, Ben Mann, Sam McCandlish, Chris Olah, and Jared Kaplan. Language models (mostly) know what they know. *CoRR*, abs/2207.05221, 2022. doi: 10.48550/arXiv.2207.05221. URL https://doi.org/10.48550/arXiv.2207.05221. 8.1, 8.3.2

[156] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020. 3.2.1

[157] Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. Learning adaptive language interfaces through decomposition. In *Proceedings of the First Workshop on Interactive and Executable Semantic Parsing*, pages 23–33, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.intexsempar-1.4. URL https://www.aclweb.org/anthology/2020.intexsempar-1.4. 4.8

[158] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE, 2020. 1.1, 3.1, 5.1, 5.4, 5.6.1, 5.2

[159] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550. 7.3.1, 7.7

[160] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6769–6781. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.550. URL https://doi.org/10.18653/v1/2020.emnlp-main.550. 8.3.3

[161] I. Keivanloo, J. Rilling, and Ying Zou. Spotting working code examples. In *36th International Conference on Software Engineering (ICSE)*, 2014. 4.9.2

[162] Mary Beth Kery and B. Myers. Exploring exploratory programming. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29, 2017. 4.9.5

[163] Mary Beth Kery, Amber Horvath, and B. Myers. Variolite: Supporting exploratory programming by data scientists. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI)*, 2017. 4.9.5

[164] Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*, 2020. 1.3, 6.5.1

[165] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Gen-

eralization through memorization: Nearest neighbor language models. In *Proceedings of ICLR*, 2020. 1.3, 5.1, 5.2, 5.2, 5.3, 5.4, 5.2, 5.6.2

[166] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*, 2020. 1.3, 6.1, 6.1, 6.2, 6.2, 6.3, 1, 6.6, 6.6.4

[167] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=HklBjCEKvH. 8.1, 8.1, 8.4, 8.10.1

[168] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR*, abs/2212.14024, 2022. doi: 10.48550/arXiv.2212.14024. URL https://doi.org/10.48550/arXiv.2212.14024. 8.1

[169] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *CoRR*, abs/2210.02406, 2022. doi: 10.48550/arXiv.2210.02406. URL https://doi.org/10.48550/arXiv.2210.02406. 8.1

[170] Sanjeev Khudanpur and Jun Wu. Maximum entropy techniques for exploiting syntactic, semantic and collocational dependencies in language modeling. *Computer Speech & Language*, 14(4):355–372, 2000. 5.3

[171] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *ArXiv preprint*, abs/2303.17491, 2023. URL https://arxiv.org/abs/2303.17491. 9.6

[172] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980. 5.5

[173] A. Ko and B. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI 2004 Conference on Human Factors in Computing Systems (CHI)*, 2004. 4.9.5

[174] A. Ko and B. Myers. Debugging reinvented. *2008 ACM/IEEE 30th International Conference*

on Software Engineering (ICSE), pages 301–310, 2008. 4.9.5

[175] Amy Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 199–206. IEEE, 2004. 4.1

[176] Ned Kock and Gary Lynn. Lateral collinearity and misleading results in variance-based sem: An illustration and recommendations. *Journal of the Association for information Systems*, 13(7), 2012. 4.5

[177] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017. 9.6

[178] Kalpesh Krishna, Aurko Roy, and Mohit Iyyer. Hurdles to progress in long-form question answering. In *North American Association for Computational Linguistics*, 2021. 8.9

[179] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1160. URL https://www.aclweb.org/anthology/D17-1160. 1.1, 2.1

[180] S. Kulal, Panupong Pasupat, K. Chandra, Mina Lee, Oded Padon, A. Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019. 4.9.3, 4.9.5

[181] Nate Kushman and R. Barzilay. Using semantic unification to generate regular expressions from natural language. In *The 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*, 2013. 4.9.1

[182] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/569ff987c643b4bedf504efda8f786c2-Abstract.html. 9.6

[183] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh,

Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026. (1)

[184] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Trans. Assoc. Comput. Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://doi.org/10.1162/tacl_a_00276. 8.1

[185] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016. 4.11.5

[186] Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. Internet-augmented language models through few-shot prompting for open-domain question answering. *CoRR*, abs/2203.05115, 2022. doi: 10.48550/arXiv.2203.05115. URL https://doi.org/10.48550/arXiv.2203.05115. 1.3, 8.1

[187] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. *ACM SIGPLAN Notices*, 49(6):542–553, 2014. 1, 4.1

[188] Yann LeCun. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, 62, 2022. 1.4, 9.1

[189] Haejun Lee, Akhil Kedia, Jongwon Lee, Ashwin Paranjape, Christopher D. Manning, and Kyoung-Gu Woo. You only need one model for open-domain question answering. *CoRR*, abs/2112.07381, 2021. URL https://arxiv.org/abs/2112.07381. 1.3, 8.1

[190] Kenton Lee, Mandar Joshi, Iulia Raluca Turc, Hexiang Hu, Fangyu Liu, Julian Martin Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. In *International Conference on Machine Learning*, pages 18893–18912. PMLR, 2023. 9.6

[191] Tao Lei, F. Long, R. Barzilay, and M. Rinard. From natural language specifications to program input parsers. In *The 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2013. 4.9.1

[192] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003. 1.3, 7.1

[193] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019. 3.2.1

[194] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html. 7.2, 7.7

[195] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html. 1.3, 8.1

[196] Fei Li and Hosagrahar V Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014. 1

[197] Junyi Li, Tianyi Tang, Wayne Xin Zhao, Jingyuan Wang, Jian-Yun Nie, and Ji-Rong Wen. The web can be your oyster for improving large language models. *CoRR*, abs/2305.10998, 2023. doi: 10.48550/arXiv.2305.10998. URL https://doi.org/10.48550/arXiv.2305.10998. 8.7

[198] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 1.1

[199] Toby Jia-Jun Li, Amos Azaria, and B. Myers. Sugilite: Creating multimodal smartphone

automation by demonstration. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI)*, 2017. 4.9.5

[200] Toby Jia-Jun Li, I. Labutov, X. Li, X. Zhang, W. Shi, Wanling Ding, Tom Michael Mitchell, and B. Myers. Appinite: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–114, 2018. 4.9.5

[201] Toby Jia-Jun Li, Marissa Radensky, J. Jia, Kirielle Singarajah, Tom Michael Mitchell, and B. Myers. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2019. 4.9.5

[202] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*, pages 577–589, 2019. 1

[203] Xinze Li, Yixin Cao, Muhao Chen, and Aixin Sun. Take a break in the middle: Investigating subgoals towards hierarchical script generation. *ArXiv preprint*, abs/2305.10907, 2023. URL https://arxiv.org/abs/2305.10907. 9.6

[204] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile UI action sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8198–8210, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.729. URL https://aclanthology.org/2020.acl-main.729. 1.4, 9.1, 9.2.3, 9.6

[205] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on Freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23–33, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1003. URL https://www.aclweb.org/anthology/P17-1003. 1.1, 2.1

[206] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023. 9.6

[207] H. Lieberman, F. Paternò, Markus Klann, and V. Wulf. End-user development: An emerging paradigm. In *End User Development*, 2006. 4.9.5

[208] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W04-1013. 8.5

[209] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*, Miyazaki, Japan, May 2018. European Languages Resources Association (ELRA). URL https://www.aclweb.org/anthology/L18-1491. 1.1, 2.1

[210] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1491. 7.4.1, 7.9.1

[211] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1057. URL https://www.aclweb.org/anthology/P16-1057. 1.1, 2.1

[212] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. The Association for Computer Linguistics, 2016. doi: 10.18653/v1/p16-1057. URL https://doi.org/10.18653/v1/p16-1057. 4.9.1

[213] C. Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and J. Grundy. Opportunities and challenges in code search tools. *ArXiv*, abs/2011.02297, 2020. 4.6.2

[214] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL

https://openreview.net/forum?id=ryTp3f-0-. 9.2.4, ??, 9.6, 9.8.3

[215] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9):195:1–195:35, 2023. doi: 10.1145/3560815. URL https://doi.org/10.1145/3560815. 8.5

[216] X. Liu, Beijun Shen, H. Zhong, and Jiangang Zhu. Expsol: Recommending online threads for exception-related bug reports. *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 25–32, 2016. 4.9.4

[217] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. 5.1

[218] Jieyi Long. Large language model guided tree-of-thought. *ArXiv preprint*, abs/2305.08291, 2023. URL https://arxiv.org/abs/2305.08291. 9.6

[219] Meili Lu, Xiaobing Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549. IEEE, 2015. 4.9.2

[220] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb. 3.2.1

[221] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. Data-driven program completion. *ArXiv preprint*, abs/1705.09042, 2017. URL https://arxiv.org/abs/1705.09042. 7.7

[222] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.90. 9.6

[223] Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Hannaneh Hajishirzi, and Daniel

Khashabi. When not to trust language models: Investigating effectiveness and limitations of parametric and non-parametric memories. *CoRR*, abs/2212.10511, 2022. doi: 10.48550/arXiv.2212.10511. URL https://doi.org/10.48550/arXiv.2212.10511. 8.7

[224] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10:16:1–16:15, 2010. 4.9.5

[225] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. *Introduction to information retrieval*. Cambridge university press, 2008. 4.9.3

[226] Mehdi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2013. 4.9.1

[227] Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. Generation-augmented retrieval for open-domain question answering. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4089–4100. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.316. URL https://doi.org/10.18653/v1/2021.acl-long.316. 8.3.2

[228] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.173. URL https://aclanthology.org/2020.acl-main.173. 8.1

[229] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2: 308–320, 1976. 4.11.5

[230] Clara Meister, Elizabeth Salesky, and Ryan Cotterell. Generalized entropy regularization or: There's nothing special about label smoothing. *arXiv preprint arXiv:2005.00820*, 2020. 6.6.7

[231] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8:795–809, 2020. 6.5.2

[232] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel

mixture models. *arXiv preprint arXiv:1609.07843*, 2016. 5.3, 5.4, 6.3

[233] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. In *Proceedings of ICLR*, 2018. 5.1, 6.1

[234] Rada Mihalcea, Hugo Liu, and Henry Lieberman. Nlp (natural language processing) for nlp (natural language programming). In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 319–330. Springer, 2006. 4.9.1

[235] Tomas Mikolov and Geoffrey Zweig. Context dependent recurrent neural network language model. In *2012 IEEE Spoken Language Technology Workshop (SLT)*, pages 234–239. IEEE, 2012. URL https://ieeexplore.ieee.org/document/6424228. 5.3

[236] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010. 1.3, 5.1

[237] Dipendra K Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research*, 35(1-3):281–300, 2016. 1.4, 9.1

[238] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5): 471–516, 2007. 4.1

[239] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890. IEEE, 2015. 4.8

[240] Yair Mundlak. On the pooling of time series and cross section data. *Econometrica: journal of the Econometric Society*, pages 69–85, 1978. 4.5

[241] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. Api designers in the field: Design practices and challenges for creating usable apis. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 249–258. IEEE, 2018. 4.1

[242] B. Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59:62 – 69, 2016. 4.9.5

[243] B. Myers, J. Pane, and A. Ko. Natural programming languages and environments. *Commun. ACM*, 47:47–52, 2004. 4.9.5

[244] Brad A Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59 (6):62–69, 2016. 4.1

[245] Brad A Myers, John F Pane, and Amy J Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9):47–52, 2004. 1

[246] Brad A Myers, Amy Ko, Thomas D LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7):44–52, 2016. 4.1

[247] Shinichi Nakagawa and Holger Schielzeth. A general and simple method for obtaining r2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution*, 4(2): 133–142, 2013. 4.5

[248] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *CoRR*, abs/2112.09332, 2021. URL https://arxiv.org/abs/2112.09332. 1.3, 8.1, 8.7

[249] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021. 9.6

[250] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. Marble: Mining for boilerplate code to identify api usability problems. In *International Conference on Automated Software Engineering (ASE)*, pages 615–627. IEEE, 2019. 1, 4.1

[251] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994. 6.8.3

[252] T. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remaui (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259, 2015. 4.9.5

[253] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint*, 2022. 1.3, 7.1

[254] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023. 1.1

[255] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023. 1.1

[256] Lorelli S Nowell, Jill M Norris, Deborah E White, and Nancy J Moules. Thematic analysis: Striving to meet the trustworthiness criteria. *International Journal of Qualitative Methods*, 16(1):1609406917733847, 2017. 4.6.1, 4.7

[257] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 133–141, 2002. 1.3, 7.1

[258] William C Ogden and Philip Bernick. Using natural language interfaces. In *Handbook of human-computer interaction*, pages 137–161. Elsevier, 1997. 1

[259] OpenAI. Chatgpt: Optimizing language models for dialogue. 2022. 9.5.1

[260] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/arXiv.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774. 8.1

[261] OpenAI. Gpt-4 technical report. *arXiv*, pages 2303–08774, 2023. 9.5.1

[262] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *CoRR*, abs/2203.02155, 2022. doi: 10.48550/arXiv.2203.02155. URL https://doi.org/10.48550/arXiv.2203.02155. 8.1, 8.1

[263] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022. 9.8.8

[264] John F Pane, Brad A Myers, et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54

(2):237–264, 2001. 1

[265] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://www.aclweb.org/anthology/P02-1040. 1.2, 4.1, 4.3, 4.9.3

[266] Emilio Parisotto, Abdel rahman Mohamed, R. Singh, L. Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *5th International Conference on Learning Representations (ICLR)*, 2017. 4.9.1

[267] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.232. URL https://aclanthology.org/2021.findings-emnlp.232. 7.2, 7.5.1, 7.7

[268] Panupong Pasupat, Yuan Zhang, and Kelvin Guu. Controllable semantic parsing via retrieval augmentation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7683–7698, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.607. URL https://aclanthology.org/2021.emnlp-main.607. 7.2, 7.5.1, 7.7

[269] Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *CoRR*, abs/2302.12813, 2023. doi: 10.48550/arXiv.2302.12813. URL https://doi.org/10.48550/arXiv.2302.12813. 8.7

[270] Gabriel Pereyra, George Tucker, Jan Chorowski, Łukasz Kaiser, and Geoffrey Hinton. Regularizing neural networks by penalizing confident output distributions. *arXiv preprint arXiv:1701.06548*, 2017. 6.6.7

[271] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick S. H. Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander H. Miller. Language models as knowledge bases? In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China,*

November 3-7, 2019, pages 2463–2473. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1250. URL https://doi.org/10.18653/v1/D19-1250. 8.1

[272] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *ACM Technical Symposium on Computer Science Education*, pages 153–160. ACM, 2012. 1

[273] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *International Conference on Software Engineering (ICSE)*, pages 1295–1298. IEEE, 2013. 4.3, 4.3, 4.9.4

[274] Luca Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining stack overflow to turn the ide into a self-confident programming prompter. In *International Conference on Mining Software Repositories (MSR)*, 2014. 4.8, 4.9.4

[275] Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022. 8.1, 8.3.2, 8.4, 8.1, 8.6.1

[276] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. Naturaljava: a natural language interface for programming in java. In *International Conference on Intelligent User Interfaces (IUI)*, pages 207–211, 2000. 4.1

[277] Sebastian Proksch, Sven Amann, and Sarah Nadi. Enriched event streams: a general dataset for empirical studies on in-ide activities of software developers. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 62–65, 2018. 4.9.4, 4.9.6

[278] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for MOOCs. In *Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH)*, pages 39–40. ACM, 2016. 1

[279] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8494–8502. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00886. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Puig_VirtualHome_Simulating_Household_CVPR_2018_paper.html. 1.4, 9.1, 9.1, ??, 9.6

[280] Hongjing Qian, Yutao Zhu, Zhicheng Dou, Haoqi Gu, Xinyu Zhang, Zheng Liu, Ruofei Lai, Zhao Cao, Jian-Yun Nie, and Ji-Rong Wen. Webbrain: Learning to generate factually

correct articles for queries by grounding on large web corpus. *CoRR*, abs/2304.04358, 2023. doi: 10.48550/arXiv.2304.04358. URL https://doi.org/10.48550/arXiv.2304.04358. 1.3, 8.1

[281] Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, Ruobing Xie, Fanchao Qi, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Webcpm: Interactive web search for chinese long-form question answering. *CoRR*, abs/2305.06849, 2023. doi: 10.48550/arXiv.2305.06849. URL https://doi.org/10.48550/arXiv.2305.06849. 8.7

[282] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1085. URL https://www.aclweb.org/anthology/P15-1085. 1.1, 2.1

[283] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://www.aclweb.org/anthology/P17-1105. 1.1, 2.1

[284] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://aclanthology.org/P17-1105. 7.7

[285] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 3.4.3, 6.8.1

[286] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1 (8), 2019. URL https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf. 8.5

[287] Karthik Radhakrishnan, Arvind Srikantan, and Xi Victoria Lin. Colloql: Robust text-to-sql over search queries. In *Proceedings of the First Workshop on Interactive and Executable*

Semantic Parsing, pages 34–45, 2020. 4.8

[288] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html. 3.2.1, 5.1, 7.3.2

[289] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html. 7.1

[290] Mukund Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean - code search and idiomatic snippet synthesis. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 357–367, 2016. 4.9.1

[291] M. M. Rahman and C. Roy. Surfclipse: Context-aware meta-search in the ide. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 617–620, 2014. 4.9.4

[292] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 194–203. IEEE, 2014. 4.3

[293] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264. 9.3.2

[294] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2124. URL https://aclanthology.org/P18-2124. 9.3.2

[295] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083*, 2023. 8.1, 8.2.1, 8.4, 8.1, 8.10.1

[296] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical

language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014. 1, 1.1, 3.1, 4.1, 5.1

[297] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 2015. 4.9.1

[298] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. 4.9.3

[299] Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th International Conference on World Wide Web*, pages 521–530, 2007. 4.6.1

[300] Kelly Rivers, Erik Harpstead, and Kenneth R Koedinger. Learning curve analysis for programming: Which concepts do students struggle with? In *ICER*, volume 16, pages 143–151. ACM, 2016. 1

[301] Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model? In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 5418–5426. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.437. URL https://doi.org/10.18653/v1/2020.emnlp-main.437. 8.1

[302] Stephen E Robertson and K Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146, 1976. 7.1, 7.3.1

[303] Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019. 8.3.3

[304] Pedro Rodriguez and Jordan Boyd-Graber. Evaluation paradigms in question answering. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9630–9642, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.758. URL https://aclanthology.org/2021.emnlp-main.758. 7.6.2

[305] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE, 2012. 7.2

[306] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 287–298. IEEE, 2020. 4.6.1

[307] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 1.1

[308] Devendra Singh Sachan, Siva Reddy, William L. Hamilton, Chris Dyer, and Dani Yogatama. End-to-end training of multi-document reader and retriever for open-domain question answering. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 25968–25981, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/da3fde159d754a2555eaa198d2d105b2-Abstract.html. 1.3, 8.1

[309] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 191–201, 2015. 4.9.2

[310] Apurvanand Sahay, Arsene Indamutsa, D. D. Ruscio, and A. Pierantonio. Supporting the understanding and comparison of low-code development platforms. *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020. 4.9.5

[311] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019. 6.8.1

[312] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. 8.3.1, 11

[313] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015. 3.4.2, 5.4, 6.3

[314] Rico Sennrich, Barry Haddow, and Alexandra Birch. Controlling politeness in neural machine translation via side constraints. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language*

Technologies, pages 35–40, 2016. 5.3

[315] Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th annual symposium on user interface software and technology*, pages 365–377, 2016. 1

[316] Ehsan Shareghi, Gholamreza Haffari, and Trevor Cohn. Compressed nonparametric language modelling. In *IJCAI*, pages 2701–2707, 2017. 5.2

[317] Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *arXiv preprint arXiv:2306.00245*, 2023. 9.6

[318] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.231. 7.4.1, 7.9.1

[319] Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144. PMLR, 2017. URL http://proceedings.mlr.press/v70/shi17a.html. 1.4, 9.1, 9.2.3, 9.2.4, **??**, 9.6, 9.8.3

[320] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. REPLUG: retrieval-augmented black-box language models. *CoRR*, abs/2301.12652, 2023. doi: 10.48550/arXiv.2301.12652. URL https://doi.org/10.48550/arXiv.2301.12652. 1.3, 8.1

[321] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. Program synthesis and semantic parsing with learned code idioms. *33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019. 4.9.5

[322] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *ArXiv preprint*, abs/2303.11366, 2023. URL https://arxiv.org/abs/2303.11366. 9.6

[323] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh

Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10737–10746. IEEE, 2020. doi: 10.1109/CVPR42600.2020.01075. URL https://doi.org/10.1109/CVPR42600.2020.01075. 1.4, 9.1, **??**, 9.6

[324] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn. 1.4, 9.1

[325] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *ICML 2022 Workshop on Knowledge Retrieval and Language Models*. 7.7

[326] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007. 3

[327] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013. 1

[328] Armando Solar-Lezama. Program synthesis by sketching. 2008. 4.9.5

[329] Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. Restgpt: Connecting large language models with real-world applications via restful apis. *arXiv preprint arXiv:2306.06624*, 2023. 11

[330] Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and Bill Dolan. A neural network approach to context-sensitive generation of conversational responses. In *Proceedings of NAACL*, 2015. 5.1

[331] Abishek Sridhar, Robert Lo, Frank F Xu, Hao Zhu, and Shuyan Zhou. Hierarchical prompting assists large language model on web navigation. *arXiv preprint arXiv:2305.14257*, 2023. 9.6

[332] Felix Stahlberg and Bill Byrne. On nmt search errors and model errors: Cat got your tongue? *arXiv preprint arXiv:1908.10090*, 2019. 6.5.2

[333] Ivan Stelmakh, Yi Luan, Bhuwan Dhingra, and Ming-Wei Chang. ASQA: factoid questions meet long-form answers. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 8273–8288. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.emnlp-main.566. 8.1, 8.5, ??

[334] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. *International Conference on Software Engineering (ICSE)*, 2014. 4.9.4

[335] Alane Suhr, Srinivasan Iyer, and Yoav Artzi. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2238–2249, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1203. URL https://www.aclweb.org/anthology/N18-1203. 1.1, 2.1

[336] Zhiqing Sun, Xuezhi Wang, Yi Tay, Yiming Yang, and Denny Zhou. Recitation-augmented language models. *CoRR*, abs/2210.01296, 2022. doi: 10.48550/arXiv.2210.01296. URL https://doi.org/10.48550/arXiv.2210.01296. 8.3.2

[337] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. 6.6.7

[338] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/arXiv.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971. 8.1

[339] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 1.1

[340] Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. Androidenv: A reinforcement learning platform for android. *ArXiv preprint*, abs/2105.13231, 2021. URL https://arxiv.org/abs/2105.13231. ??, 9.6

[341] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE, 2019. 1.2, 4.1, 4.3, 4.9.3

[342] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *CoRR*, abs/2212.10509, 2022. doi: 10.48550/arXiv.2212.10509. URL https://doi.org/10.48550/arXiv.2212.10509. 8.1, 8.2.1, 8.4, 8.5, 8.1, 8.10.1, 8.10.2

[343] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014. 4.8, 5.6.2

[344] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers*. " O'Reilly Media, Inc.", 2022. 3.1, 3.2.1

[345] David Vadas and James R Curran. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*, pages 191–199, 2005. 4.1

[346] Neeraj Varshney, Man Luo, and Chitta Baral. Can open-domain QA reader utilize external knowledge efficiently like humans? *CoRR*, abs/2211.12707, 2022. doi: 10.48550/arXiv.2211.12707. URL https://doi.org/10.48550/arXiv.2211.12707. 8.3.2

[347] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of NeurIPS*, 2017. 1.3, 5.1, 5.4

[348] Venkatesh Vinayakarao, A. Sarma, R. Purandare, Shuktika Jain, and Saumya Jain. Anne: Improving source code search using entity retrieval approach. In *WSDM '17*, 2017. 4.9.2

[349] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021. 1.1, 3.1, 3.2.1

[350] Dexin Wang, Kai Fan, Boxing Chen, and Deyi Xiong. Efficient cluster-based k-nearest-neighbor machine translation. *ArXiv*, abs/2204.06175, 2022. 6.1

[351] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *ArXiv preprint*, abs/2305.16291, 2023. URL https://arxiv.org/abs/

2305.16291. 9.5.2, 9.6

[352] Tian Wang and Kyunghyun Cho. Larger-context language modelling with recurrent neural network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1319–1329, 2016. 5.3

[353] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *CoRR*, abs/2203.11171, 2022. doi: 10.48550/arXiv.2203.11171. URL https://doi.org/10.48550/arXiv.2203.11171. 8.5

[354] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685. 1.3, 3.2.1, 7.1, 7.3.1, 7.3.2, 7.4

[355] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pages 1332–1342, 2015. 4.9.3

[356] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. *ArXiv preprint*, abs/2212.10481, 2022. URL https://arxiv.org/abs/2212.10481. 1.4, 7.4.2, 9.1

[357] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL https://arxiv.org/abs/2201.11903. 8.1, 8.5, 8.10.2

[358] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022. 9.4

[359] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. Building Bing Developer Assistant. Technical report, MSR-TR-2015-36, Microsoft Research, 2015. 1, 4.1, 4.2, 4.3, 4.3, 4.8, 4.9.2

[360] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media,

2012. 3

[361] Frank Wood, Jan Gasthaus, Cédric Archambeau, Lancelot James, and Yee Whye Teh. The sequence memoizer. *Communications of the ACM*, 54(2):91–98, 2011. 5.2

[362] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=TrjbxzRcnf-. 7.7

[363] Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1341–1350, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1127. URL https://www.aclweb.org/anthology/P16-1127. 1.1, 2.1

[364] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.538. URL https://aclanthology.org/2020.acl-main.538. 1.1, 1.3, 4.3, 4.6.2, 4.9.1, 4.9.3, 7.1

[365] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *ArXiv preprint*, abs/2202.13169, 2022. URL https://arxiv.org/abs/2202.13169. 7.1

[366] Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica Lam. Grounding open-domain instructions to automate web support tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1022–1032, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.80. URL https://aclanthology.org/2021.naacl-main.80. 1.4, 9.1, 9.6

[367] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W Cohen. Breaking the softmax bottleneck: A high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017. 6.4.2, 6.4.2, 6.8.2

[368] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable

multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259. (1), 9.3.2

[369] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 5754–5764, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/dc6a7e655d7e5840e66733e9ee67cc69-Abstract.html. 5.1

[370] Zhixian Yang, Renliang Sun, and Xiaojun Wan. Nearest neighbor knowledge distillation for neural machine translation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5546–5556, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.406. URL https://aclanthology.org/2022.naacl-main.406. 6.6, 6.6.7

[371] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. volume abs/2207.01206, 2022. URL https://arxiv.org/abs/2207.01206. 1.4, 9.1, **??**, 9.6, 9.8.1

[372] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *CoRR*, abs/2210.03629, 2022. doi: 10.48550/arXiv.2210.03629. URL https://doi.org/10.48550/arXiv.2210.03629. 8.1, 8.1, 11

[373] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv preprint*, abs/2210.03629, 2022. URL https://arxiv.org/abs/2210.03629. 9.4

[374] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *ArXiv preprint*, abs/2305.10601, 2023. URL https://arxiv.org/abs/2305.10601. 9.6

[375] Xuchen Yao and Benjamin Van Durme. Information extraction over structured data: Question answering with freebase. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 956–966, 2014. 4.9.3

[376] Ziyu Yao, Daniel S. Weld, W. Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. *Proceedings of the 2018 World Wide Web Conference (WWW)*, 2018. 35, 4.9.3

[377] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703. International World Wide Web Conferences Steering Committee, 2018. URL https://dl.acm.org/citation.cfm?id=3186081. 1.1, 2.1

[378] Ziyu Yao, Xiujun Li, Jianfeng Gao, Brian Sadler, and Huan Sun. Interactive semantic parsing for if-then recipes via hierarchical reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 33, pages 2547–2554, 2019. 4.8

[379] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. *The World Wide Web Conference (WWW)*, 2019. 4.9.2

[380] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, pages 2203–2214. ACM, 2019. URL https://dl.acm.org/citation.cfm?id=3313632. 1.1, 2.1

[381] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL https://www.aclweb.org/anthology/P17-1041. 1.1, 2.1

[382] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL https://aclanthology.org/P17-1041. 1.3, 7.1, 7.7

[383] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, 2017. Association

for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL https://aclanthology.org/P17-1041. 4.2, 4.9.1, 4.9.3

[384] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2002. URL https://www.aclweb.org/anthology/D18-2002. 1.1, 2.1, 2.1, 2.3.1

[385] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *Conference on Empirical Methods in Natural Language Processing (EMNLP), Demo Track*, 2018. 4.3, 4.6.2, 4.9.1

[386] Pengcheng Yin and Graham Neubig. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1447. URL https://www.aclweb.org/anthology/P19-1447. 1.1, 2.1, 2.1, 2.3.1, 4.8

[387] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pages 476–486. IEEE, 2018. 1.2, 7.1, 7.4, 7.4.2, 7.9.2

[388] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408. 1.1, 2.1, 2.1, 2.2.2, 2.3.1, 4.2, 4.3, 4.3

[389] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://www.aclweb.org/anthology/D18-1425. 1.1, 2.1

[390] Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan,

Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1204. URL https://www.aclweb.org/anthology/D19-1204. 1.1, 2.1

[391] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1443. URL https://www.aclweb.org/anthology/P19-1443. 1.1, 2.1

[392] Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chenguang Zhu, Michael Zeng, and Meng Jiang. Generate rather than retrieve: Large language models are strong context generators. *CoRR*, abs/2209.10063, 2022. doi: 10.48550/arXiv.2209.10063. URL https://doi.org/10.48550/arXiv.2209.10063. 8.3.2

[393] Wenhao Yu, Zhihan Zhang, Zhenwen Liang, Meng Jiang, and Ashish Sabharwal. Improving language models via plug-and-play retrieval feedback. *CoRR*, abs/2305.14002, 2023. doi: 10.48550/arXiv.2305.14002. URL https://doi.org/10.48550/arXiv.2305.14002. 8.7

[394] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. *2nd Workshop on Neural Abstract Machines & Program Induction (NAMPI), ICML*, 2018. 4.9.3

[395] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996. 1.1, 2.1, 4.9.3

[396] Yury Zemlyanskiy, Michiel de Jong, Joshua Ainslie, Panupong Pasupat, Peter Shaw, Linlu Qiu, Sumit Sanghai, and Fei Sha. Generate-and-retrieve: Use your predictions to im-

prove retrieval for semantic parsing. In Nicoletta Calzolari, Chu-Ren Huang, Hansaem Kim, James Pustejovsky, Leo Wanner, Key-Sun Choi, Pum-Mo Ryu, Hsin-Hsi Chen, Lucia Donatelli, Heng Ji, Sadao Kurohashi, Patrizia Paggio, Nianwen Xue, Seokhwan Kim, Younggyun Hahm, Zhong He, Tony Kyungil Lee, Enrico Santus, Francis Bond, and Seung-Hoon Na, editors, *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pages 4946–4951. International Committee on Computational Linguistics, 2022. URL https://aclanthology.org/2022.coling-1.438. 8.7

[397] Luke Zettlemoyer and Michael Collins. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 678–687, 2007. 4.9.3

[398] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *CoRR*, abs/2303.12570, 2023. doi: 10.48550/arXiv.2303.12570. URL https://doi.org/10.48550/arXiv.2303.12570. 8.7

[399] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022. 8.1

[400] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *CoRR*, abs/2303.18223, 2023. doi: 10.48550/arXiv.2303.18223. URL https://doi.org/10.48550/arXiv.2303.18223. 8.1

[401] Ming Zhong, Yang Liu, Da Yin, Yuning Mao, Yizhu Jiao, Pengfei Liu, Chenguang Zhu, Heng Ji, and Jiawei Han. Towards a unified multi-dimensional evaluator for text generation. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 2023–2038. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.emnlp-main.131.

8.5

[402] Ruiqi Zhong, Mitchell Stern, and D. Klein. Semantic scaffolds for pseudocode-to-code generation. In *ACL*, 2020. 4.9.3, 4.9.5

[403] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. arxiv 2017. *ArXiv preprint*, abs/1709.00103, 2017. URL https://arxiv.org/abs/1709.00103. 1.4, 9.1

[404] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017. 1, 1.1, 2.1, 4.9.3

[405] Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. Rtfm: Generalising to novel environment dynamics via reading. *ArXiv preprint*, abs/1910.08210, 2019. URL https://arxiv.org/abs/1910.08210. 7.7

[406] Zexuan Zhong, Tao Lei, and Danqi Chen. Training language models with memory augmentation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5657–5673, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.382. 6.7

[407] Chunting Zhou, Graham Neubig, Jiatao Gu, Mona Diab, Francisco Guzmán, Luke Zettlemoyer, and Marjan Ghazvininejad. Detecting hallucinated content in conditional neural sequence generation. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1393–1404, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.120. URL https://aclanthology.org/2021.findings-acl.120. 8.1

[408] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. Doccoder: Generating code by retrieving and reading docs. *arXiv preprint arXiv:2207.05987*, 2022. 6.1

[409] Shuyan Zhou, Pengcheng Yin, and Graham Neubig. Hierarchical control of situated agents through natural language. In *Proceedings of the Workshop on Structured and Unstructured Knowledge Integration (SUKI)*, pages 67–84, Seattle, USA, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.suki-1.8. URL https://aclanthology.org/2022.suki-1.8. 9.5.2, 9.6

[410] Shuyan Zhou, Li Zhang, Yue Yang, Qing Lyu, Pengcheng Yin, Chris Callison-Burch, and Graham Neubig. Show me more details: Discovering hierarchies of procedures from

semi-structured web data. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2998–3012, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.214. URL https://aclanthology.org/2022.acl-long.214. 9.6

[411] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Xh5eMZVONGF. 3.1