

Tools and Techniques for Building Programming Assistants for Data Analysis

Rohan Bavishi



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-208

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-208.html>

August 12, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Tools and Techniques for Building Programming Assistants for Data Analysis

by

Rohan Jayesh Bavishi

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair

Professor Ion Stoica

Professor Joseph Hellerstein

Dr. Mukul Prasad

Summer 2022

Tools and Techniques for Building Programming Assistants for Data Analysis

Copyright 2022
by
Rohan Jayesh Bavishi

Abstract

Tools and Techniques for Building Programming Assistants for Data Analysis

by

Rohan Jayesh Bavishi

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

We live in the data age. Today, data analytics drives much of business decision-making, logistics, advertising, and recommendations. Data wrangling, profiling, and visualization are some of the key tasks in a data analytics workflow. These tasks also account for a majority of the time spent in data analysis. Academics and industry leaders have long attributed the disparity to the inherent domain-specific nature of data, which necessitates highly custom treatment for every new source of data. Specifying these custom analysis steps using low-level tools such as Excel can be prohibitively cumbersome. In response, much research has focused on smarter interactive and graphical tools for data processing and visualization tasks. Successful commercialization of this research has contributed to a \$3 billion *self-service* analytics industry.

However, analysts with a programming background have not adopted such tools as widely as their non-programmer colleagues have. The desire to avoid shuffling between tools and work in a single environment, as well as a need for the full, unbounded expressivity of programming-based analysis tools, are a few major reasons. This does not mean that programmers are immune to the specification burden; the expressivity of programming comes at the cost of complexity and steep learning curves. Novices have to spend much time learning these tools from books or fragmented resources online. Even experts report a loss in productivity from having to look up documentation frequently to get uninteresting details such as function names and argument values right.

Thus, there is a need for *programming assistants* that reconcile the need to reduce the specification burden for programmer analysts with their desire to work with code in their preferred development environments. These assistants should help programmer-analysts write code more efficiently by automatically generating human-readable and readily-integrable code from high-level specifications.

This dissertation introduces techniques and corresponding prototypical assistants that accept

input-output examples, demonstrations, or natural language specifications and automatically generate suitable data processing and visualization code utilizing popular data science libraries such as pandas, matplotlib, seaborn, and scikit-learn. Automatic code generation has long faced the tradeoff barrier between expressivity and performance/accuracy: supporting a large number of analysis tasks makes the problem of generating the right code quickly that much more difficult. Accordingly, prior research in program synthesis and semantic parsing has largely sacrificed full expressivity to support efficient code generation for a small but useful subset of tasks. The code-as-text approach of modern natural language processing systems, including the use of large language models, promises unbounded expressivity, but their sub-optimal accuracy remains a concern.

This dissertation tries to push the boundaries in terms of breaking this tradeoff barrier — can we build programming systems that are fully expressive while remaining fast and accurate? Specifically, this dissertation builds upon prior work and introduces novel code generation techniques that combine insights from synthesis, automated testing, program analysis, and machine learning. It contributes four core techniques and corresponding assistants, namely `AUTOPANDAS`, `GAUSS`, `VIZSMITH`, and `DATANA`. `AUTOPANDAS` and `GAUSS` constitute core advances in search space and algorithm design for example-based synthesis. `VIZSMITH` and `DATANA` introduce novel mining and auto-summarization techniques to automatically build aligned code and natural language corpora, which `DATANA` uses to greatly improve the code-generation capabilities of modern large language models. Compared to prior work, these assistants improve the expressivity of synthesis-based systems and the accuracy of machine-learning-based systems.

To my parents Mita and Jayesh, my brother Karan, and my partner Shivane.

Contents

Contents	ii
List of Figures	vi
List of Tables	ix
1 Introduction	1
1.1 Reducing the Specification Burden via Interaction	3
1.2 The Need for Programming Assistants	4
1.3 Choice of High-Level Specifications	5
1.4 Code Generation from High-Level Specifications	6
2 AutoPandas	8
2.1 Overview	10
2.2 Technique	14
2.2.1 Generators	14
2.2.2 Generator-Based Program Synthesis	16
2.2.2.1 Program Candidate Generator	17
2.2.2.2 Building an Exhaustive Depth-First Enumerative Synthesis Engine	18
2.2.2.3 Building a Smart Enumerative Synthesis Engine	19
2.2.3 Neural-Backed Generators for Pandas	19
2.2.3.1 Neural-Network Query	21
2.2.3.2 Query Encoding	21
2.2.3.3 Operator-Specific Graph Neural Network Models	23
2.2.4 Training Neural-Backed Generators for Pandas	26
2.3 Evaluation	27
2.3.1 Implementation	28
2.3.2 Training and Setup	28
2.3.3 RQ1: Performance on Real-World Benchmarks	28
2.3.4 RQ2: Analysis of Neural Network Models	29
2.3.4.1 Function Sequence Prediction Performance	29

2.3.4.2	Comparison with Deterministic and Randomized Semantics	30
2.4	Discussion	31
2.4.1	Generator Implementation	31
2.4.2	Representative Training Data	31
2.4.3	Ease of Providing I/O Examples	34
2.5	Summary	34
3	Gauss	35
3.1	Overview	37
3.1.1	Extracting Query Graphs	40
3.1.2	Deciding Skeletons for Exploration	41
3.1.3	Learning from Failures	42
3.1.4	Smart Enumeration	43
3.2	Technique	43
3.2.1	Preliminaries and Notation	43
3.2.1.1	Table Transformation Programs	43
3.2.1.2	Graphs	45
3.2.2	Graph Abstractions	46
3.2.3	Problem Statement	49
3.2.4	Synthesis Algorithm	49
3.2.4.1	Graph Decompositions	50
3.2.4.2	Overall Algorithm	54
3.2.4.3	Enumeration	54
3.2.4.4	The FEASIBLE Check	56
3.2.4.5	Strengthening Decompositions	57
3.2.4.6	The Oracle	59
3.2.4.7	Soundness and Completeness	60
3.2.5	User Interface Implementation	61
3.3	Evaluation	63
3.3.1	Baselines and Benchmarks and Hardware	64
3.3.2	RQ1: Pruning Power	65
3.3.3	RQ2: Reduction in Size of Specifications	66
3.4	Discussion	67
3.4.1	Necessity of a User Interface	67
3.4.2	Ease of Use	67
3.4.3	Multiple Possible Representations	68
3.4.4	Noise in Demonstrations	68
3.4.5	Experience with Real Users	68
3.5	Summary	69
4	VizSmith	71
4.1	Overview	72

4.2	Technique	74
4.2.1	Architecture of VIZSMITH	74
4.2.2	Mining	74
4.2.2.1	Collecting and Replaying Notebooks	74
4.2.2.2	Instrumentation and Execution	75
4.2.2.3	Visualization Objects and Visualization Slices	75
4.2.2.4	Minimizing Visualization Slices	77
4.2.3	Extracting Visualization Functions	78
4.2.3.1	Participating Columns vs. Column Parameters	81
4.2.4	Analysis of Mined Visualization Functions	82
4.2.4.1	Defining Reusability	83
4.2.4.2	Metamorphic Testing for Checking Reusability	83
4.2.5	Visualization Code Generation	85
4.2.5.1	Search	86
4.2.5.2	Generating Visualizations	86
4.3	Evaluation	87
4.3.1	RQ1: Diversity of Functionality in Mined Corpus	87
4.3.2	RQ2: Accuracy of Metamorphic Testing	89
4.3.3	RQ3: Code Generation Performance	90
4.4	Discussion	91
4.4.1	Real-World Usage	91
4.4.2	Code Licensing and Security	91
4.4.3	Construct Validity	92
4.4.4	Integrating Direct Manipulation	92
4.5	Summary	92
5	Datana	94
5.1	Motivation	96
5.1.1	Correcting Algorithmic or Functional Errors	97
5.1.2	Highlighting Important Query Fragments	98
5.1.3	Providing Solutions with Low Adaptation Overhead	98
5.2	Technique	99
5.2.1	Architecture Overview	99
5.2.2	Mining Pandas Expressions	101
5.2.3	Auto Code Summarization using Large Language Models	102
5.2.4	Bidirectional Consistency	103
5.2.4.1	Better Approximating Semantic Equivalence	105
5.2.5	Generating Imprecise and Incomplete Descriptions	106
5.2.5.1	Automatic Parameterization of Code Snippets	107
5.2.6	Augmenting Codex for Improved Code Generation	108
5.2.6.1	Retrieval	109
5.2.6.2	Prompt Engineering for Augmenting Codex	109

5.3	Evaluation	110
5.3.1	Benchmarks	110
5.3.2	Baselines and Other Systems	111
5.3.2.1	Corpus Baselines	111
5.3.2.2	Codex Baseline	112
5.3.2.3	Simple Corpus Baseline	112
5.3.2.4	Jigsaw Results	112
5.3.3	Codex Versions	112
5.3.4	RQ1: End-to-End Performance Improvement Over Codex and Search Baselines	112
5.3.5	RQ2: Comparison With Jigsaw	113
5.3.6	RQ3: Complementarity with Core LLM Improvements	113
5.3.7	RQ4: Ablation Study	114
5.3.7.1	Effect of Varying Top-k	114
5.3.7.2	Ablations for Retrieval Architecture and Description Styles	114
5.4	Discussion	116
5.4.1	Disambiguation and Ranking	116
5.4.2	Contextual Discoverability	116
5.5	Summary	116
6	Related Work	118
6.1	Code Generation	118
6.1.1	Intent Specifications	118
6.1.1.1	Logical Specifications	118
6.1.1.2	Input-Output Examples	119
6.1.1.3	Natural Language	119
6.1.1.4	Demonstrations	119
6.1.1.5	Combinations of Specifications	119
6.1.2	Algorithms	120
6.1.2.1	Pruning via Logical Reasoning.	120
6.1.2.2	Pruning via Domain-Specific Inductive Synthesis	120
6.1.2.3	Pruning via Abstractions	121
6.1.2.4	Pruning via Type Information	121
6.1.2.5	Biasing via Machine Learning	121
6.1.2.6	Generation via Machine Learning	122
6.2	Alternate Programming-Based Data Analysis Assistants	122
7	Conclusion	124
	Bibliography	128

List of Figures

2.1	A DataFrame input-output example.	10
2.2	A generator of all valid arguments to the <code>pivot</code> function from the <code>pandas</code> API. <code>Select(D, c, i)</code> returns a single element from the domain <code>D</code> , according to the semantics in Figure 2.4.	11
2.3	A procedure to find the arguments to the <code>pandas</code> function <code>pivot</code> that turn <code>inp_df</code> into <code>out_df</code>	12
2.4	Operator Semantics for Generators. σ and δ are initialized to empty maps before the first invocation of the generator. t is set to the integer zero before every invocation of the generator. Op_{End} is a special operator that is implicitly called at the end of each invocation of the generator. A detailed explanation is provided in Section 2.2.1	18
2.5	Generator-Based Enumerative Synthesis Engine	19
2.6	A Simplified Program Candidate Generator for <code>pandas</code> Programs.	20
2.7	Graph encoding of the query passed to the <code>Select</code> call at Line 4 in Figure 2.2, on the I/O example from Figure 2.1.	23
2.8	Operator-specific neural network architectures.	25
2.9	Smart Model Accuracies on Function Prediction Task, compared to a Random Baseline. Per-sequence Top- k accuracies provided. Color gives accuracy; darker is better. The color point (x, y) gives the top- x accuracy for sequence with ID y . Sequence IDs are sorted based on top-1 accuracy of the smart model.	32
2.10	Per-operator Top- k accuracies. Color gives accuracy; darker is better. The color point (x, y) gives the top- x accuracy for operator with ID y . Operator IDs are sorted based on the top-1 accuracy of the smart model.	33
3.1	Input-output examples alone discards user intent information that was present while creating the output. In this example, it is not immediately clear that 102.5 is the mean of 50, 70, 100, and 190.	36
3.2	An input (i), partial output (o) example, as well as a graph abstraction of user intent (G_{user}).	37
3.3	A user interaction with the UI that builds the graph abstraction of user intent from Figure 3.2.	38
3.4	The solution program for the synthesis problem in Figure 3.2, its intermediate and final output, and its graph abstraction.	39

3.5	Walkthrough of GAUSS run on the specification in Figure 3.2, with components <code>gather</code> and <code>group_by</code>	40
3.5	Walkthrough of GAUSS run on the specification in Figure 3.2, with components <code>gather</code> and <code>group_by</code>	41
3.6	Table abstraction for the input in Figure 3.2.	46
3.7	Component abstraction of a call to <code>group_by</code> . The constant arguments are embedded inside the call.	46
3.8	Trace Abstraction for Program in Figure 3.4. The nodes and edges in the blue and orange boxes correspond to graphs G_1 and G_2 respectively.	47
3.9	Constructing decompositions with respect to skeleton ($\nu_1 = \text{gather}(t_1, \vec{\square}_1); \nu_2 = \text{group_by}(\nu_1, \vec{\square}_2)$)	53
3.10	Walkthrough of how the UI creates a graph spec. capturing intent as the user constructs the output.	62
3.11	Comparison to VISER, MORPHEUS and NEO. Red dots in (b) indicate timeouts. In (b) and (c), dots above the black line indicate that GAUSS is better, and dots above the teal dotted line indicate that GAUSS is 10x better.	65
3.12	Maximal reduction in the number of output nodes such that GAUSS still synthesizes the correct program. Dots above the green line and grey line indicate that reduction is more than 10x and 100x, respectively.	67
4.1	VIZSMITH’s Jupyter notebook frontend. VIZSMITH is provided with a table as a Pandas dataframe along with columns to visualize as input. It has a search bar to input text queries. (A) shows how Alice uses VIZSMITH to search for normalized stacked bar charts for her call quality dataset. (B) and (C) show the visualization selected by Alice and its code respectively.	72
4.2	Overview of VIZSMITH.	74
4.3	Example of a visualization and a corresponding slice extracted from Kaggle.	76
4.4	Minimized version of visualization slice in Figure 4.3.	77
4.5	Dependencies between top-level statements for code in Figure 4.3. Edges labeled 1, 3, 4, 5 and 6 capture dependency between the use and definition of a variable (<code>df_train</code> , <code>df_train</code> , <code>ax</code> , <code>sns</code> and <code>pd</code> respectively) while 2 captures the dependency between attribute reads and writes of an object (the dataframe in <code>df_train</code>).	79
4.6	Visualization functions extracted from slice in Figure 4.3.	79
4.7	A visualization function taking no arguments.	81
4.8	A visualization function with hard-coded values.	82
4.9	A visualization function using a specialized predicate.	83
4.10	Examples of each category in Table 4.4.	90
5.1	Overview of DATANA architecture	100

5.2	Inference rules, applied till fixed-point, for mining expressions from a computational notebook. \mathcal{E}_{all} is the set of all Python expressions in the notebook, \mathcal{E}_{pandas} is the set of expressions with types related to pandas, and \mathcal{E}_{mined} is the final set of expressions mined.	101
5.3	Prompt supplied to Codex for generating descriptions of code snippets. The gray, italicized text at the end corresponds to the Codex-generated description.	103
5.4	Prompt supplied to Codex for generating code from natural language descriptions. The gray, italicized text at the end corresponds to the Codex-generated code.	104
5.5	Examples of helper descriptions for snippets, italicized with a yellow highlight.	105
5.6	Prompt supplied to Codex for generating code from possibly imprecise or incomplete descriptions as part of the bidirectional consistency check. Additional code context is provided to fill the information gap in the description. The gray, italicized text towards the end corresponds to the Codex-generated code.	106
5.7	Prompt supplied to Codex for generating parameterizations of code snippets and their bidirectionally consistent descriptions.	108
5.8	Prompt supplied to Codex for code generation in the presence of retrieved examples. The gray, italicized text towards the end corresponds to the Codex-generated code.	109

List of Tables

2.1	List of Available Operators.	17
2.2	Performance on Real-World Benchmarks. Dashes (-) indicate timeouts by the technique. AP represents AUTOPANDAS and BL stands for BASELINE	30
4.1	Column Mutation Operators	85
4.2	Competition Statistics. # notebooks is the number of notebooks eligible for execution. ✓, ∅, ⊥, × indicate that at least one viz was mined, no visualizations mined, timeout and error respectively. # viz. funcs is the number of visualization functions mined with reusable count in brackets.	88
4.3	Top-10 API functions in each category, and the number of reusable viz. functions that use the API.	88
4.4	Characterization of misclassifications by our metamorphic testing approach. FP and FN stand for false positive and false negative respectively	89
5.1	Performance (% Accuracy) on JIGSAW benchmarks. Results marked (*) are reported directly from JIGSAW evaluation results [52].	111
5.2	Performance (% accuracy) of DATANA on JIGSAW benchmarks for different top-k.	114
5.3	Ablation Study for JIGSAW benchmarks. DATANA is compared against using vanilla CodeBERT as the search engine (Vanilla CodeBERT) and without using additional descriptions in the search corpus (No-Add-Desc)	115

Acknowledgments

They say, and rightfully so, that a Ph.D. is a formative period in the life of people who pursue one, and I am not an exception. As I wrote this dissertation, I got a chance to revisit many things from back when I started graduate school. It was surreal to be able to fully acknowledge my growth not only as a researcher but also as a person. I have discovered qualities in myself, both scientific and spiritual, that I am sincerely proud of. Not an iota of this growth was achieved alone. I feel extremely fortunate to have been in the company of some wonderful people who have helped shape me in countless ways, and I'm grateful for this opportunity to convey my deepest thanks to them.

First, I must thank my advisor Koushik Sen for his incredible mentorship over the last many years. I joined Berkeley as a fledgling student attracted to all things shiny and complex, and soon I found myself distressed watching all my seemingly brilliant ideas fail equally spectacularly. Koushik patiently taught me the power of simple yet effective ideas, the art of failing fast, and how to look beyond the fancy greek letters and at the bigger picture. He gave me the freedom to pick problems I was excited about, the space to fail, and the encouragement to learn from my mistakes and have another go. As I leave graduate school, I feel confident in my technical and communication abilities as an independent researcher, and I owe this, in large part, to Koushik. I do hope to follow in his footsteps and impart the same lessons in my future mentorship roles.

My first and foremost advice to any junior student has always been to seek mentors, stemming from my own fortunate experience of having a highly-supportive group of faculty and industry mentors to rely on for guidance throughout my Ph.D. I have worked extensively with Mukul Prasad ever since I interned at Fujitsu in my first summer here, and I cannot thank him enough for his role in my professional development. I learned a lot from him about how to critically approach research, which has significantly improved my technical communication skills. Ion Stoica has always championed my projects and was instrumental in the success of my first project at Berkeley. Joe Hellerstein's kind encouragement, critical feedback, and deep insights around empowering data analysts were pivotal in shaping my research and putting my work in perspective in this dissertation. While my internship work at Microsoft did not fully relate to my thesis, the mentorship from Vu Le, Ashish Tiwari, Sumit Gulwani, and the rest of the PROSE team has been nothing short of transformational. I am especially grateful to Sumit for his confidence in me and for affording me great exposure within the company even though I was just an intern. Finally, I must thank my undergraduate research advisor, Subhajit Roy, for introducing me to research in the first place. I would not be here if it were not for his mentorship, and for that, I am deeply thankful.

I am indebted to my closest collaborator and dear friend, Caroline Lemieux. We developed the AUTOPANDAS and GAUSS projects together, and as the senior student on the team, she played a major role in guiding the research in the right direction. She also helped me communicate ideas and results more effectively in our weekly research meetings with faculty, which greatly helped in reducing stress in my junior years. The Covid-19 pandemic coincided with the formative years of my Ph.D., and I am grateful for her constant support,

guidance, and friendship through those difficult times. As she embarks on a new journey as a professor herself, I have no doubt her students will similarly benefit greatly from their own interactions with her.

Roy Fox overlapped with me at Berkeley while I was developing AUTOPANDAS, and I am thankful for his insights and profound machine learning knowledge that was crucial in getting the project going. I would also like to extend my gratitude to Hiroaki Yoshida, who was my mentor when I interned at Fujitsu. He helped me complete and publish my first research project as a graduate student at Berkeley (this was even before AUTOPANDAS). The first paper is always so special, and I am thankful for Hiroaki’s help in getting there.

I have learned a lot from my junior collaborators as well. I had the good fortune of collaborating with Shadaj Laddad on VIZSMITH, and I am deeply appreciative of his creative solutions that were instrumental in making the project a success. I am always left in awe of his technical prowess, and I, for one, cannot wait to see what he builds next. It was a privilege and honor to be able to mentor two extremely talented undergraduate students: Samy Cherfaoui and Arushi Somani. I am constantly amazed by their ingenuity in solving difficult problems and juggling Berkeley’s difficult coursework with research. They did, however, make me feel older than I actually am, so that’s the only thing I won’t cherish across all my interactions with them. Or maybe I will, who knows.

The arduous walk up from North Berkeley to Soda Hall was worth it thanks to a wonderful group of colleagues: Benjamin Brock, Kevin Lauefer, Aayan Kumar, Rohan Padhye, Ed Younis, Sahil Bhatia, Justin Lubin, Parker Zielger, Azad Salam, Shishir Patil, and many, many others. Even though the pandemic took away a lot of our time together, I leave with fond memories of our lunches and group meetings.

I also feel lucky for the constant support from my friends from the time I was an undergraduate: Saransh Srivastava, Pranav Vaish, Sundararajan Renganathan, Sanchit Mall, Abhimanyu Goyal, Anand Singh Kunwar, Aadi Yogakar, and Skand Upmanyu. While all of us chose different career paths, I am thankful that we stayed in touch.

I must say I had what one might call a cheat code through graduate school. It was the unwavering support, from the opposite side of the world, of my partner Shivanee Ghadge. While no amount of writing will do justice to the impact she has had on my success in graduate school, I might as well still try. Besides knowing how to make me laugh at any point, her powers also include tremendous empathy and maturity that helped me navigate the last few years. In spirit, at least, we have both earned this degree.

Finally, my family has been my biggest cheerleader throughout my graduate school journey. My parents have always been there to supply that extra ounce of strength to get me over a hurdle and celebrate my achievements louder than I ever have or ever will. I also want to thank my elder brother Karan Bavishi for his sagely advice from time to time and also for his help in writing VIZSMITH and proofreading this dissertation!

As we prepare to start talking about programming assistants, I will close with a quote from Tennyson that I had taped to my desk and that I turned to for motivation throughout:

“To strive, to seek, to find, and not to yield.”

Chapter 1

Introduction

Data is the new oil.

— Clive Humby, 2006

This now-famous quote succinctly captures how data has changed the world. But as Palmer [80] and others noted, this is not the complete picture — just as oil needs refinement and processing to be harnessed for profit, data needs to be *analyzed* to extract value.

Most data analyses involve data preparation, exploration, and visualization tasks [55]. Examples include parsing fields from unstructured data, restructuring, and reshaping, identifying and handling outliers, data imputation, and visual exploration to gather insights. It is widely accepted that these tasks take up the majority of the time spent in data analysis pipelines. Around 2012, the fraction of time devoted to such tasks was estimated to be greater than half [55], and even as high as up to 80% [85, 70]. Heer et al. attribute this phenomenon to the inherent domain-specific nature of these tasks, which necessitate custom transformations for every new source of data [47]. Specifying such custom analysis steps in low-level editing tools such as the widely-used Excel [77] spreadsheet software can often prove to be prohibitively burdensome. This *specification burden* has historically forced domain experts who do not have sophisticated technical or programming backgrounds to have to rely on IT teams for performing complex analyses, greatly decreasing efficiency at the organizational level [48].

Significant research has been devoted to empowering this class of non-programmer analysts by building smarter interactive and graphical tools to reduce the specification burden. This includes novel visual interfaces for data processing [142, 79, 93] and visualizations [111], program synthesis and inference technology to translate ambiguous clicks, examples, and direct data manipulations into data processing steps [56, 42, 47] as well as proactive suggestions to recommend next steps or visualizations [44, 71]. A number of these techniques have been successfully commercialized: Trifacta [6] extends the predictive interaction framework [47], while Tableau [115] emerged from Polaris [111]. Overall, these tools enable *self-service* [48]: end-to-end data processing and visualization without the need for programming. The market size for self-service analytics was estimated to be \$3 billion in 2020 and is projected to reach \$13 billion by 2028 [99].

Given the successful adoption of interactive and graphical tools by analysts with non-programming backgrounds, one might reasonably expect their adoption from the programmer class of analysts as well. However, recent studies [130, 5, 31] indicate that a sizeable proportion of programmer-analysts eschew these tools in favor of working in their preferred coding environments such as computational notebooks [105, 61]. A primary reason is the desire to avoid the prohibitive cost of switching between coding and multiple interactive tools [130, 5]. Another reason is the need for even greater flexibility and customization [31].

The specification burden still remains, however. Inefficiency due to specification burden is not exclusive to non-programmers. The complexity of modern languages and data analysis libraries [118, 50, 86] imposes a barrier to productivity for programmer analysts when specifying analyses using code. Analysts cite *recall* as a major issue [130, 31, 20] as they report spending significant time in revisiting documentation and searching Stackoverflow to recollect the syntax and names of functions in these libraries [31].

Thus, any approach towards reducing the specification burden for programmer analysts must *meet them where they work* [133]. In other words, there is a need for *programming assistants* to help analysts write code more efficiently, and in the environments they prefer. Several approaches aim to bring the benefits of direct manipulation and interactivity directly to computational notebooks [133, 31, 65]. The main challenges include maintaining fluid interoperability between code and interaction [58, 133], and translating scattered interaction steps or intermediate representations into compact, idiomatic code [58, 31]. Consequently, most tools in this space target a subset of tasks such as visual exploration [65] and interactive filtering and selection [133] in order to maximize the benefits of interaction while still remaining practical to implement. An alternate class of approaches, which includes the techniques contributed by this dissertation, tries to be more general by focusing on building programming assistants that *automatically generate code* from high-level specifications such as input-output examples and natural language.

Automatic code generation from such high-level specifications is made difficult by the conflicting goals of expressivity and performance and/or accuracy. Program synthesis techniques for code generation from input-output examples [42, 92, 34, 125] and semantic parsing approaches for natural language interfaces [103, 68, 138] have largely focused on supporting a useful but limited set of tasks, sacrificing expressivity to reap performance benefits. In contrast, recent advances in machine learning, particularly natural language processing, have shown promise in realizing full expressivity through approaches for free-form code generation from arbitrary natural language [135, 82, 21, 10]. However, their sub-optimal accuracy remains a concern. The key question underpinning this dissertation thus emerges naturally:

Can we break the expressivity and performance/accuracy tradeoff barrier to build practical programming assistants that automatically generate code from high-level specifications?

This dissertation builds upon prior work and introduces novel techniques that combine insights from synthesis, automated testing, program analysis, and machine learning to make significant progress toward resolving this question. Specifically, we contribute four core tech-

niques and corresponding assistants, namely AUTO-PANDAS, GAUSS, VIZSMITH, and DATANA, that generate data processing and visualization code from input-output examples, demonstrations, or natural language specifications. AUTO-PANDAS and GAUSS constitute core advances in search space and algorithm design for example-based synthesis. VIZSMITH and DATANA introduce novel mining and auto-summarization techniques to automatically build aligned code and natural language corpora, which DATANA uses to greatly improve the code-generation capabilities of modern large language models such as Codex [21]. Compared to prior work, these assistants improve the expressivity of synthesis-based systems and the accuracy of machine-learning-based systems. Chapters 2-5 formalize the techniques behind these tools. Chapter 6 discusses related work on assistants. Finally, Chapter 7 concludes with a summary of the key contributions of this dissertation and outlines opportunities for future work. The remainder of this chapter elaborates on the main arguments presented thus far, as well as the design of the four tools contributed by this dissertation.

1.1 Reducing the Specification Burden via Interaction

Heer et al. attribute the difficulty faced by analysts in completing an end-to-end data processing and exploration analysis to the domain-specific nature of data [47]. Different data sources need different treatment, and the resulting analysis is made up of steps that are highly custom in nature. No analysis tool can provide core actions or abstractions catering to each of the steps. This, in turn, imposes a *specification burden*: the analyst needs to break down or *decompose* the task further into lower-level steps that are expressible in their analysis tool of choice, if at all possible. Performing such decomposition inherently requires familiarity with computational thinking, rendering such analyses out of reach for domain experts and analysts who do not possess a programming background. As an example, consider a data cleaning task: filling missing values in a column with the last valid value in the same column. This task is common enough to warrant a dedicated function alias `ffill` in the pandas [118] data-processing library in Python. However, doing this in the widely-used graphical spreadsheet software Excel [77] would require the use of a fairly complex sequence of UI interactions [89]. Heer et al. recognize the importance of reducing this specification burden to make complex and custom analyses accessible to a larger workforce.

A large body of literature is dedicated to building better interactive and visual tools to reduce the specification burden. Broadly speaking, these works employ one or more of four strategies. First, developing novel visual interfaces that map directly to a subset of a query or transformation language [142, 79, 93] or specially-designed domain-specific languages [111, 45]. Second, using program synthesis to convert ambiguous interactions or input-output examples [56, 42, 139, 47] to programs in a transformation DSL using program synthesis. Third, automatically generating preparation or visualization recommendations based on interesting properties of data [93, 71, 88]. Fourth, mixed-initiative approaches that use a unified medium for recommendations and feedback to enable tight user interaction loops [44, 132, 131]. A number of commercial tools employ these techniques: Trifacta builds upon

the predictive interaction framework [47], which among other features, unifies Wrangler’s [56] *reactive* system for generating contextual suggestions from user interactions and the *proactive wrangling* framework [44] for generating suggestions to help users discover next steps. Tableau (née Polaris [111]) offers a rich and easy-to-use graphical interface for creating visualizations, powered by a specially designed visual query language VizQL [45]. Users drag and drop data attributes or “pills” onto “shelves” that correspond to visual channels such as the chart axes (columns and rows), mark type, and color, among others. Overall, this class of tools enables *self-service* [48]: it empowers domain experts who do not have a programming background to efficiently perform end-to-end data preparation. The self-service analytics industry has grown rapidly in recent years. Acumen [99] estimated the market size to be around \$3 billion as of 2020, with the value projected to rise up to \$13 billion by 2028.

1.2 The Need for Programming Assistants

Given the success of interactive tools and the efficiency resulting from their deliberate design, one might expect widespread adoption of these tools by programmer analysts to ease their workflow as well. However, recent interview studies of professional analysts [130, 5, 31] suggest that a sizeable proportion of programmer-analysts eschew these tools in favor of working in their preferred coding environments such as computational notebooks [105, 61] along with the data analysis frameworks and libraries of their choice such as pandas [75, 118], matplotlib [50], and scikit-learn [86] among others. A major reason cited in these studies was the prohibitive cost of switching between multiple interactive tools as their sets of capabilities are disjoint [130, 5]. In contrast, a coding environment provides the convenience of being able to perform any desired task in a single place. Limited customization and the “black-box” feel [31] of interactive tools was also a common line of reasoning. Some analysts also expressed the need for “homegrown automation” [5]; they often need to build their own automation scripts for custom yet repetitive and mundane workflows. Interactive tools, however, only have limited support for such automation via macros or templates. Lastly, the ability to share, reproduce [94], and collaborate on [87] code-based workflows was also seen as a compelling reason to stick to programming-based tools.

While analysts with a programming background are not as constrained as those without the same level of technical ability, they are not immune to inefficiencies that arise due to the *specification burden*. After all, programming is an inherently difficult exercise, and writing data analysis code is no different. Novices face a steep learning curve when it comes to learning about the abstractions offered by programming-based analysis libraries and frameworks and how to combine them effectively. Even experts cite *recall* as a major barrier to productivity [130, 31, 20]; having to constantly revisit documentation and Stackoverflow to check the syntax and names of functions in libraries like pandas and matplotlib is time-consuming [31]. Thus, efficient specification of a task using these powerful and expressive data analysis libraries remains a challenge for novices and experts alike.

Clearly, there is a need to reduce the specification burden for programmer analysts, but

it is equally important to *meet them where they work* [133]. In other words, there is a need for *programming assistants* that ease the process of writing analysis code. Several approaches to this problem employ the strategy of bringing the benefits of direct manipulation and interactivity directly to computational notebooks [133, 31, 65]. Analysts, regardless of programming ability, have also explicitly expressed a desire for such tools that combine code and interaction [130, 5], acknowledging the strengths of the two mediums. However, it is not simply a matter of integrating a web-based UI into a computational notebook environment. It is important to reduce the friction involved in switching between the two modes [58, 133]. Reconciling the reproducible nature of code with the transient nature of interactions [133] introduces a further challenge of translating scattered interaction steps into compact idiomatic code [58, 31]. The use of domain-specific languages (DSLs) as an intermediate language to represent a transformation also introduces the problem of translating it to human-readable code [31]. Consequently, tools in this space tend to target a subset of tasks such as visual exploration [65] and interactive filtering and selection [133] in order to maximize the benefits of interaction while still remaining practical to implement. Complex and compositional reshaping, aggregation, or visualization tasks still need to be expressed via code in these systems. An alternate class of approaches, including the ones contributed by this dissertation, focuses on building tools that accept high-level specifications such as input-output examples and natural language commands and automatically generate suitable code.

1.3 Choice of High-Level Specifications

Input-output examples and natural language descriptions are just two of many possible choices for specifications, which include logical specifications [7], sketches [109], direct manipulation and interaction [47, 111], demonstrations or traces [18, 19], and other variations. There is little doubt regarding the *viability* of the use of input-output examples and natural language as specifications. After all, their use is prevalent across software engineering as a means to communicate unknown functionality through formal documentation. Additionally, much of the programming help solicited on StackOverflow heavily utilizes descriptions and examples as a means of communicating intent.

A more important question than *viability*, however, is *utility*: are input-output examples and/or natural language sufficiently useful for specifying a wide variety of data preparation and visualization tasks? That is, do they truly reduce the *specification burden* for these tasks? This question is especially important in cognizance of the rich history of innovations in direct manipulation and interaction technology over the past many decades. At the very least, this history strongly suggests that input-output examples and natural language are *not* the optimal choices for a reasonable subset of core analyses.

User studies in prior work have validated input-output examples as a useful mode of specification for string processing [42], table transformation and reshaping [34], and visualization tasks requiring data transformation prior to mapping attributes to visual channels [124]. In general, participants found input-output examples most useful for solving *complex tasks*;

users found it difficult to manually break down these complex tasks into simpler chunks that map straightforwardly to functions in a library. The utility and convenience of natural language as a means of querying databases and making visualizations is also evidenced by a large body of academic work [103, 41, 67, 68, 138], as well as features in commercial systems such as PowerBI [39] and Tableau [115]. The latter example is especially notable as Tableau’s primary product is powered by direct manipulation.

As Alspaugh et al. observed, analysts generally prefer the convenience of direct manipulation for exploratory tasks where they wish to evaluate multiple options quickly and at scale [5]. Direct manipulation is also more convenient for tasks such as formatting plots or browsing data. This makes the goal and motivation of this dissertation *complementary* to parallel efforts on bridging the code and interaction mediums to bring the benefits of direct manipulation to computational notebook environments, such as the B2 [133], Lux [65], and WREX [31] systems. More concretely, analysts can utilize these direct manipulation tools for tasks they are best suited for while switching to examples or natural language specifications for the more complex tasks.

1.4 Code Generation from High-Level Specifications

Once a particular form of specification has been fixed, such as input-output examples or natural language descriptions, automatic code generation techniques can be studied from the lens of its two underlying dimensions: the search space and the search algorithm [43]. The search space represents the space of possible programs that can be generated by an assistant and thus controls *expressivity*. On the other hand, the search algorithm, which may be systematic (enumerative search, deduction) or statistical (machine learning (ML) models), controls the performance and accuracy of the assistant. These goals of expressivity and performance and/or accuracy are inherently conflicting; increasing expressivity by enlarging the search space will, in turn, make systematic search harder to scale or possibly affect the accuracy of these assistants in the case of ML models. Program synthesis techniques for code generation from input-output examples [42, 63, 92, 126, 123, 8, 34, 125, 36, 128, 23], as well as semantic parsing approaches for natural language interfaces [68, 138] have largely focused on supporting a useful but limited set of tasks, thus sacrificing expressivity to reap performance benefits. In contrast, recent advances in machine learning, particularly natural language processing (NLP) with the advent of large language models [16], have shown promise in realizing full, general expressivity through approaches for free-form code generation from arbitrary natural language [135, 82, 21, 10]. However, their sub-optimal accuracy remains a concern. In this dissertation, we push the boundaries of this tradeoff barrier between expressivity and performance/accuracy for assistants that generate code from examples or natural language. We introduce techniques and corresponding assistants that combine insights from program synthesis, automated testing, program analysis, and machine learning to support a richer range of data analyses as compared to prior work while still being practically fast and accurate.

AUTOPANDAS (Chapter 2) contributes *neural-backed generators*, a novel combination of QuickCheck-style generators [26] and machine learning, to enable generation of dataframe transformation code from input-output examples. Generators are helpful in capturing a large complex space of test inputs (candidate programs in our case) through the use of well-placed *choice points* interconnected by program logic in any general-purpose language. Choice points help capture the inherent parametric nature of pandas, and the complex constraint validation system of pandas can be captured in a straightforward manner using (Python) code. But generators are only one piece of the puzzle — they can be invoked multiple times to produce different candidates, but for code-generation from examples, we need the generator to return candidates *more likely* to satisfy the example first. AUTOPANDAS *backs* the choice points with graph neural network models, which it trains to iterate over more likely choices at each choice point first. Compared to prior work, AUTOPANDAS supports a much more comprehensive subset of pandas functionality thanks to the use of these generators. Despite this larger search space, AUTOPANDAS shows promising performance on reshaping benchmarks, solving 50% of benchmarks in under 30 seconds. To tackle the subset of tasks involving aggregation or computation, where AUTOPANDAS underperforms due to the lack of precise information in traditional input-output examples, GAUSS (Chapter 3) introduces an element of interaction to input-output examples. Specifically, GAUSS captures the precise relationships between the cells and columns of the input and output dataframe as a graph. Consequently, this graph contains a lot more information than plain input-output examples. Exploiting this information enables GAUSS to significantly outperforms other state-of-the-art approaches targeting the same subset of tasks but only using plain input-output examples.

VIZSMITH (Chapter 4) and DATANA (Chapter 5) correspond to a shift in focus to code generation from natural language specifications — an inherently harder problem as candidate programs cannot be automatically evaluated for correctness, making it challenging from an accuracy perspective as opposed to performance. To balance expressivity and accuracy, given a natural language query requesting a visualization, VIZSMITH uses keyword-based retrieval on a corpus of *aligned* code and its natural language descriptions to generate relevant visualizations. The novelty lies in the design of this corpus: it is fully automatically constructed, using program analysis to mine code and associated comments from publicly available notebooks on the data science platform Kaggle [116]. DATANA expands VIZSMITH’s scope in two ways. First, it eliminates the reliance on human-written code annotations, which are often imprecise or even irrelevant in notebook workflows. It uses large language models such as Codex [21] to automatically generate high-quality descriptions for code and, consequently, higher-quality aligned code and natural language corpora. Second, it augments the code-generation approach of Codex with automatically retrieved examples from such corpora, leading to better performance as well as support for a larger class of queries than VIZSMITH. The next four chapters describe each of the tools in detail.

Chapter 2

AutoPandas: Tackling Large Search Spaces via Neural-Backed Generators

This chapter introduces the techniques behind AUTO-PANDAS, our first in the series of programming assistants developed in this dissertation that automatically generates dataframe transformation code that uses the `pandas` library from input-output examples. We focus on `pandas` as it is a prominent and widely-used library for data manipulation in Python. As mentioned in the introduction (Chapter 1), the use of input-output examples is motivated by the observation that `pandas` novices often include such examples when asking a question on StackOverflow. Moreover, prior studies have independently validated their usefulness for the domain of dataframe or table transformations [34].

The underlying code generation problem in AUTO-PANDAS belongs to a class of synthesis problems called programming-by-example [43], where the goal is to generate a program that adheres to a given input-output example. This means that the code generated by AUTO-PANDAS is always *guaranteed* to satisfy the example. That is, if a returned solution is run on the input, its output will match the target output. Programming-by-example systems have been integrated as successful features in widely-used commercial systems. Trifacta [6] solves a programming-by-example problem as part of its automatic text parsing and splitting functionalities. FlashFill [42] is another success story within the Excel ecosystem, automating string processing using input-output examples. A common design decision in these commercial features, as well as a large body of academic work on programming-by-example systems [8, 92, 123, 127, 128, 34, 36] is the use of a small and specialized domain-specific language or grammar to capture the space of possible programs. This, in turn, allows the underlying search algorithms to scale to real-world tasks. Typically, these languages only contain tens of operators or functions. In contrast, `pandas` has hundreds of functions just operating on dataframes, rendering a similar treatment infeasible without compromising on *expressivity*. The central goal of this dissertation is to develop techniques that do not deliberately restrict the expressivity of the corresponding assistants, and we thus deviate sharply from this class of approaches.

Beyond the sheer number of functions in `pandas`, finding the correct arguments for a

given function is a challenge. Many API functions in `pandas` accept multiple types of values for a single argument and have non-trivial inter-argument constraints that must be satisfied to constitute a valid invocation. Intuitively, any code generation system not possessing this knowledge in some form, either implicit or explicit, may face issues forming a correct program altogether, let alone one that satisfies an input-output example.

Thus, innovation is needed along the remaining dimensions of synthesis: design of the search space and, consequently, the algorithm. We first focus on the former; based on the need to precisely represent the complex space of programs allowed by `pandas`, we propose a generator-based approach to the programming-by-example problem. Fundamentally, the generator is a Python function that yields a different well-formed `pandas` program each time it is called. This generator encodes expert domain knowledge about the `pandas` API to enable the generation of *valid*, executable `pandas` code. That is, when producing arguments to a function, the generator should almost never produce argument combinations that cause the function to immediately error out. We strongly believe that writing such a candidate program generator is relatively straightforward for someone with knowledge of the `pandas` API. Since the generator is written in Python, it can utilize `pandas` code itself to express various constraints. Given such a precise program generator, the simplest way to search for a solution given an example is to repeatedly call the generator until it produces a program that satisfies the input-output example.

However, such an approach is, unsurprisingly, inefficient. For such an approach to be quicker, the generator must return *likely* or more promising programs first. Since generators are simply arbitrary functions, one could work with a program synthesis expert to build heuristics that prioritize more program candidates that are more likely to pass the test. Building such heuristics is extremely *tedious*, *error-prone*, *non-scalable*, and requires a long process of trial-and-error to extract performance.

The *key insight* behind AUTO-PANDAS is to leverage the structure of the program candidate generator. A generator contains multiple *choice points* connected by arbitrary program logic. For `pandas` programs, the domain of options at these choice points would directly correspond to the heuristics one would wish to develop for synthesis. For example, there would be exactly one choice point corresponding to the selection of a column argument of the `pivot` function. Instead of requiring the developer of the generator to write sophisticated heuristics to pick this argument, we provide a *smart operator* which uses a *learned probabilistic model* to make the most likely choice over the domain given the input-output example. Thus, the generator need only capture domain-specific constraints; the fuzzier decisions are left to smart operators.

Another advantage of using localized smart operators is that the probabilistic models that back them thus only need to be trained for a specific selection task (i.e., given this context, which element should be selected from the set). This is in contrast to the use of probabilistic models in past work such as that of Lee et al. [66] and Devlin et al. [29] where these models are trained over the full language, an inherently harder task.

AUTO-PANDAS supports 119 `pandas` operations on dataframes out of 136 that were available at the time of development. We deliberately ignored functions whose participation in

	Date	Category	Location	Expense	Balance
0	2018-02-18	Social	Terrace	98.34	9971.66
1	2018-02-18	Lunch	Pox	245.63	9726.03
2	2018-02-24	Social	Gate 320	121.89	9604.14
3	2018-02-24	Lunch	Pox	248	9356.04

(a) An example input dataframe.

	Lunch	Social
2018-02-18	245.63	98.34
2018-02-24	248	121.89

(b) Desired output.

Figure 2.1: A DataFrame input-output example.

a program may render the task unsuitable for specification via input-output examples. An example of such a function is `sample` where the presence of randomness makes it infeasible to provide a fixed target output to adhere to. We hand-wrote a program candidate generator that supports all these functions. This generator encodes, in native Python, the pre-conditions for each supported `pandas` API.

On a collection of 26 real-world benchmarks, we find that `AUTOPANDAS` can efficiently solve 13 benchmarks requiring less than 30s per task, and 17 benchmarks if we relax the time requirement to 20 minutes. While we could not compare with prior work due to less functionality supported, we note that these performance numbers are below those generally reported in the aforementioned systems that enjoy a smaller search space to work with. The complexity of `AUTOPANDAS`'s space places a significant burden on the model part, which does struggle with tasks requiring longer composition of operations or aggregation and computation operations. The reason for the latter tasks stems from the design of the models (Section 2.2.3.2). We also analyze the accuracy of our smart operators, which we find to be substantially higher (Section 2.3.4) for the task of argument prediction as compared to using deterministic or randomized semantics.

Before we dive into the technical details behind `AUTOPANDAS`, we walk through a small example illustrating the need for example-based assistance for `AUTOPANDAS` along with a deeper overview of `AUTOPANDAS`'s internals.

2.1 Overview

Consider the following scenario. Suppose an analyst needs to use `pandas` to process some expense data, loaded up into a `pandas` dataframe as shown in Figure 2.1a into the form. Specifically, they need to transform it to the dataframe shown in Figure 2.1b. Being a novice when it comes to using `pandas`, they need to pick out the right function or combination of functions from the large API offered by `pandas`.

However, the analyst is quite familiar with Excel and knows that they could use the “pivot_table” functionality to perform the transformation in Excel. Luckily enough, `pandas` also contains a function with the same name in its API, and the analyst is confident about being able to leverage it for their task.

```

1 @generator
2 def gen_pivot_args(inp_df: pandas.DataFrame, out_df: pandas.DataFrame):
3     context = (inp_df, out_df)
4     arg_col = Select(inp_df.columns, context, id=1)
5     arg_idx = Select({None} | inp_df.columns - {arg_col}, context, id=2)
6     if isinstance(inp_df.index, pandas.MultiIndex) and arg_idx is None:
7         arg_val = None
8     else:
9         arg_val = Select(inp_df.columns - {arg_col, arg_idx}, context, id=3)
10
11     return {'columns': arg_col, 'index': arg_idx, 'values': arg_val}

```

Figure 2.2: A generator of all valid arguments to the `pivot` function from the `pandas` API. `Select(D, c, i)` returns a single element from the domain `D`, according to the semantics in Figure 2.4.

But, even after knowing which function in `pandas` to use, the analyst is faced with the challenge of understanding what all the arguments to `pivot_table` mean. For example, one can supply the `agg` keyword argument to add an aggregation, which defaults to taking the average of values. But there is no such aggregation happening in Figure 2.1. After much struggle and reading up posts on StackOverflow, they finally realize that `pandas` also offers a much simpler function named `pivot`, which can be used to achieve the same task.

The primary motivation behind an assistant like AUTOPANDAS is to help analysts out in precisely these situations where the complexity of an API such as AUTOPANDAS can significantly affect productivity. This is also not limited to novices, one can reasonably imagine a seasoned `pandas` user also struggling to *recall* the semantic differences between `pivot` and `pivot_table` or the meanings of the various arguments they accept.

AUTOPANDAS is a programming-by-example system: given a dataframe transformation represented by an `(input, output)` example like the one in Figure 2.1, AUTOPANDAS outputs a program p which performs the transformation. That is, $p(\text{input}) = \text{output}$. A central tenet of our approach in AUTOPANDAS is the desire to be as expressive as possible. In particular, AUTOPANDAS aims to solve the above problem for a wide variety of `pandas` tasks. At the heart of this approach is the use of *generators*.

A generator is simply a function that enumerates all valid `pandas` programs for a given input. Let us revisit the example in Figure 2.1. Assuming for a moment that `pandas` only has a single `pivot` function, the generator then would correspond to the function `gen_pivot_args` in Figure 2.2. In particular, `gen_pivot_args(df)` returns a different *valid* argument combination every time it is invoked. Thus we can straightforwardly use `gen_pivot_arg(df)` to enumerate all possible argument combinations to `pivot` and return the correct one to the struggling analyst. Figure 2.3 shows the pseudo-code for this simple search. The code simply calls

```

1 def find_pivot_args(inp_df: pandas.DataFrame, out_df: pandas.DataFrame):
2     while True:
3         cur_kwargs = gen_pivot_args(inp_df, out_df)
4         cur_out = pandas.DataFrame.pivot(inp_df, **cur_kwargs)
5         if cur_out == out_df:
6             return cur_kwargs

```

Figure 2.3: A procedure to find the arguments to the `pandas` function `pivot` that turn `inp_df` into `out_df`.

`gen_pivot_args(df)` (Line 3) iteratively until it returns an argument combination `kwargs` such that calling `pivot` on the input in Figure 2.1a with these arguments would return the output equivalent to the one in Figure 2.1b.

How does `gen_pivot_args(df)` only return valid argument combinations? This stems from the rich constraints, which are embedded within the body of the function, that correspond exactly to `pandas`' own internal validation requirements:

1. `arg_col` should be selected from the list of column names of `df`, `df.columns`.
2. `arg_idx` is either `None`, or selected from the list of column names of `df`, except from the column name used in `arg_col` (`df.columns-{arg_col}`).
3. Finally, the `arg_val` argument should either be (1) selected from the list of column names except for the ones used in `arg_col` and `arg_idx`, or (2) `None`, in the case where `arg_idx` is `None` and `df` has a multi-level index.

These constraints are universal for the `pivot` function, and an expert (us, for the purpose of developing `AUTOPANDAS`) can straightforwardly derive them from the documentation. The actual selection of the arguments is delegated to the `Select` operators. Every time the generator is invoked, the operators jointly return a new combination of arguments. Essentially, these calls to `Select` allow `gen_pivot_args(df)` to cycle through different argument combinations across different invocations.

However, there is still a problem. If there are many argument combinations, the basic search in Figure 2.3 may take some time to terminate. If the order in which `gen_pivot_args` returns arguments is arbitrary, the correct argument combination is unlikely to show up early enough for the code in Figure 2.3 to be practical. Also, we are only considering the `pivot` function here; the problem is far worse when considering the full `pandas` API. The problem is exacerbated if sequences of multiple functions are required to perform the transformation, as the total number of possible argument combinations grows exponentially.

To make `gen_pivot_args` output the correct argument combination more quickly, the API expert could replace the calls to `Select(D,c,i)` with a particular enumeration order through `D`. The enumeration order would be based on some additional *heuristics*, for example:

1. The values in the column from `input` that is used as `arg_col` end up being column names in the `output`. Therefore, the generator should look at the output's column names and first try to use as `arg_col` any column from the input that shares values with the output's column names.
2. The values in the column from `input` that are used as the `arg_val` argument end up in the main data of the table. Hence, the generator should look at the output and first try to those columns as `arg_val` whose values are the same as the output's data cells. However, the values argument also accepts `None` as a valid argument, in which case all the remaining column values are used as the main data of the output. Therefore the generator should take this into account as well.
3. ... (*more heuristics omitted*)

Designing such heuristics is error-prone. They are not guaranteed to be effective, especially if the I/O example provided by the user cannot actually be solved with a single call to `pivot`. Further, it is much more tedious for the expert to write a generator that uses these heuristics than it is to write a generator that encodes the basic validity constraints, like that in Figure 2.2. Overall, using heuristics is an *error-prone, tedious, and non-scalable* way to more quickly find the correct argument combination.

We propose a different route. Instead of relying on humans to write more heuristics, we propose a smart backend for operators like `Select(D, c, i)`. This smart backend for `Select` first derives from the context `c` a probability distribution p over D . Then, it returns elements $d \in D$ in descending order of their probability $p(d)$. The distribution model can be represented by a neural network and learned from a training set of inputs, programs, and their outputs, as detailed in Section 2.2.4. Over a validation set of (`input`, `output`) pairs where `output = pivot(input, **kwargs)` for some arguments `kwargs`, our smart backend has 99% top-1 accuracy in retrieving the correct `kwargs`.

Further, instead of using the smart backends only to generate arguments for `pivot`, we use them to build a prototype synthesis engine for the `pandas` library, AUTOPANDAS. AUTOPANDAS takes in a pair of (`inputs`, `output`) representing a dataframe transformation and outputs a program `p` in the `pandas` API such that `p(input) == output`. We achieve this by (1) implementing a *program candidate generator* which outputs straight-line `pandas` programs that run without error on `input` and (2) using smart backends for `Select` and other operators so that the program candidate generator outputs `p` such that `p(input) == output` early in the search. AUTOPANDAS currently supports 119 `pandas` functions and can form programs with multiple function calls. Given the I/O example in Figure 2.1, AUTOPANDAS finds the correct program:

```
out_df = inp_df.pivot(index='Date', columns='Category', values='Expense')
```

after checking only *one* program candidate.

2.2 Technique

In the next section, we formalize (1) generators, and the semantics of **Select** and other operators, (2) generator-based synthesis, and (3) the smart backend we use to synthesize **pandas** programs.

2.2.1 Generators

We first formally describe *generators*. In our setting, a generator \mathcal{G} is a program that, when invoked, outputs values from a space of possible values. Figure 2.2 shows an example of such a generator \mathcal{G} for function arguments in the Python **pandas** library [118] for **DataFrame** (i.e. table) manipulation. In particular, the generator in Figure 2.2 takes a **pandas DataFrame** as an input, and returns one of the possible argument combinations of the **pandas** API function **pivot**, by selecting various argument values.

Our generators \mathcal{G} can contain arbitrary Python code, along with a set of stateful operators that govern the behavior of \mathcal{G} across runs. An example of such an operator is **Select**, which is also used in the generator in Figure 2.2. Given a collection of values, **Select** returns a single value from the collection. For example, the call to **Select** in Line 4 selects one of the columns of the input dataframe **df**. The generator then assigns this value to **arg_col**, to be used as the pivot column. Similarly, the call to **Select** in Line 5 picks either **None** or one of the columns in **df** *except* the one selected as **arg_col** (i.e., **df.columns** - {**arg_col**}), to be used as the index. Choosing **arg_val** is more complicated. In particular, if the input dataframe has a multi-level index, and **arg_idx** is **None**, **arg_val** must be **None** for **pivot** to not throw an error. Generators are a natural form in which to specify such contracts. The checks in place in Figure 2.2 ensure that the generator only generates arguments that follow the contract and thus, can be given to **pivot** without error.

On different invocations of the generator in Figure 2.2, the calls to **Select** may yield different values. There are a few different ways in which **Select** can do this. First, it can simply randomly choose values from **D**. Or, it can assure new values will be different by maintaining an internal state which records the values it returned in previous runs. Further, it can use its context argument **c** to determine the order in which it returns these values. We elaborate on this below.

Operators Apart from **Select**, we support three other operators, namely (1) **Subset**, (2) **OrderedSubset** and (3) **Sequence**. An informal description of their behavior is provided in Table 2.1, while a formal treatment is presented in Figure 2.4.

Each operator Op is of the form $Op(\mathcal{D}, \chi, id)$ where \mathcal{D} is the domain passed to the operator; χ is the context passed to the operator to control its behavior, and id is the unique static ID of the operator. The static ID of Op simply identifies each call to an operator uniquely based on its static program location. It is provided explicitly in Figure 2.2 for clarity but may be inserted automatically via a static instrumentation pass of the generator

code. The behavior of the generator across runs can be controlled by changing the semantics of these operators, some of which are described below.

Randomized. The simplest case is for the generator to be *randomized*. That is, the generator will follow a random execution path as governed by the values returned by its constituent operator calls. This can be achieved by simply randomizing the underlying operators, the semantics of which are given in Figure 2.4c. These semantics are rather straightforward — each operator simply returns a random element from the collection of possible values (defined by \mathcal{W} , the definition of which is given in Figure 2.4a). This collection of possible values \mathcal{W} is a function of the operator type (one of { **Select**, **Subset**, **OrderedSubset**, **Sequence** }) and the domain \mathcal{D} passed to the operator call.

Exhaustive (Depth-First). Another option is to have an *exhaustive* generator which systematically explores all possible execution paths as governed by the constituent operator calls. That is, all the operators work in unison by returning a fresh combination of values on each invocation of the generator. The operators also signal *Generator-Exhausted* when all possible values have been explored. Figure 2.4d presents the operator semantics that achieve this behavior. In particular, the semantics in Figure 2.4d enforce a *depth-first exhaustive* behavior across runs, where the generator explores all possible values of operator calls occurring later in the execution trace of the generator before exploring the ones occurring before. For example, when using the semantics in Figure 2.4d, the generator in Figure 2.2 will first explore all possible values of the **Select** call at Line 9 before moving on to the next possible value for the **Select** call at Line 5.

The operator semantics in Figure 2.4d uses three internal state variables t , σ and δ . The variable t keeps track of the number of operator calls made in the *current* invocation of the generator. The variable σ is a map from the operator call index to the choice to be made by the operator call in the current invocation of the generator. Note that the operator call index is distinct from the static identifier id as it keeps track of the *dynamic* index of the operator call in the generator call stack. For example, if an operator at a program location is called twice as a result of an enclosing loop, it will have two distinct entries in σ . Finally, δ represents a map from the operator call index to the collection of possible values \mathcal{W} as defined by the operator type and the passed domain \mathcal{D} . The variables σ and δ are initialized to empty maps before the first invocation of the generator but are persisted across the later ones. However, t is reset to zero before every fresh generator invocation. We also introduce a special operator called Op_{End} that is implicitly invoked at the end of each invocation in the generator. We now briefly explain the rationale behind all of these variables, Op_{End} and the rules themselves.

1. **OP-EXTEND** - This rule governs the behavior of the operator when it is invoked for the *first time* (as signified by $t \notin \text{dom}(\sigma)$). The operator returns the first value from \mathcal{W} and records this choice in σ . It also stores \mathcal{W} in δ for future use.
2. **OP-REPLAY** - The hypothesis $t \in \text{dom}(\sigma)$ signifies that this operator call needs to *replay* the choice as dictated by $\sigma(t)$.

3. OP-END-1 - This rule captures the first behavior of the special operator Op_{End} . It finds the last (deepest) operator call, indexed by k , that has not exhausted all possibilities and increments its entry in σ . This is key to enforcing depth-first semantics - a later call explores all possibilities before previous calls do the same. Note that it also *pops-off* the later entries ($> k$) from σ and δ . This is required as the generator may take an entirely new path based on the new value returned by this operator and may therefore make an entirely new set of operator calls. Overall, this maintains the invariant that σ stores the choice to be made by the operators in the current generator run.
4. OP-END-2 - The final rule covers the case when all operator calls have exhausted all possible values. This makes the special Op_{End} operator signal **Generator-Exhausted** after the last invocation of the generator, indicating that we have explored all possible executions of the generator.

Smart Semantics. Notice that the semantics presented in Figures 2.4c and 2.4d do not utilize the context χ or the static id passed to the operator. The significance of this is that given the same domain, the behavior of the operator is *fixed*, regardless of the actual input with which the generator is invoked. This is not suitable for tasks such as the one presented in Section 2.1, where the goal is to quickly find an argument combination to the `pivot` function such that when it is called on `inp_df`, it produces the target output `out_df`. In this case, we want to change the behavior of the operators based on the input and output dataframe and bias it towards the values that have a higher probability of guiding the generator execution in the right direction.

This *smart* behavior of operators is captured in the semantics presented in Figure 2.4e. The only difference with the semantics in Figure 2.4d is that the set of possible values \mathcal{W}_M for each operator is now the result of a function $Rank_{(Op,id)}$ that takes in the original collection of possible values, the passed domain \mathcal{D} as well as the context χ passed to the operator and reorders the values in the decreasing order of significance w.r.t to the task at hand. Note that the *Rank* function is sub-scripted by (Op, id) implying that every operator call can have a separate ranking function.

As shown in the generator in Figure 2.2, the context passed at every operator call is the input and output dataframe. Therefore given suitable ranking functions $Rank_{(Select,1)}$, $Rank_{(Select,2)}$ and $Rank_{(Select,3)}$, the generator can be biased toward producing an argument combination that, when passed to the `pivot` function along with the input dataframe `inp_df`, is likely to result in `out_df`.

2.2.2 Generator-Based Program Synthesis

We now describe how to build an *enumerative* synthesis engine using generators. The input to this engine is an input-output (I/O) example. The result is a program in the target language that produces the output when run on the input given in the I/O example. Our target language is the python pandas API. Figure 2.5 describes the basic algorithm behind

Table 2.1: List of Available Operators.

Operator	Description
<code>Select(domain)</code>	Returns a single item from <code>domain</code>
<code>Subset(domain)</code>	Returns an unordered subset, without replacement, of items in <code>domain</code>
<code>OrderedSubset(domain)</code>	Returns an ordered subset, without replacement, of items in <code>domain</code>
<code>Sequence(len)(domain)</code>	Returns an ordered sequence, with replacement, of items in <code>domain</code> with a maximum length of <code>len</code>

this engine in Python-like pseudo-code. The engine consists of two components — (1) a program candidate generator and (2) a checker that checks if the candidate program produces the correct output. The checker is rather straightforward to implement: we simply execute the program and test the exact match of its output to the target output. The bulk of the work is done by the program candidate generator.

2.2.2.1 Program Candidate Generator

A program candidate generator \mathcal{P} is a generator that, given an input-output example, generates program candidates. First, assume \mathcal{P} is a generator in *exhaustive* mode (see Section 2.2.1). That is, on each invocation, \mathcal{P} yields a program candidate that hasn't been produced so far. Figure 2.6 shows an excerpt of our program candidate generator for `pandas` programs. This generator produces straight-line programs, each of which is a sequence of up to `max_len` `pandas` function calls. The program given at the end of Section 2.1 is an example of such a candidate.

The generator in Figure 2.6 generates candidate programs as follows. First, it picks a sequence of functions from a list of supported functions (Lines 3-4). Then, for each function in the sequence, the generator selects the arguments (Lines 8-26), and computes the result by running the function with the arguments and stores it as an *intermediate* (e.g. Line 27). Intermediates are the outputs produced by previous functions in the sequence. These are essential to allow the generator to generate meaningful multi-function programs, where a function can operate on the output of a previously applied function. As shown in Lines 3-4, argument generation is done on a case-by-case basis depending on the given function. For example, for the function `pivot`, the generator follows the argument generation logic of Figure 2.2, applies the function with the selected arguments to a selected input or intermediate `df`, and stores the output as an intermediate. The program candidate generator can handle `pandas` functions that operate on multiple dataframes, e.g. `merge` on Lines 19-24, by selecting each dataframe from the set of input and intermediates (Lines 20-21).

$\mathcal{P}(\mathcal{D}) \stackrel{\text{def}}{=} \text{Power-Set of } \mathcal{D}$ $\text{Perms}(x) \stackrel{\text{def}}{=} \text{Set of all permutations of } x$ $W(Op, \mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{D} & \text{if } Op = \textit{Select} \\ \mathcal{P}(\mathcal{D}) & \text{if } Op = \textit{Subset} \\ \cup\{\text{Perms}(x) \mid x \in \mathcal{P}(\mathcal{D})\} & \text{if } Op = \textit{OrderedSubset} \\ \{(a_1, \dots, a_k) \mid k \leq l, a_i \in \mathcal{D}\} & \text{if } Op = \textit{Sequence}(l) \end{cases}$ $\mathcal{R}(W) \stackrel{\text{def}}{=} \text{Random Element from } W$	$\mathcal{W} \stackrel{\text{def}}{=} W(Op, \mathcal{D})$ $\sigma_k \stackrel{\text{def}}{=} \forall t. ((t < k) \Rightarrow (\sigma_k(t) = \sigma(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin \text{dom}(\sigma_k))$ $\delta_k \stackrel{\text{def}}{=} \forall t. ((t < k) \Rightarrow (\delta_k(t) = \delta(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin \text{dom}(\delta_k))$ $\mathcal{W}_M \stackrel{\text{def}}{=} \text{Rank}_{(Op, id)}(W(Op, \mathcal{D}), \mathcal{D}, \mathcal{X})$
--	--

(a) Common Definitions

(b) Common Definitions (continued)

$$\overline{Op(\mathcal{D}, \mathcal{X}, id) \Downarrow \mathcal{R}(W)} \quad \text{OP-RANDOM}$$

(c) Operator Semantics - Randomized

$\frac{t \notin \text{dom}(\sigma) \quad \delta' \equiv \delta[t := \mathcal{W}] \quad \sigma' \equiv \sigma[t := 0]}{\langle Op(\mathcal{D}, \mathcal{X}, id), \sigma, \delta, t \rangle \Downarrow \langle \mathcal{W}[0], \sigma', \delta', t+1 \rangle} \quad \text{OP-EXTEND}$	$\frac{t \notin \text{dom}(\sigma) \quad \delta' \equiv \delta[t := \mathcal{W}_M] \quad \sigma' \equiv \sigma[t := 0]}{\langle Op(\mathcal{D}, \mathcal{X}, id), \sigma, \delta, t \rangle \Downarrow \langle \mathcal{W}_M[0], \sigma', \delta', t+1 \rangle} \quad \text{OP-EXTEND}$
$\frac{t \in \text{dom}(\sigma)}{\langle Op(\mathcal{D}, \mathcal{X}, id), \sigma, \delta, t \rangle \Downarrow \langle \mathcal{W}[\sigma(t)], \sigma, \delta, t+1 \rangle} \quad \text{OP-REPLAY}$	$\frac{t \in \text{dom}(\sigma)}{\langle Op(\mathcal{D}, \mathcal{X}, id), \sigma, \delta, t \rangle \Downarrow \langle \mathcal{W}_M[\sigma(t)], \sigma, \delta, t+1 \rangle} \quad \text{OP-REPLAY}$
$\frac{\exists k. k \text{ is largest such that } (k \in \text{dom}(\sigma) \wedge \sigma(k) < \delta(k) - 1)}{\langle Op_{End}, \sigma, \delta, t \rangle \Downarrow \langle \sigma_k[k := \sigma(k) + 1], \delta_k, t+1 \rangle} \quad \text{OP-END-1}$	$\frac{\exists k. k \text{ is largest such that } (k \in \text{dom}(\sigma) \wedge \sigma(k) < \delta(k) - 1)}{\langle Op_{End}, \sigma, \delta, t \rangle \Downarrow \langle \sigma_k[k := \sigma(k) + 1], \delta_k, t+1 \rangle} \quad \text{OP-END-1}$
$\frac{\nexists k. (k \in \text{dom}(\sigma) \wedge \sigma(k) < \delta(k) - 1)}{\langle Op_{End}, \sigma, \delta, t \rangle \Downarrow \text{Generator-Exhausted}} \quad \text{OP-END-2}$	$\frac{\nexists k. (k \in \text{dom}(\sigma) \wedge \sigma(k) < \delta(k) - 1)}{\langle Op_{End}, \sigma, \delta, t \rangle \Downarrow \text{Generator-Exhausted}} \quad \text{OP-END-2}$

(d) Semantics - Depth-First Exhaustive

(e) Semantics - *Smart* Depth-First Exhaustive

Figure 2.4: Operator Semantics for Generators. σ and δ are initialized to empty maps before the first invocation of the generator. t is set to the integer zero before every invocation of the generator. Op_{End} is a special operator that is implicitly called at the end of each invocation of the generator. A detailed explanation is provided in Section 2.2.1

2.2.2.2 Building an Exhaustive Depth-First Enumerative Synthesis Engine

Using the exhaustive depth-first semantics for operators presented in Figure 2.4d for the generator in Figure 2.6 gives an exhaustive depth-first synthesis engine. This means that the engine explores all possible program candidates and in depth-first order i.e. it explores all possible programs using the same sequence of functions before exploring another sequence. Also, when enumerating the arguments, it explores all values of a later argument before moving on to the next value for the previous argument.

```

1 def synthesize(input, output, max_len):
2     generator = generate_candidates(input, output, max_len)
3     while (not generator.finished()):
4         candidate = next(generator)
5         if candidate(input) == output:
6             return candidate

```

Figure 2.5: Generator-Based Enumerative Synthesis Engine

2.2.2.3 Building a Smart Enumerative Synthesis Engine

The generator in Figure 2.6 describes a space of programs that is extremely large for such an enumerative pandas synthesis engine to explore in reasonable time. This generator supports 119 pandas functions, each taking 3 arguments on average. This causes an *enormous* combinatorial explosion in the number of argument combinations and choices made by the generator operators.

Hence, we need a *smart* generator that tailors itself to the presented synthesis task. That is, we need to use the smart semantics for operators presented in Figure 2.4e. For the generator in Figure 2.6, the context passed to each operator call is explicitly shown. The function sequence selection, as well as the selection of dataframes on which the functions operate (Lines 4, 9, 20, 21) all take the input-output example along with any intermediates as the context. The operator calls used to select values for arguments depends primarily on the dataframe(s) on which the function will be run, so only that dataframe and the output is passed as context.

With this formulation in place, we can now define the $Rank_{(Op, id)}$ function that is at the heart of the semantics in Figure 2.4e. Given the domain \mathcal{D} and the context χ passed to Op , this function reorders the space of possible values $W(Op, \mathcal{D})$ according to a probability distribution over this space. We exploit the recent advances in the area of deep learning and define these *Rank* functions per operator using novel neural network models that we describe in the following section. We call generators that use operators backed by these neural network models *Neural-Backed Generators*.

2.2.3 Neural-Backed Generators for Pandas

In AUTOPANDAS, we use neural networks to define the *Rank* functions for the operators used in our generators. In short, we design a neural network model for each kind of operator (see Table 2.1). The first time an operator Op is called with a particular domain \mathcal{D} and context χ , a query is constructed using \mathcal{D} and χ . This query is passed to the neural network model, which returns a probability distribution over the possible values for the operator (as defined by $W(Op, \mathcal{D})$ in Figure 2.4a). The *Rank* function then uses this distribution to reorder the elements in $W(Op, \mathcal{D})$ in the decreasing order of probabilities (\mathcal{W}_M in Figure 2.4b). The

```

1 @generator
2 def generate_candidates(input, output, max_len):
3     functions = [pivot, drop, merge, ...]
4     function_sequence = Sequence(max_len)(functions, context=[input, output], id=1)
5     intermediates = []
6     for function in function_sequence:
7         c = [input, *intermediates, output]
8         if function == pivot:
9             df = Select(input + intermediates, context=c, id=2)
10            arg_col = Select(df.columns, context=[df, output], id=3)
11            arg_idx = Select(df.columns - {arg_col}, context=[df, output], id=4)
12            if isinstance(df.index, pandas.MultiIndex) and arg_idx is None:
13                arg_val = None
14            else:
15                arg_val = Select(df.columns - {arg_col, arg_idx},
16                               context=[df, output], id=5)
17            args = (df, arg_col, arg_idx, arg_val)
18
19            elif function == merge:
20                df1 = Select(input + intermediates, context=c, id=6)
21                df2 = Select(input + intermediates, context=c, id=7)
22                common_cols = set(df1.columns) & set(df2.columns)
23                arg_on = OrderedSubset(common_cols, context=[df1, df2, output], id=8)
24                args = (df1, df2, arg_on)
25            # Omitted code: case for each function
26                :
27            intermediates.append(function.run(*args))
28
29    return function_sequence

```

Figure 2.6: A Simplified Program Candidate Generator for pandas Programs.

operator functions as before, but now returns values in an order conditioned on the context. We now define the query concretely, its encoding as well as the neural network architectures for each operator.

2.2.3.1 Neural-Network Query

The query \mathcal{Q} to each neural network model, regardless of the operator, is of the form $\mathcal{Q} = (\mathcal{D}, \chi)$ where \mathcal{D} and χ are the domain and context passed to the operator.

2.2.3.2 Query Encoding

Encoding this query into a neural network suitable format poses several challenges. Recall that the context and the domain passed to operators in the `pandas` program candidate generator (Figure 2.6) contain complex structures such as dataframes. Dataframes are 2-D structures that can contain arbitrary Python objects as primitive elements. Even restricting ourselves to strings or numbers, the set of possible primitive elements is infinite. This renders all common value-to-value map-based encoding techniques popular in machine learning, such as one-hot encoding, inapplicable. At the same time, the encoding needs to retain enough information about the context to generalize to unseen queries which may occur when the synthesis engine is deployed in practice. Therefore, simply abstracting away the exact values is not viable. In summary, a suitable encoding needs to (1) abstract away only irrelevant information and (2) be suitably structured for neural processing. To this end, we designed a graph-based encoding that possesses all these desirable properties.

Graph-Based Encoding. We now describe how to encode the domain \mathcal{D} and the context χ as a graph, consisting of nodes, edges between pairs of nodes, and labels on nodes and edges. The overall rationale is that it is not as much the concrete values but rather the *relationships* amongst values that really encode the transformation at hand. That is, relationship edges should be sufficient information for a neural network to learn the transformation. For example, the essence of the transformation represented by Figure 2.1 is that the values of the column ‘Category’ now become the columns of the pivoted dataframe, with the ‘Date’ column as an index, and the ‘Expense’ as values. One could replace the concrete name using an arbitrary one-to-one mapping and still obtain the same transformation. This is true only for reshaping operations such as joining, grouping, and pivoting. In contrast, the transformations are only equivalent up to homomorphisms on the values when statistical or aggregation operations are involved.

Recall that the domain and context are essentially collections of elements. Therefore, we first describe how to encode each such element e individually as a graph G_e . Later we describe the procedure to combine these graphs into a single graph $G_{\mathcal{Q}}$ representing the graph-encoding of the full query \mathcal{Q} . Figure 2.7 shows the graph-encoding of the query generated as a result of the `Select` call at line 4 in Figure 2.2 and will be used as a running example.

Encoding Primitives. If the element e is a primitive value (strings, ints, float, lambda, NaN etc.), its graph encoding G_e contains a single node representing e . This node is assigned a label based on the data type of the element as well as the *source* of the element. The source of an element indicates whether it is part of the domain, if it is one of the input or output, if it is one of the intermediate results obtained after applying the first few functions, or none of these.

Encoding DataFrames. If the element e is a dataframe, each cell element in the dataframe is encoded as a node in the graph G_e . The label of the node includes the type of the element (string, number, float, lambda, NaN, etc.). The label also includes the source of the dataframe, i.e., whether the dataframe is part of the domain, input, output, intermediate, or none of these. We also add nodes to G_e that represent the schema of the dataframe by creating a node for each row index and column name of the dataframe. Finally, we add a *representor* node to G_e that represents the whole of the dataframe. The label of this node contains the type “dataframe” as well as the source of the parent dataframe. Note that this additional representor node is not created when encoding primitive elements. The node representing the primitive element itself acts as its representor node.

The graph encoding of a dataframe also contains three kinds of edges to retain the structure of the dataframe. The first kind is adjacency edges. These are added between each pair of cell nodes, column name nodes, or row index nodes that are adjacent to each other in the dataframe. We only add adjacency edges in the four cardinal directions. The second kind is indexing edges, which are added between each column name node (resp. row index node) and all the cell nodes that belong to that column (resp. row). Finally, the third kind of edge is a representation edge, between the representor node to all the other nodes corresponding to the contents of the dataframe.

Encoding the Query Q . Finally, to encode Q , we construct G_e for each element in \mathcal{D} and χ as described above, and create G_Q such that it contains these G_e s as sub-graphs. Additionally, to capture relationships amongst these elements, we add a fourth kind of edge - *the equality edge*, between nodes originating in different G_e s such that the elements they represent are equal. Formally, we add an equality edge between nodes n_1 and n_2 if $n_1 \in G_{e_i} \wedge n_2 \in G_{e_j} \wedge i \neq j \wedge Value(n_1) = Value(n_2)$ where *Value* is a function that given n , retrieves the value encoded as n . For representor nodes, *Value* returns the whole element it represents. For example, for a dataframe df , *Value(df)* would return df itself as the representor node.

Equality edges are key to capturing relationships between the inputs and the output in the I/O example, as well as the domain \mathcal{D} and the I/O example. The neural network model can then learn to extract these relationships and use them to infer the required probability distribution.

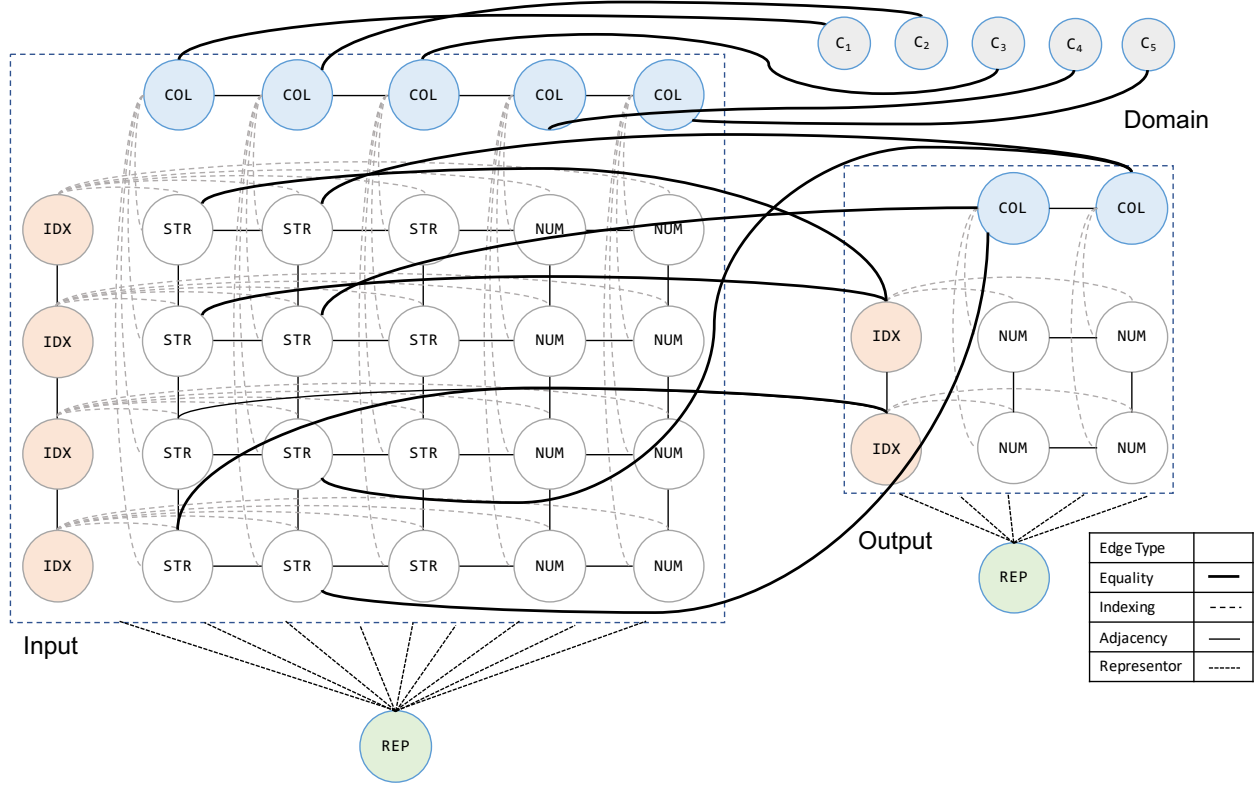


Figure 2.7: Graph encoding of the query passed to the `Select` call at Line 4 in Figure 2.2, on the I/O example from Figure 2.1.

2.2.3.3 Operator-Specific Graph Neural Network Models

Given the graph-based encoding $G_{\mathcal{Q}}$ of a query \mathcal{Q} , we feed it to a graph neural network model. Each operator has a different model. These models are based on the gated graph neural network, introduced by [69]. We base our model on the implementation by Microsoft [76, 4] as it was the most efficient available at the time of development. We first describe the common component of all the neural network models. Then, we provide an individual description for the neural network model for each of the operators listed in Table 2.1.

The input to all our network models is an undirected graph $G = (V, E)$ where V are the nodes and E are the edges. Every edge $e \in E$ is a 3-tuple $\langle v_s, v_t, t_e \rangle$ where v_s and v_t are the source and target nodes respectively. The type t_e of the edge is one of $\Gamma_e \equiv \{\text{adjacency}, \text{indexing}, \text{representer}, \text{equality}\}$ and is one-hot encoded.

Each node v is assigned a state vector $h_v \in \mathbb{R}^d$ where d is a hyper-parameter. We initialize the vector to the node embedding $h_v^{(0)} = M(v)$ where $M : V \rightarrow \mathbb{R}^d$ maps each node to a one-hot encoding of its label of size d . The network then propagates information via r rounds of *message passing*. During round k ($0 \leq k < r$), messages are sent across edges. In particular, for each edge (v_s, v_t, t_e) , v_s sends the message $m_{v_s \rightarrow v_t} = f_k(h_{v_s}^{(k)}, t_e)$ to v_t . Our

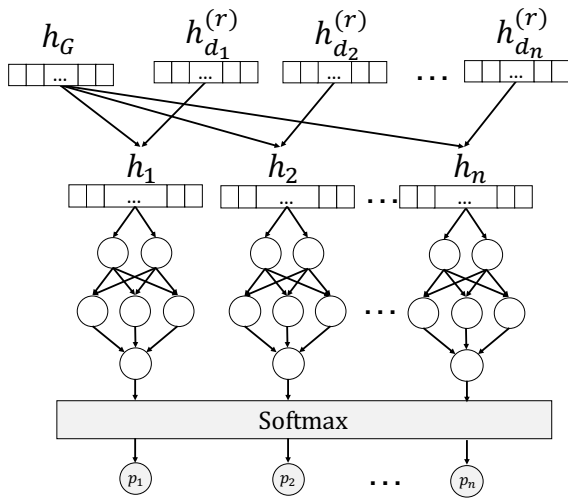
$f_k : \mathbb{R}^{d+|\Gamma_e|} \rightarrow \mathbb{R}^d$ is a single linear layer. These are parameterized by a weight matrix and a bias vector, which are learnable parameters. Each node v aggregates its incoming messages into $m_v = g(\{m_{v_s \rightarrow v} \mid (v_s, v, t_e) \in \mathcal{E}\})$ using the aggregator g . In our case, we take g to be the element-wise mean of the incoming messages. The new node state vector $h_v^{(k+1)}$ for the next round is then computed as $h_v^{(k+1)} = GRU(m_v, h_v^{(k)})$ where GRU is the gated recurrent unit [24] with start state as $h_v^{(k)}$ and input m_v . We use $r = 3$ rounds of message passing, as we noticed experimentally that further increasing the number of message passing rounds did not increase validation accuracy.

After message passing is completed, we are left with updated state vectors $h_v^{(r)}$ for each node v . Now depending on the operator, these node vectors are further processed in different ways as described below to obtain the corresponding probability distributions over the space of values defined by the operator (see Figure 2.4a). A graphical visualization is also provided in Figure 2.8.

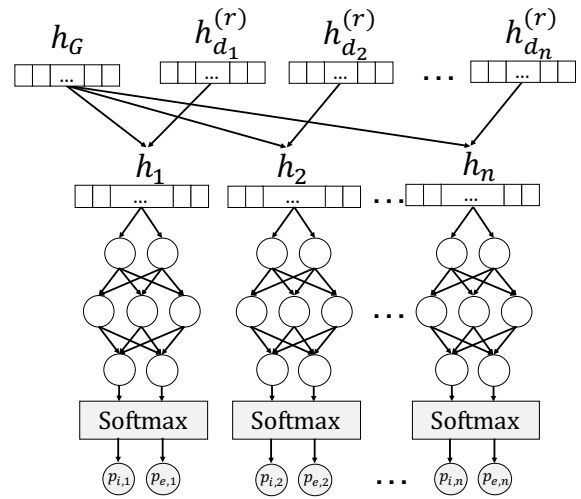
For each of the operators below, we deliberately choose the simplest architecture that allows the computation of the correct probability distribution. This helps keep the focus on the quality of the message passing step, forcing the models to learn good representations for the nodes and edges rather than compensating at the outer layers.

Select : We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G . We now concatenate h_G with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_i for each element in the domain \mathcal{D} in the query \mathcal{Q} , to obtain vectors $h_i = h_G \circ h_{d_i}^{(r)}$. We pass the h_i s through a multi-layer perceptron with one hidden layer and a one-dimensional output layer and apply softmax over the output values for all the elements to produce a probability distribution over the domain elements (p_1, \dots, p_n) . During inference, the entire inferred distribution is returned as the result, while during training we compute cross-entropy loss w.r.t this distribution and the correct distribution where $p_i = 1$ for the correct choice i and $\forall j \neq i, p_j = 0$. Figure 2.8a shows an illustration of the model.

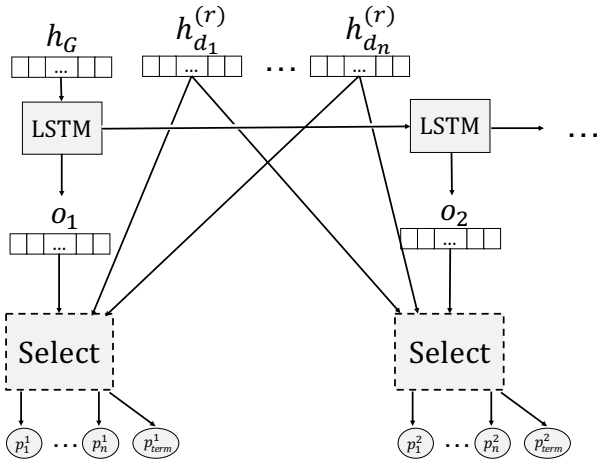
Subset : As in **Select**, we perform element-wise sum-pooling of the node state vectors and concatenate it with the state vectors of representor nodes to obtain the vectors $h_k = h_G \circ h_{d_k}^{(r)}$ for each element $d_k \in \mathcal{D}$. However, we now pass the h_k s through a multi-layer perceptron with one hidden layer and apply softmax activation on the output layer to obtain a distribution (p_{i_k}, p_{e_k}) over two label classes “include” and “exclude” for each of the domain element $d_k \in \mathcal{D}$ individually. Recall that the space of possible outputs for the **Subset** operator is the power-set of the domain \mathcal{D} . The probability of these labels corresponds to the probability with which an element is included and excluded from the output set respectively. To compute the probability distribution, the probability of each possible output set is computed as simply the product of the “include” probabilities for the elements included in the set and the “exclude” probabilities for the elements excluded from the set. Again, this distribution is returned as the result during inference, while during training, loss is computed w.r.t this distribution and the correct individual distribution of the elements where $p_{i_k} = 1 \wedge p_{e_k} = 0$ if element d_k is present in the correct output, else $p_{i_k} = 0 \wedge p_{e_k} = 1$. Figure 2.8b shows an



(a) Illustration of the **Select** model.



(b) Illustration of the **Subset** model.



(c) Illustration of the **OrderedSubset/Sequence** model. The box label “Select” expands to (a).

Figure 2.8: Operator-specific neural network architectures.

illustration of the model.

OrderedSubset and Sequence : We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G . We then pass h_G to an LSTM that is unrolled for $T + 1$ time-steps, where $T = |\mathcal{D}|$ for **OrderedSubset** and $T = l$ for **Sequence**(1) where 1 the max-length parameter passed to **Sequence**. The extra time-step is to accommodate a terminal token which we describe later. For each time-step t , the output o_t is concatenated with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_i s for each element in the domain passed to the operator to obtain vectors $h_i^t = o_t \circ h_{d_i}^{(r)}$. At time-step t , in a similar fashion as **Select**, a probability distribution is then computed over the domain elements plus an arbitrary terminal token *term*. The terminal token is used to indicate the end of a sequence/set. Now, to compute the probability distribution, the probability of each set or sequence (a_0, \dots, a_k) where $(k \leq T)$ is simply the product of probabilities of a_i at time-step i and the probability of the terminal token *term* at time-step $k + 1$. As before, this distribution is directly returned during inference, while during training, the loss is aggregated over individual time steps; the loss for a time-step is computed as described in **Select**. Figure 2.8c illustrates the model.

All the models are trained with the ADAM optimizer [60] using cross-entropy loss.

2.2.4 Training Neural-Backed Generators for Pandas

A Neural-Backed Generator consists of operators backed by *Rank* functions that influence their behavior. We implement these *Rank* functions using neural networks. as described in Section 2.2.3.3. Training each of these networks for each call to an operator with static ID id requires training data consisting of tuples of the form $\mathcal{T}_{id} = (\chi, \mathcal{D}, c)$ where c is the correct choice to be made by the operator call with static id id . Put another way, the neural network behind the operator call at location id is trained to predict the choice c with the highest probability given the context χ and domain \mathcal{D} .

Unfortunately, such training data is not available externally as it is highly specific to our generators. Therefore, we aim to synthesize our training data automatically, i.e., synthesize a random tuple containing a context χ , domain \mathcal{D} , and the target choice c . This is a highly non-trivial problem, as there are two strong constraints that need to be imposed on χ , \mathcal{D} , and c for this tuple to be a useful training data-point. First, the random context, domain, and choice should be *valid*. That is, there should exist *an execution of the generator* for some input such that the operator call in question receives the random context and domain as input and makes the same choice. Second, this tuple of context, domain, and choice should be *meaningful*, i.e., the choice should lead to progress on the task contained in the context. In our synthesis setting, this translates to the property that the generator makes a step towards producing a program that actually produces the output from the input as passed in the context. We rely on two key insights to solve these problems for our pandas program candidate generator.

Suppose we have tuples of the form $\langle \vec{t}_{in}, t_o, P, K \rangle$ where P is a pandas program such that $P(\vec{t}_{in}) = t_o$ i.e. it produces t_o when executed on inputs \vec{t}_{in} . Also, K is the sequence of choices made by the operators in the generator such that the generator produces the program P when it is fed \vec{t}_{in} and t_o as inputs. Then, it is straightforward to extract training data tuples (χ, \mathcal{D}, c) for each operator call by simply running the generator on \vec{t}_{in} and t_o and recording the concrete context χ and domain \mathcal{D} passed to the operator and forcing the operator to make the choice c . These tuples are also *meaningful by construction*, as the operators make choices that lead to the generation of the program P , which solves the synthesis task described by \vec{t}_{in} and t_o .

The second insight is that we can obtain these $\langle \vec{t}_{in}, t_o, P, K \rangle$ tuples by using the generator itself. We generate random inputs \vec{t}_{in} (DataFrames), run the generator on \vec{t}_{in} using the randomized semantics presented in Figure 2.4c while simultaneously recording the choices made as K . The program P returned by the generator is then run on \vec{t}_{in} to yield t_o .

The sheer size of APIs such as `pandas` presents another problem in this data generation process. The large number of functions yields a huge number of possible sequences of these functions (Lines 3-4 in Figure 2.6). Even when considering sequences of length ≤ 3 , the total number of sequences possible from the 119 `pandas` functions we support is $\sim 500,000$. Generating enough examples for all function sequences to cover a satisfactory portion of all the possible argument combinations is prohibitively expensive and would result in a dataset of enormous size that cannot be processed and learned from in a reasonable amount of time.

However, not all sequences actually occur in practice. Practitioners of the library come up with sequences that are useful in solving real-world examples. So, we mine Github and StackOverflow to collect the function sequences used in the real world. We were able to extract ~ 4300 sequences from both these sources. Then, while generating the tuples $(\vec{t}_{in}, t_o, P, K)$ using randomized semantics, we tweak the semantics of just the call to `Sequence` at Line 4 in Figure 2.6 to randomly return sequences from only this mined set of sequences.

2.3 Evaluation

We first evaluate the feasibility and effectiveness of our technique by answering two main research questions:

RQ1: Can AUTO-PANDAS solve real-world input-output example benchmarks? We evaluate the end-to-end ability of AUTO-PANDAS to synthesize solutions for real-world benchmarks within a practical time bound. We also compare with a baseline using models in a limited fashion.

RQ2: Do smart operators, which are backed by models, make better choices than their deterministic and randomized counterparts? In other words, we evaluate the efficacy of every model backing an individual operator in the main generator used in

AUTOPANDAS, in terms of the quality of choices made as opposed to using deterministic or randomized semantics.

2.3.1 Implementation

We implement the overall technique described in Section 2.2 in a tool called AUTOPANDAS. AUTOPANDAS consists of 25k lines of Python code and uses Tensorflow [73] to implement the neural network models. The code is available at <https://github.com/rbavishi/autopandas>.

2.3.2 Training and Setup

We generated 6 million (input, output, program, generator choices) training tuples (as described in Section 2.2.4) containing 2 million tuples each for programs consisting of one, two, and three function calls. Similarly, we generate 300K validation tuples with 100K tuples each for the three function sequence lengths. From these tuples, we extract training and validation data for the 320 operator calls in our program candidate generator for `pandas`, and train their respective models for 10 epochs on four Nvidia Titan V GPUs. We finished training all the models in 48 hours. All our synthesis experiments are run on a single 8-core machine containing Intel i7-7700K 4.20GHz CPUs running Ubuntu 16.04.

2.3.3 RQ1: Performance on Real-World Benchmarks

We evaluated AUTOPANDAS on 26 benchmarks taken from StackOverflow questions containing the `dataframe` tag. We ran AUTOPANDAS with a time-out of 20 minutes and used smart depth-first enumeration semantics for the program candidate generator. We also impose an upper bound on the number of REPLAYS an operator is allowed to make (see the OP-REPLAY semantics in Figure 2.4d). This prevents any operator from limiting the scope of exploration when it can return a very large number of values given a single domain. In our experiments, we set a bound of 1000. For comparison, we also implement a baseline version of AUTOPANDAS called BASELINE that follows depth-first exhaustive enumeration semantics (Figure 2.4d) for all operator calls except the **Sequence** invocation. The rationale is that given the size of the search space, it is more meaningful to compare the performance of the models backing the exploration of function arguments given the same function sequences. Table 2.2 contains the results.

The column *Depth* contains the length of the function sequence used in the official solution for the benchmark. *Cand. Explored* denotes the number of candidates both approaches had to check for correctness before arriving at one which produces the target output. *Seq. Explored* contains the number of function sequences explored (by the **Sequence** call at Line 4 in Figure 2.6), while the *Time* column contains the time taken (in seconds) to produce a solution, if any.

AUTOPANDAS can solve 13 out of the 26 benchmarks in under 30 seconds and 17 out of 26 in under 20 minutes. The BASELINE approach also solves 13 out of 26 benchmarks in under 30 seconds, but 14 out of 26 in under 20 minutes. While the gap between the two approaches in the number of benchmarks solved as well as the time taken is small, AUTOPANDAS explores $5\times$ fewer programs than BASELINE on average. This does not translate to performance gains due to two reasons: (1) the absolute difference in the number of programs is not enough to cause differences in the time taken to verify candidates, and (2) the overhead of neural models is not low enough. We expect the latter issue to go away with the use of more modern implementations.

Both approaches tend to miss the 20-minute mark more often on benchmarks with higher depths or those involving computation. Poor performance on computational benchmarks is expected as the graph encodings in AUTOPANDAS only contain equality edges and, as such, cannot capture computational relationships.

2.3.4 RQ2: Analysis of Neural Network Models

In this section, we perform a deeper evaluation of the performance of the individual models used in AUTOPANDAS.

2.3.4.1 Function Sequence Prediction Performance

We single out the call to **Sequence** in our program candidate generator as it is the component most critical to the performance of the generator and dissect the performance of the neural network model backing it; on our synthetic validation dataset in Figure 2.9. In particular, we measure top-1 to top-10 accuracies on a per-sequence basis. Recall that these are the sequences mined from GitHub and StackOverflow. Figures 2.9a-2.9c show the performance of the model when predicting sequences of lengths 1, 2 and 3 respectively. As expected, the performance for shorter sequences is better as the logical distance between the input and output is lower, and therefore the encoding can capture sufficient information. Another reason for poorer accuracies at higher lengths is the fact that large APIs like **pandas** functions often have overlapping semantics. Therefore multiple sequences may produce viable solutions for a given output example. This is reinforced by the results on real-world benchmarks in Table 2.2. In particular, the numbers in the “Sequences Explored” column for AUTOPANDAS suggest that the model indeed predicts useful sequences, even if they don’t match the ground-truth sequence.

Figures 2.9d-2.9f present the expected accuracies of a purely random model on the same dataset. As expected, the accuracies are almost zero (there is a slight gradient in Figure 2.9d). The sheer number of possible sequences makes it improbable for a random model to succeed on this task; even our baseline benefited from the neural model’s predictions.

Table 2.2: Performance on Real-World Benchmarks. Dashes (-) indicate timeouts by the technique. AP represents AUTOPANDAS and BL stands for BASELINE

Benchmark	Depth	Candidates Explored		Sequences Explored		Solved		Time(s)	
		AP	BL	AP	BL	AP	BL	AP	BL
SO_11881165	1	15	64	1	1	Y	Y	0.54	1.46
SO_11941492	1	783	441	8	8	Y	Y	12.55	2.38
SO_13647222	1	5	15696	1	1	Y	Y	3.32	53.07
SO_18172851	1	-	-	-	-	N	N	-	-
SO_49583055	1	-	-	-	-	N	N	-	-
SO_49592930	1	2	4	1	1	Y	Y	1.1	1.43
SO_49572546	1	3	4	1	1	Y	Y	1.1	1.44
SO_13261175	1	39537	-	18	-	Y	N	300.20	-
SO_13793321	1	92	1456	1	1	Y	Y	4.16	5.76
SO_14085517	1	10	208	1	1	Y	Y	2.24	2.01
SO_11418192	2	158	80	1	1	Y	Y	0.71	1.46
SO_49567723	2	1684022	-	2	-	Y	N	753.10	-
SO_13261691	2	65	612	1	1	Y	Y	2.96	3.22
SO_13659881	2	2	15	1	1	Y	Y	1.38	1.41
SO_13807758	2	711	263	2	2	Y	Y	7.21	1.81
SO_34365578	2	-	-	-	-	N	N	-	-
SO_10982266	3	-	-	-	-	N	N	-	-
SO_11811392	3	-	-	-	-	N	N	-	-
SO_49581206	3	-	-	-	-	N	N	-	-
SO_12065885	3	924	2072	1	1	Y	Y	0.9	4.67
SO_13576164	3	22966	-	5	-	Y	N	339.25	-
SO_14023037	3	-	-	-	-	N	N	-	-
SO_53762029	3	27	115	1	1	Y	Y	1.90	1.50
SO_21982987	3	8385	8278	10	10	Y	Y	30.80	13.91
SO_39656670	3	-	-	-	-	N	N	-	-
SO_23321300	3	-	-	-	-	N	N	-	-
Total						17/26	14/26		

2.3.4.2 Comparison with Deterministic and Randomized Semantics

We demonstrate the efficacy of the smart semantics for operators by comparing the top-k accuracy of the underlying neural network models with corresponding deterministic and randomized baselines. In the deterministic baseline, the order in which operators return values is fixed for a given input domain (see Figure 2.4d). In the randomized baseline, the operator returns values in a random order (see Figure 2.4c). We expect the neural network approach (see Figure 2.4e) to perform better than both these baselines as it utilizes the context. Figure 2.10 shows the results.

We see that while a randomized approach smooths results compared to the deterministic approach (ref. Figure 2.10c vs. Figure 2.10b), both still have significant difficulty on certain operator calls (top-left corners of all graphs). The neural network model performs quite

well in comparison. There are operator calls where all three approaches perform poorly or all perform well. The former can be attributed to insufficient information in the context. For example, if a `pandas` function supports two modes of operation which can both lead to a solution, the model may be penalized in terms of accuracies but may not affect its performance in the actual task. The latter case, where all approaches perform well, can be mostly attributed to small domains. For example, many `pandas` functions take an `axis` argument that can only take the value 0 or 1, which can be modeled as `Select({0,1})` in the generator. Hence the top-2 accuracy of all the approaches will be 100%.

Overall, we see that the neural-backed operators arrive at the correct ‘guess’ much more quickly than their randomized or deterministic counterparts, thus helping the generator as a whole to arrive at the solution more efficiently. In fact, the accuracies in Figure 2.10 are quite high for the neural-backed operators overall. We think this is a very encouraging result, as we are able to learn useful operator-level heuristics.

The contrast between the overall high accuracies in Figure 2.10a and the accuracies in Figure 2.9 suggests that the biggest bottleneck is predicting the correct function sequence. This and the previous observation are reinforced by the columns containing the number of candidates and function sequences explored in Table 2.2.

2.4 Discussion

We elaborate on the main talking points behind the design of AUTOPANDAS, the limitations, and opportunities for improvement.

2.4.1 Generator Implementation

Our program candidate generator has been hand-written by consulting the Pandas source code and, therefore, may not be completely faithful to the full internal usage specification of all functions. Due to the complexity of the API pre-conditions for pandas, writing this generator required substantial manual labor. While the writer of such a generator needs only knowledge of the API and not of the synthesis engine, this still poses a practical barrier to implementing the technique for other APIs. Our technique requires that all programs be expressible in terms of the generator, so the generator restricts the space of programs we can synthesize. The current generator behind AUTOPANDAS only expresses programs consisting of a sequence of API calls of a maximum length of three.

2.4.2 Representative Training Data

Another limitation is that our synthetic data may not be representative of the usage of pandas in the real world. For example, our dataset may contain many transformations that do not “look useful” to a human, and thus our models may be biased away from useful ones. This may be a factor in why we are unable to synthesize 9 of our real-world benchmarks

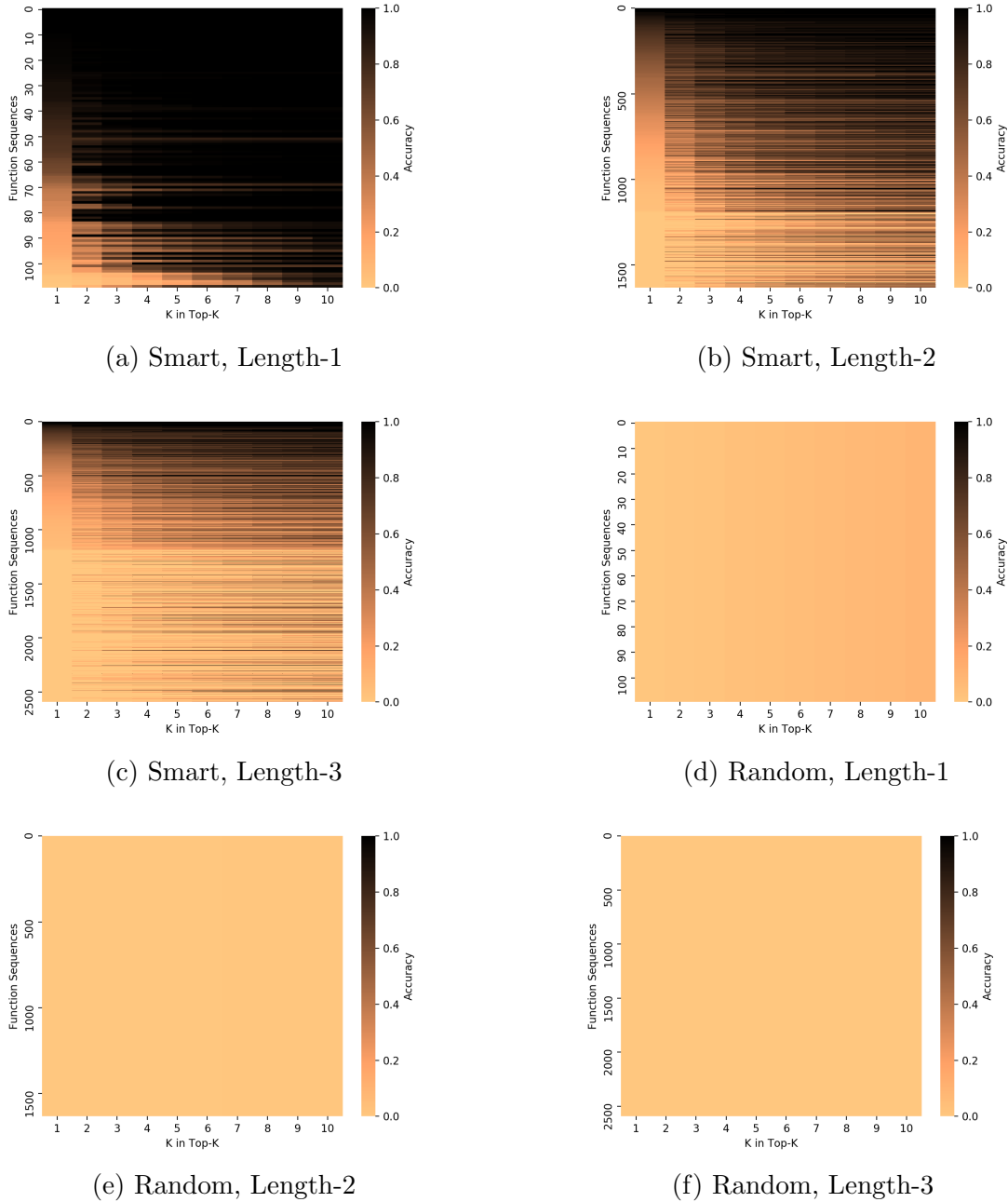
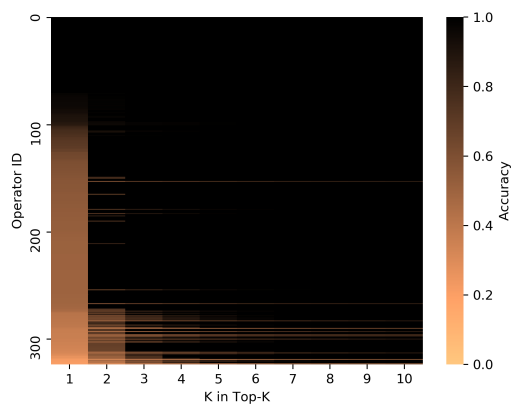
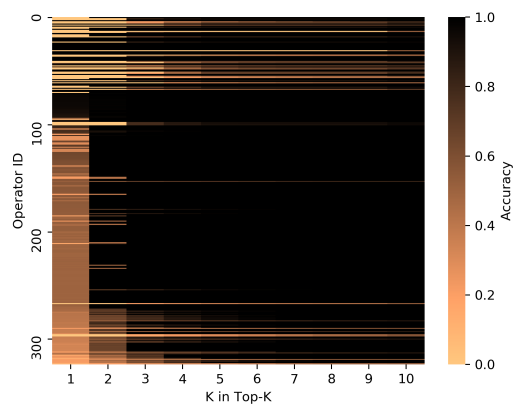


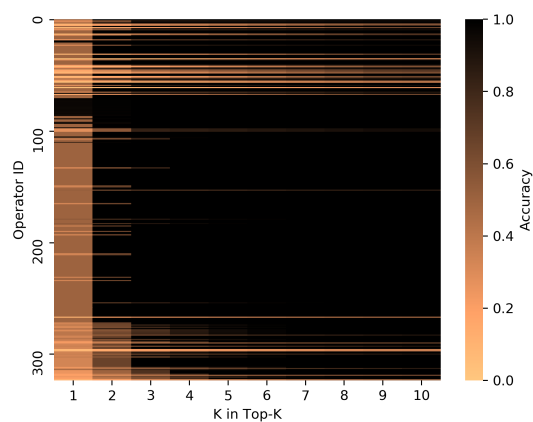
Figure 2.9: Smart Model Accuracies on Function Prediction Task, compared to a Random Baseline. Per-sequence Top- k accuracies provided. Color gives accuracy; darker is better. The color point (x, y) gives the top- x accuracy for sequence with ID y . Sequence IDs are sorted based on top-1 accuracy of the smart model.



(a) Smart Model



(b) Baseline-Deterministic



(c) Baseline-Randomized

Figure 2.10: Per-operator Top- k accuracies. Color gives accuracy; darker is better. The color point (x, y) gives the top- x accuracy for operator with ID y . Operator IDs are sorted based on the top-1 accuracy of the smart model.

within the timeout. Further, our training dataset may not contain enough data points for effective training for each operator call in our candidate generator.

2.4.3 Ease of Providing I/O Examples

Finally, while input-output examples are arguably a usable specification, from discussions with potential users, we find that they do suffer from two main issues. Firstly, input-output examples can be tedious to provide if the output involves computation that is hard to do by hand or if it needs to be specially crafted from scratch because the original data is too complex. Secondly, input-output examples lose information — consider an input-output example representing any form of computation or aggregation. The output, and consequently our graph encoding, do not capture the computational relationship between the input and output. Interestingly, the assistant introduced in the next chapter, GAUSS, tackles exactly these issues.

2.5 Summary

In this chapter, we introduced the concept of neural-backed generators, a novel combination of generators and machine learning, specifically graph neural networks, to help capture large and complex candidate program spaces as well as efficiently search for a solution. Neural-backed generators help retain a high degree of expressivity while retaining decent performance. Our implementation of AUTOPANDAS supports 119 functions out of the 136 available at the time of development. We hand-wrote a candidate program generator for the subset of the Python `pandas` API dealing with dataframe transformations. The candidate program generator includes argument generators for each of these 119 functions, each of which captures the key argument constraints for the given function. Experiments suggest that AUTOPANDAS excels at solving *shallow* tasks where only one or two functions are required to perform the desired transformation. Thus, AUTOPANDAS is effective at reducing the *recall* burden — remembering the right function names or the arguments.

However, AUTOPANDAS does suffer from a number of limitations. It does not perform well for benchmarks involving computations as the output, as the graph encoding does not capture the relevant information, effectively reducing the search to trial-and-error. In the next chapter, we address this issue by introducing an element of interaction in traditional input-output examples.

Chapter 3

Gauss: Improving Code Generation via User Intent Graphs

Chapter 2 described AUTOPANDAS, an assistant solving the programming-by-example problem of generating `pandas` code for data processing tasks given an input-output dataframe. The design of AUTOPANDAS was influenced by the need to retain expressivity; AUTOPANDAS supported 119 out of 136 `pandas` functions available for dataframes. To jointly combat the challenge of representing the space of programs and efficiently searching it to find the correct program, we introduced the concept of *neural-backed generators*. Neural-backed generators replace or *back* the traditional randomized *choice-points* in QuickCheck-style [26] generators with machine learning models. Specifically, these models are graph neural networks, with their design based on the insight that dataframe transformations can be represented as graphs where the nodes correspond to the cells, columns, and rows of the dataframes, and edges correspond to the relationships between the input and output nodes.

We found AUTOPANDAS performance to be poor on benchmarks involving aggregation or computation. This was not surprising due to the design of the encoding process that converts concrete input-output examples into graphs to feed to the models. Specifically, the encoding into graphs only preserved equality and structural edges; it could not automatically infer more complex relationships. This meant that the models had little information about the underlying operation through the supplied graph, and hence the enumerative search simply devolved into trial and error.

A natural research direction in pursuit of improving performance on this class of benchmarks is to innovate on the graph-encoding component, presumably with the goal of inferring more complex relationships. Instead, in this chapter, we look at how we can innovate at the level of the specification itself by identifying deeper, practical issues in the use of input-output examples itself as specifications for this class of tasks.

Consider the example in Figure 3.1. Even as a human, The source of the number 102.5 in the first cell of the output is not immediately clear from the input-output example alone. However, the user who created this example *knows exactly* how they derived this number. It is simply the mean of all the numerical cells in rows labeled “Pants”: 50, 70, 100, and

input	0	Type	Low	High
	1	Pants	50	70
	2	Pants	100	190
	3	Shirts	80	110

output	0	Type	Avg
	1	Pants	102.5
	2	Shirts	95

Figure 3.1: Input-output examples alone discards user intent information that was present while creating the output. In this example, it is not immediately clear that 102.5 is the mean of 50, 70, 100, and 190.

190. The user could have viably provided this information given a suitable user interface (UI). Moreover, this information could not only potentially help speed up synthesis but also prevent overfitting by filtering programs whose semantics do not exactly match the intent. The availability of such additional information can also help with removing redundancy elsewhere in the example. Consider the example in Figure 3.1 again. If the output cell containing 102.5 is provided along with precise information about its computation, the rest of the cells in the output are not as important anymore from the perspective of specification. Put differently, if one were to simply explain how the single 102.5 entry was computed to a fellow programmer, chances are that they would come up with the correct solution as the simplest program that produces an output containing 102.5 indeed produces the rest of the output entries in Figure 3.1. Thus, there is a possibility of ultimately reducing the overall burden on the user — users only need to provide precise relationships between a part of the output and input. We call this reduced input-output example a *partial* input-output example, in line with the terminology used in VISER [125].

In this chapter, we present GAUSS, a synthesis algorithm for dataframe transformations that utilizes a user interface to capture *partial* input-output examples with additional relationship information to aid synthesis. GAUSS records the additional relationships provided by the user as a graph, whose design is similar to the graphs used in AUTOPANDAS. We call this graph the *user intent graph*. As a proof-of-concept, we also create a Jupyter notebook extension offering such a UI that creates a graph under the hood.

GAUSS is an enumerative synthesis algorithm. It employs a novel reasoning procedure over graphs that helps it quickly prune away large classes of infeasible programs. At a high-level, GAUSS adopts a divide-and-conquer approach: it breaks down the user intent graph into smaller subgraph specifications and uses these as a measure of progress while enumerating the search space. Whenever it finds that a class of similar programs does not satisfy these specifications, it detects a core subgraph explaining the root cause of failure. The algorithm then *learns* from this failure by performing inductive reasoning against a knowledge base of example program invocations to rule out other programs in the search space.

Since GAUSS aims to improve over AUTOPANDAS in a specific set of tasks, the core implementation of GAUSS adopts the subset of transformation used in MORPHEUS as it is expressive enough to represent those tasks. Since MORPHEUS targets the R language, we translate the

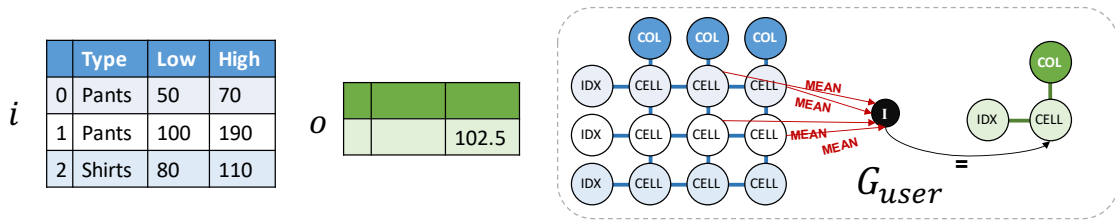


Figure 3.2: An input (i), partial output (o) example, as well as a graph abstraction of user intent (G_{user}).

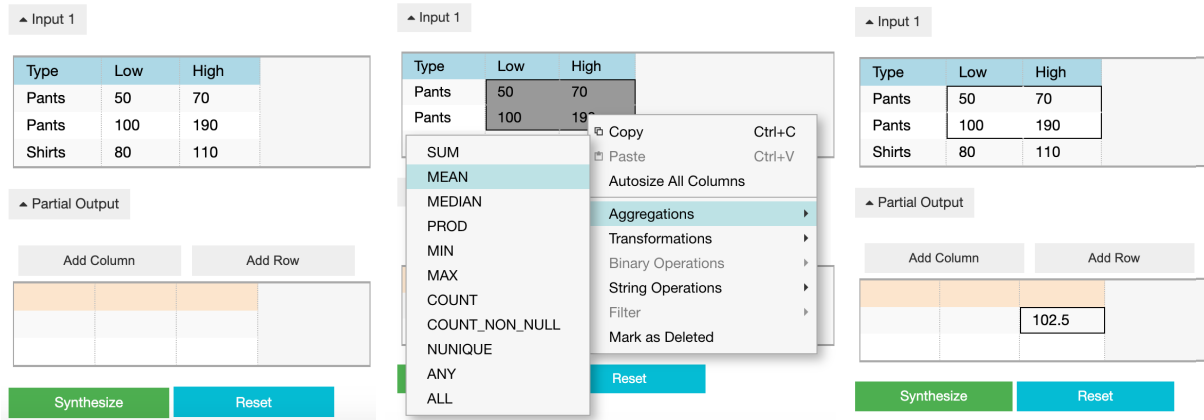
operators into their `pandas` equivalents. We evaluate GAUSS on two main fronts. First, do richer graph specifications enable significant pruning compared to synthesis techniques that only leverage input-output examples? And second, can graph specifications allow the user to provide partial information, such as a partial output, as opposed to needing to construct a full input-output example? Again, since GAUSS extends AUTO-PANDAS on a specific set of tasks, instead of comparing it with AUTO-PANDAS, we compare against three state-of-the-art systems for synthesizing data transformation code from vanilla input-output examples, namely MORPHEUS [34], VISER [125], and NEO[36], on their respective language subsets. We find that GAUSS explores $56\times$, $73\times$, and $664\times$ fewer candidates than the state-of-the-art systems on their respective benchmarks because of the additional graph specification. Additionally, we find that output size can be reduced by $33\times$ on average while still obtaining the correct answer without any noticeable loss in performance.

As before, prior to diving into the technical details behind GAUSS, we walk through an example to illustrate the core ideas behind GAUSS. Note that we use R function names and syntax throughout the discussion as well as the rest of the paper. This includes the use of `gather` instead of `melt`, and `group_by` instead of `groupby` followed by `.agg` where `agg` is a particular aggregation. We do this for a couple of reasons: (1) it eases the discussion with respect to the state-of-the-art baselines in our evaluation, and (2) they are generally more concise than their `pandas` equivalents and thus help with the presentation.

3.1 Overview

We begin with a high-level overview of GAUSS’s algorithm for synthesizing table transformations. Figure 3.1 shows input and output tables describing a table transformation that involves a two-dimensional aggregation — the average of all `Low` and `High` values having the same `Type` category.

Figure 3.2 shows a synthesis specification that a user might supply to GAUSS to synthesize code for this transformation. The specification consists of an input i , a *partial* output o and a graph abstraction of user intent G_{user} . The partial output in Figure 3.2 only contains the cell value 102.5, rather than the full output in Figure 3.1. G_{user} captures the core semantics of the transformation: the value 102.5 in the output is the mean of all the numbers in the `Low`



(a) First, the user loads the input table into the UI. (b) The user selects the group of cells labeled “pants”, right clicks, and selects the aggregation operation “MEAN”. (c) After the aggregated value is copied to the clipboard, the user pastes it into the partial output.

Figure 3.3: A user interaction with the UI that builds the graph abstraction of user intent from Figure 3.2.

and `High` columns with `Pants` in the `Type` column. Note that G_{user} is not provided directly by the user. We provide a user interface (UI) that observes user interaction and automatically creates the graph G_{user} .

Figure 3.3 shows a series of interactions that create the user intent graph on the right hand side of Figure 3.2. First, in Figure 3.3a, the user loads the input dataframe into the UI. Then in Figure 3.3b, the user selects the quadrant of cells with values 50, 70, 100, and 190, and right clicks the selection. This brings up a menu of options, and the user selects the aggregation operation “MEAN”: after clicking this operation, the mean of the selected values is copied to the clipboard. Finally, in Figure 3.3c, the user pastes the value to a cell in the output section of the UI. At this point, note that the input-output example is identical to the partial input-output example given on the left-hand side of Figure 3.2. Behind the scenes, the UI has constructed the graph G_{user} : first constructing the input table part of the graph on load (Figure 3.3a), then adding the intermediate computation and output nodes on paste (Figure 3.3c), thanks to the information provided by the user in their selection in Figure 3.3b.

The goal of GAUSS is to find a program that, when executed on the input table i , produces an output table that *contains* the partial output o provided by the user. Figure 3.4a shows the program synthesized by GAUSS. It first uses a reshaping operation `gather`, that “flattens” the `Low` and `High` columns into a single column (indicated by the arguments ‘`Low`’, ‘`High`’), while keeping the `Type` column as its own column (indicated by the –‘`Type`’ argument). This call to `gather` results in the intermediate output t_1 in Figure 3.4a. Then,

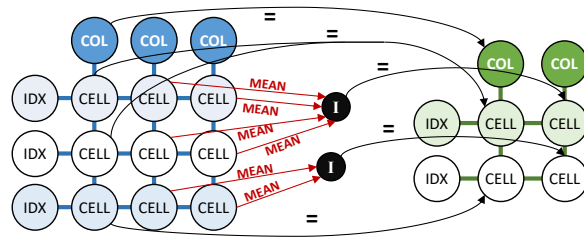
```

t1 = gather(i, "Low", "High", -"Type")
o = group_by(t1, by="Type", Avg=mean("Low"))
    
```

	Type	Var	Value
0	Pants	Low	50
1	Pants	High	70
2	Pants	Low	100
3	Pants	High	190
4	Shirts	Low	80
5	Shirts	High	110

	Type	Avg
0	Pants	102.5
1	Shirts	95

o



(a) First, `gather` melts the input table into a “long” format, where `High` and `Low` are row values rather than columns, producing t_1 . Then, `group_by` can then average all values for each item type, producing o . (b) The final graph abstraction of the solution captures the direct relationship between input and output.

Figure 3.4: The solution program for the synthesis problem in Figure 3.2, its intermediate and final output, and its graph abstraction.

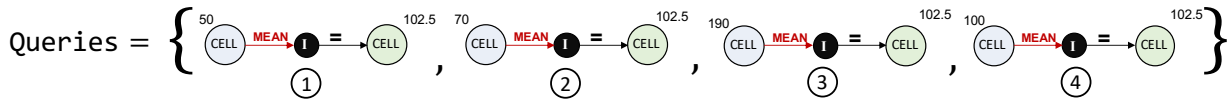
the program performs a `group_by` operation, grouping on the `Type` column to compute the required averages. This results in the final output o in Figure 3.4a.

Additionally, G_{user} must be a subgraph of the *graph abstraction* of the program synthesized by GAUSS when run on the input i . The graph abstraction of a program captures the relationship between its concrete inputs and output as a graph. For instance, Figure 3.4b shows the graph abstraction for the program at the top of Figure 3.4a (when using i from Figure 3.2 as input). This graph abstraction is obtained dynamically by applying a special function to the program and its inputs; the process is described in Section 3.2.2. For the purpose of this section, whenever we say a graph abstraction of a program, we assume the inputs to the program are the same as the input tables i of the user-provided specification.

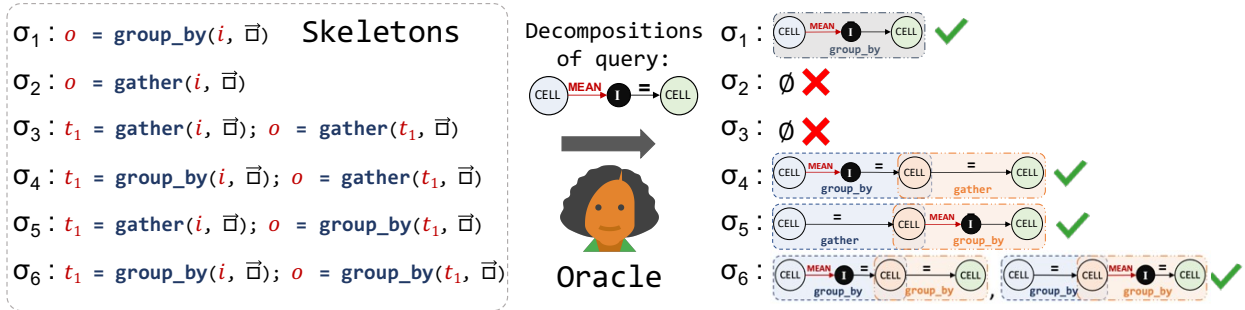
Enforcing that G_{user} is a subgraph of a solution’s graph abstraction ensures that the program matches the user’s intent. Given the spec in Figure 3.2, GAUSS returns the program shown in Figure 3.4.

We will now walk through Figure 3.5, which shows the steps followed by GAUSS to arrive at the solution in Figure 3.4. To synthesize this program, GAUSS employs enumerative search: it enumerates programs one-by-one, runs them, and checks their output against the specification. The key to GAUSS’s performance is its ability to *prune* large parts of the search space of programs without enumerating them.

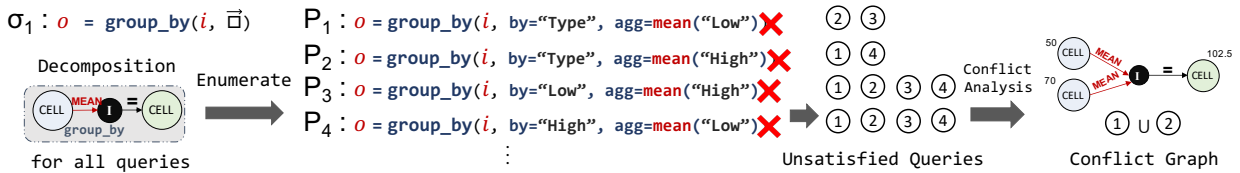
For simplicity in our walkthrough, we assume that we have only two table transformation components, `gather` and `group_by`, and that GAUSS only explores programs containing a maximum of two component calls. Note that every program synthesized by GAUSS is a linear sequence of component calls.



(a) Unit query graphs, or simply *queries*, of G_{user} in Figure 3.2.



(b) For each skeleton, GAUSS uses the oracle to compute decompositions for each query. Because **gather** does not perform aggregation, there are no decompositions for σ_2 and σ_3 . So GAUSS prunes skeletons σ_2, σ_3 .



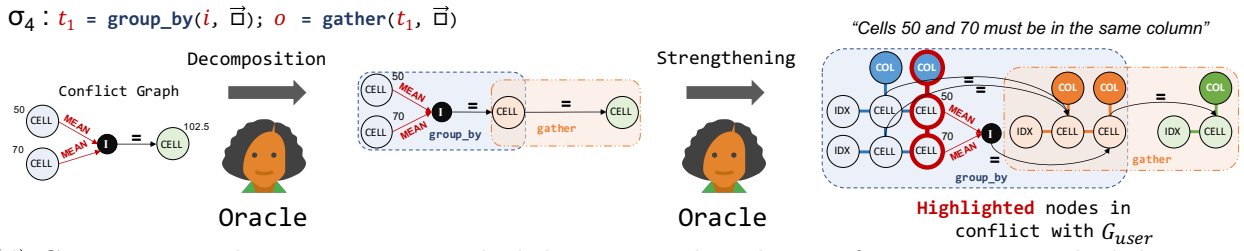
(c) While enumerating arguments for σ_1 , GAUSS finds that no program realizes a decomposition for *all* queries. The smallest set of queries whose decompositions are not simultaneously realized is the *conflict set*.

Figure 3.5: Walkthrough of GAUSS run on the specification in Figure 3.2, with components **gather** and **group_by**.

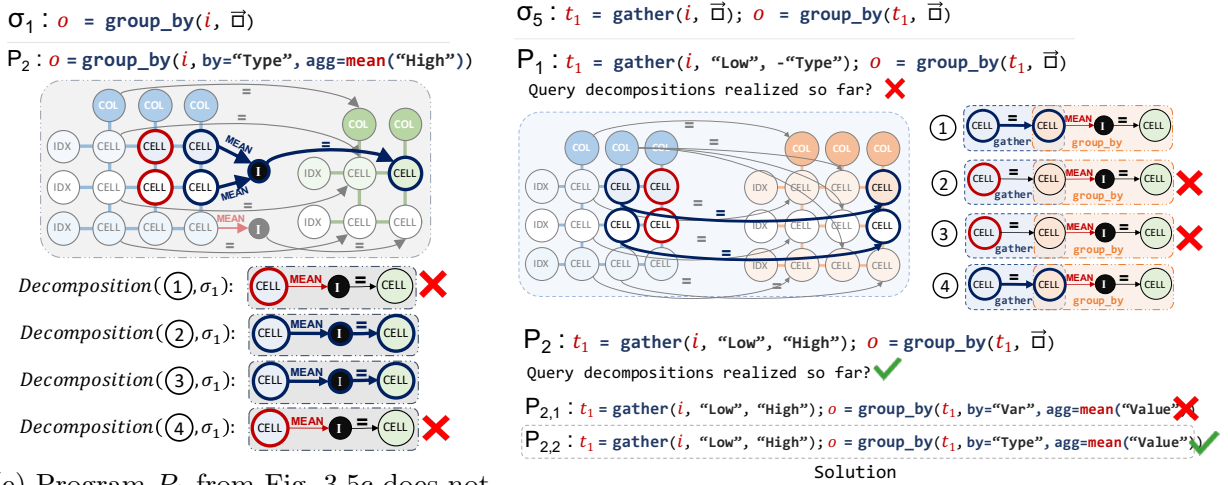
3.1.1 Extracting Query Graphs

To conduct this pruning, GAUSS uses *query graphs*, or simply *queries*, from G_{user} . A **query** is a subgraph of G_{user} containing at least one input and one output node. GAUSS first extracts unit queries that have exactly one input and one output node. Figure 3.5a shows the four unit queries extracted from G_{user} , one for each edge between the “input” and “output” parts of G_{user} . The numbers next to nodes indicate the corresponding table cell.

Observe that if G_{user} is a subgraph of the final graph abstraction G_P of a program, the query graphs from Figure 3.5a *must* be subgraphs of G_P as well. This means GAUSS can reason about the simpler query graphs, rather than the potentially complex G_{user} , to prune programs.



(d) GAUSS proves that no program with skeleton σ_4 is the solution: for a program with skeleton σ_4 to realize the decomposition of the conflict graph, the input cells 50 and 70 must be in the same column.



(e) Program P_2 from Fig. 3.5c does not realize the decompositions of, i.e. follow plans for, queries ① and ④. (f) GAUSS prunes the partial program P_1 because it cannot realize the decompositions of queries ② and ③.

Figure 3.5: Walkthrough of GAUSS run on the specification in Figure 3.2, with components `gather` and `group_by`.

3.1.2 Deciding Skeletons for Exploration

First, GAUSS prunes out *skeletons* which can be safely discarded. A *skeleton* is simply a program with constant arguments unfilled. Thus, it captures only the components (function calls) of the program. Again, for simplicity of exposition, GAUSS is only considering the two components `gather` and `group_by`, so it only has six possible skeletons to explore. These six skeletons are enumerated in Figure 3.5b as $\sigma_1, \dots, \sigma_6$. The symbol $\vec{\square}$ represents unfilled constant arguments.

First, for every skeleton, GAUSS determines all possible *decompositions* of each query graph. At a high level, a *decomposition* of a query graph G_q corresponds to a *plan* that a program P can follow to make sure that G_q is a subgraph of its final graph abstraction.

The right-hand-side of Figure 3.5b shows the decompositions of our query graphs for

each skeleton $\sigma_1, \dots, \sigma_6$. Since all the query graphs in Figure 3.5a are isomorphic, these decompositions are the same for all of them. Consider, for example, the decomposition of query graphs for skeleton σ_5 . It lays out a plan that specifies that (1) the output table of the call to `gather` must contain a cell with the value of the input cell (captured by the edge with the label “=”), and (2) the call to `group_by` should perform aggregation with this cell, captured via the `MEAN` edge to ❶.

The queries have two possible decompositions with respect to the skeleton σ_6 : either of the `group_by` calls perform the aggregation. The skeletons σ_2 and σ_3 do not have any associated decompositions. This is because they only call `gather`, which is a reshaping operation. It cannot aggregate values and thus cannot have a `MEAN` edge in its abstraction.

So, GAUSS discards skeletons σ_2 and σ_3 , meaning it will not enumerate any programs with those underlying skeletons. GAUSS then goes through the remaining skeletons one by one: for each skeleton, it enumerates programs by populating the skeleton’s constant arguments.

3.1.3 Learning from Failures

Figure 3.5c shows GAUSS exploring all programs with underlying skeleton σ_1 . Unfortunately, none of these programs satisfy the specification. This is because they do not follow the “plan”, i.e. **realize** the decomposition for all the query graphs. Figure 3.5e shows this in detail for the program $P_2 = \text{group_by}(i, \text{by}='Type', \text{agg}=\text{mean}('High'))$. Because the graph abstraction for `group_by` has the decomposition graphs for ❷ and ❸ as subgraphs, we say P_2 *realizes* those decompositions. However, the graph does not have the decomposition graphs for queries ❶ and ❹ as subgraphs. So, we say that P_2 satisfies the queries ❷ and ❸, while ❶ and ❹ are unsatisfied.

Similarly, for all the other programs enumerated in Figure 3.5c, some queries remain unsatisfied. To zero in on what went wrong, GAUSS creates the *conflict set*. This conflict set is the smallest possible set of queries such that no program satisfied *all* the queries in the set. As evident in Figure 3.5c, no program satisfies both query ❶ and ❷. So, in our running example, the conflict set contains the queries ❶ and ❷¹. The graph union of the queries in the conflict set is shown on the far right of Figure 3.5c. This union, called the *conflict graph*, is the subgraph of G_{user} where the cells containing 50 and 70 are involved in aggregation.

Before exploring another skeleton, GAUSS makes sure that the same failure will not occur again. Suppose the skeleton σ_4 is the next to be explored. As shown in Figure 3.5d, GAUSS first uses the oracle to get the decomposition of the conflict graph with respect to σ_4 .

Then, GAUSS asks the oracle to *strengthen* this decomposition with respect to the skeleton σ_4 . During strengthening, the oracle uncovers additional nodes and edges that must be present in any program with skeleton σ_4 realizing this decomposition. The right-hand-side of Figure 3.5d shows the strengthened decomposition of the query graph with respect to σ_4 .

¹The set of queries ❸ and ❹ is another viable alternative for the conflict set and would result in the same pruning ability.

The additional nodes and edges impose the condition that the cells with values 50 and 70 must be in the same column.

However, this condition is not satisfied in the user example: the cells with value 50 and 70 are not in the same column in the input i (Figure 3.2). Hence, GAUSS can safely discard the skeleton σ_4 .

3.1.4 Smart Enumeration

After ruling out the skeleton σ_4 (Figure 3.5d), GAUSS moves on to the skeleton σ_5 . This process is illustrated in Figure 3.5f. First, GAUSS fills in the arguments for `gather`, resulting in partial program $P_1 : (t_1 = \text{gather}(i, \text{‘‘Low’’}, -\text{‘‘Type’’}); o = \text{group_by}(t_1, \vec{\square}_2))$. Before exploring further arguments to fill into $\vec{\square}_2$, GAUSS checks whether the program so far is on track to realize the query decompositions.

In particular, GAUSS evaluates the call to `gather`, and computes its graph abstraction. This graph abstraction is on the left-hand side of Figure 3.5f. With this set of arguments, `gather` discards cells corresponding to the `High` column.

However, the decompositions for queries ② and ③, shown on the right of Figure 3.5f, require the nodes highlighted in red to be connected to cells in the intermediate output. This is not the case in the graph abstraction on the left because the corresponding cells were discarded by the call to `gather`!

Thus, the decomposition for queries ② and ③ cannot be realized, regardless of the arguments for `group_by`. That is, any completion of P_1 will not satisfy the queries ② and ③, and thus can be safely pruned by GAUSS even before exploring any arguments for `group_by`.

All these pruning strategies allow GAUSS to quickly explore more promising arguments to σ_5 , before arriving at the solution in the bottom right of Figure 3.5f.

3.2 Technique

In this section, we describe and formalize the technique behind GAUSS.

3.2.1 Preliminaries and Notation

This section establishes common notation used throughout the formal description of the GAUSS algorithm, and may be worth referring back to while reading Sections 3.2.2, 3.2.3, and 3.2.4.

3.2.1.1 Table Transformation Programs

GAUSS synthesizes a linear **table transformation program**, say P . This program takes in a list of input table variables $\vec{\nu}_{in}$ and a program P of length k of the form:

$$(\nu_1 = \mathcal{C}_1(\vec{p}_1, \vec{c}_1); \dots; \nu_k = \mathcal{C}_k(\vec{p}_k, \vec{c}_k)),$$

where:

- each C_i is a table transformation component (e.g. an API function) with a list of table arguments \vec{p}_i and a list of constant arguments \vec{c}_i ,
- each ν_i is a variable representing the table output of $C_i(\vec{p}_i, \vec{c}_i)$,
- each $p_i^j \in \vec{p}_i$ is either an input table variable in $\vec{\nu}_{in}$ or a table variable from the set $\{\nu_1, \dots, \nu_{i-1}\}$.

Let \mathcal{D} be the domain of all such programs. The set of available components, denoted by $Components(\mathcal{D})$, consists of the standard *projection*, *selection* and *cross-product* relational algebra operators along with other operations such as `gather`, `group_by`, `mutate`, `spread` that allow a mix of common reshaping and summarization operations. The domain of constants, $Constants(\mathcal{D})$, consists of the (countably infinite) set of column names, cell values, row indices, etc. GAUSS borrows this set of components from prior work [34], which the reader can refer to for a more detailed discussion.

The **execution trace** of a program P on input tables $t_{in}^{\vec{}}$ is:

$$\langle (C_1, \vec{t}_1, \vec{c}_1, o_1), \dots, (C_k, \vec{t}_k, \vec{c}_k, o_k) \rangle$$

where for each component C_j :

- \vec{t}_j is the vector of tables passed to C_j , and
- $o_j = C_j(\vec{t}_j, \vec{c}_j)$ is the table produced by the execution of $C_j(\vec{t}_j, \vec{c}_j)$.

We denote such a trace as $\tau(P, t_{in}^{\vec{}})$. The output of P is the output of the last component: $P(t_{in}^{\vec{}}) = o_k$.

A program skeleton, or just **skeleton**, is obtained by replacing all *constant* arguments of the program's components with holes. Precisely, a skeleton σ of length k is of the form:

$$\sigma = (\nu_1 = C_1(\vec{p}_1, \vec{\square}_1); \dots; \nu_k = C_k(\vec{p}_k, \vec{\square}_k)).$$

$Programs(\sigma)$ is the set of all programs sharing the skeleton σ and $Skeleton(P)$ is the skeleton of the program P . We use the shorthand $C_i(\sigma)$ to refer to the i^{th} component of σ .

A **partial program** is a partially filled skeleton. That is, a partial program P_{part} with respect to some skeleton σ maps the first d holes of σ to appropriate constant argument vectors, i.e. $\sigma[\vec{\square}_1 \mapsto \vec{c}_1, \dots, \vec{\square}_d \mapsto \vec{c}_d]$.

The **partial execution trace** of P_{part} on input tables $t_{in}^{\vec{}}$, denoted $\tau(P_{part}, t_{in}^{\vec{}})$, is:

$$\langle (C_1, \vec{t}_1, \vec{c}_1, o_1), \dots, (C_d, \vec{t}_d, \vec{c}_d, o_d) \rangle.$$

3.2.1.2 Graphs

A graph G consists of a set of nodes $N(G)$ and edges $E(G)$. Each node $n \in N(G)$ has a label $lbl(n)$ and an entity $entity(n)$. Entities define groups of nodes, where all $n \in N(G)$ with the same $e = entity(n)$ belong to the same group. We use $Entities(G)$ to denote the set of all entities in G , $\{entity(n) \mid n \in N(G)\}$. We use $N(G, x)$ to refer to the set of nodes in G with entity x .

Our edges are directed; we use $src(e)$ and $dst(e)$ to refer to the source and destination of the edge e , respectively. The label of an edge e is denoted $lbl(e)$. We say (n_1, n_2, ℓ) is an edge in G if there exists $e \in E(G)$ such that $src(e) = n_1 \wedge dst(e) = n_2 \wedge lbl(e) = \ell$. As an example, consider the graph in Figure 3.4. Node labels are COL, CELL and IDX while edge labels are = and MEAN. The nodes with the blue color scheme have the same entity i , while those in green have entity o .

We say G_1 is a **subgraph** of G_2 , denoted by $G_1 \subseteq G_2$ if $N(G_1) \subseteq N(G_2) \wedge E(G_1) \subseteq E(G_2)$.

The subgraph **induced** in G by a set of nodes S_N contains only the nodes of G present in S_N and edges with end-points amongst this set of nodes.

The union of two graphs is the graph $G = G_1 \cup G_2$ such that $(N(G) = N(G_1) \cup N(G_2)) \wedge (E(G) = E(G_1) \cup E(G_2))$.

A graph G is a **unit graph** if there is exactly one node in G with entity ent for every entity $ent \in Entities(G)$.

A graph G_1 is **isomorphic** to G_2 , denoted by $G_1 \simeq G_2$, if there exists a bijection $M : N(G_1) \rightarrow N(G_2)$ such that:

- $\forall n \in N(G_1). lbl(n) = lbl(M(n))$,
- $\forall (n_1, n_2) \in N(G_1). (entity(n_1) = entity(n_2)) \iff (entity(M(n_1)) = entity(M(n_2)))$, and
- (n_1, n_2, ℓ) is an edge in $G_1 \iff (M(n_1), M(n_2), \ell)$ is an edge in G_2

That is, there is a mapping between the nodes of G_1 and G_2 that preserves the edge structure along with the edge labels. The mapping also preserves the node labels and groupings. We use $G_1 \simeq_M G_2$ to explicitly specify a mapping M .

A graph G_s is **subgraph isomorphic** to G , denoted by $G_s \subsetneq G$, if there exists an injective mapping $M : N(G_s) \mapsto N(G)$ and a subgraph G'_s of G ($G'_s \subseteq G$) such that $G_s \simeq_M G'_s$.

Throughout the chapter, for ease of notation, whenever we use $G_1 \simeq G_2$ or $G_1 \subsetneq G_2$, we enforce that the isomorphism mapping M has $\forall n \in N(G_1) \cap N(G_2). M(n) = n$. That is, M is the *identity mapping* for the nodes common to G_1 and G_2 .

The * next to nodes in a sequence of graphs $\langle \text{CELL} \xrightarrow{=} \text{CELL}^* \text{ , } \text{CELL}^* \xrightarrow{\text{MEAN}} \text{CELL} \xrightarrow{=} \text{CELL} \rangle$, indicate that the node is *shared* amongst the graphs. We use $*_1, *_2$, etc. to disambiguate

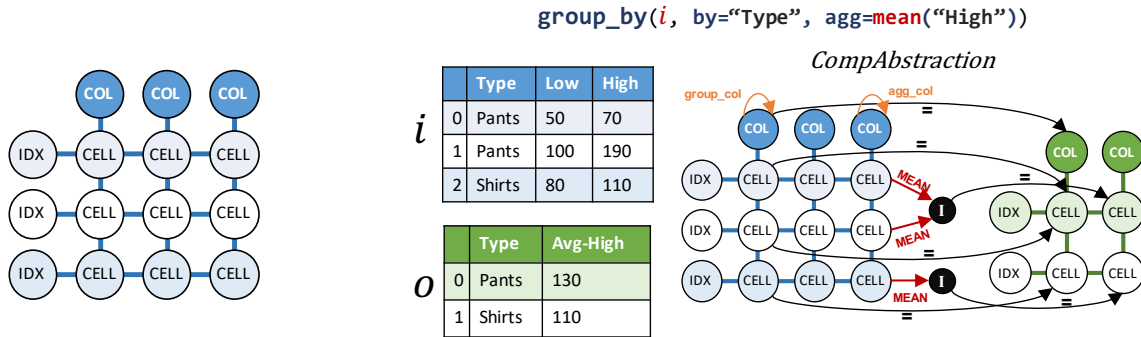


Figure 3.6: Table abstraction for the input in Figure 3.2.

Figure 3.7: Component abstraction of a call to `group_by`. The constant arguments are embedded inside the call.

multiple shared nodes. In figures, we use the notation $\boxed{\text{CELL}} = \boxed{\text{CELL}} \xrightarrow{\text{MEAN}} \boxed{1} \rightarrow \boxed{\text{CELL}}$ to concisely denote sequences (e.g. Fig. 3.5b.).

3.2.2 Graph Abstractions

We now define the concept of the *graph abstraction of a program*, which was alluded to in Section 3.1 but not formally defined. This concept is core to the entire GAUSS algorithm.

Suppose a program P produces an output table t_o when executed on input tables \vec{t}_{in} . The graph abstraction of a program P represents, as a graph, the relationship between (a) the input tables \vec{t}_{in} , (b) the constant arguments \vec{c} embedded in P , and (c) the final output t_o . We denote the graph abstraction as $GraphAbstraction(P, \vec{t}_{in}, t_o)$.

There are two main ingredients required to define *GraphAbstraction*: (1) a *table abstraction* function $TableAbstraction(t)$ that represents table t as a graph and (2) a *component abstraction* function $CompAbstraction(\mathcal{C}, \vec{i}, \vec{c}, o)$ that captures the relationship between input tables \vec{i} and constants \vec{c} and the output o of a single component \mathcal{C} when executed with the inputs as a graph.

Our definition of the **table abstraction** function $TableAbstraction(t)$ returns a graph G_t with a node with label **CELL** for every table cell, a node with label **COL** for every column header and a node with label **IDX** for every row index. There is an edge with label **COLUMN** between the nodes corresponding to a column and every cell in that column. Similarly, there is an edge with label **ROW** between a row node and a cell node for every cell in that row. Thus G_t captures the structure of the table while disregarding the concrete values, just like **AUTOPANDAS**. Additionally, every node in G_t has the table t as the associated entity i.e. $\forall n \in N(G_t). entity(n) = t$. This captures the fact that nodes belong to the group of nodes associated with t . Figure 3.6 illustrates the table abstraction for the input table in Figure 3.2. The vertical and horizontal lines with color (—) depict both the **COLUMN** and **ROW** edges.

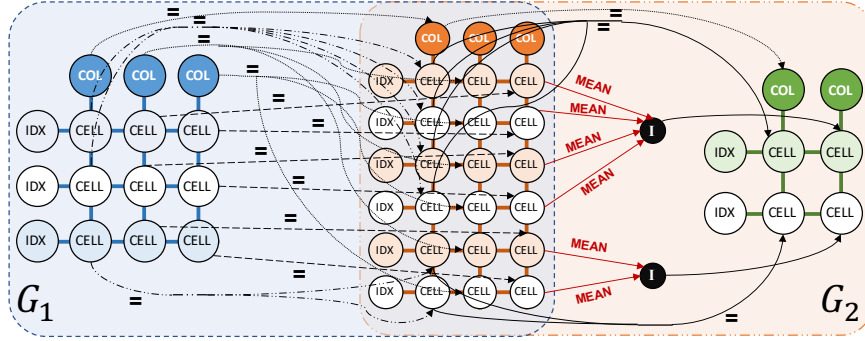


Figure 3.8: Trace Abstraction for Program in Figure 3.4. The nodes and edges in the blue and orange boxes correspond to graphs G_1 and G_2 respectively.

Our definition of the **component abstraction** function $CompAbstraction(\mathcal{C}, \vec{i}, \vec{c}, o)$ returns a graph G which contains the table abstractions of the inputs and output, i.e.,

$$(TableAbstraction(o) \subseteq G \wedge \forall i \in \vec{i}. TableAbstraction(i) \subseteq G),$$

along with nodes and edges that capture the relationship between the inputs and output, i.e., the semantics of the component. Figure 3.7 shows the graph for a call to `group_by` applied on the input table from Figure 3.2. Apart from the nodes and edges corresponding to the table abstractions, the equality edges (labeled “=”) capture the grouping semantics. This corresponds exactly thus far to the graphs in `AUTOPANDAS`. However, this graph contains additional edges such as the `MEAN` edges, which along with the *computation* nodes **I**, capture the aggregation. Additional self edges on the column nodes of the input table capture the interpretation of the constant arguments. As we shall see later in the section, these edges do not play a role in the final graph abstraction but significantly speed up program enumeration during synthesis (Section 3.2.4.3).

We implemented the *TableAbstraction* and *CompAbstraction* procedures for each of our API components—as imperative Python programs, each 50 LoC long on average. Next, we use these procedures to define the abstraction of an execution trace.

Given a trace $\tau(P, \vec{t}_{in}) = \langle (\mathcal{C}_1, \vec{t}_1, \vec{c}_1, o_1), \dots, (\mathcal{C}_k, \vec{t}_k, \vec{c}_k, o_k) \rangle$, the **trace abstraction** is simply the sequence of component abstractions of its constituents:

$$TraceAbstraction(\tau(P, \vec{t}_{in})) = \langle G_1, \dots, G_k \rangle \text{ where } \forall i. G_i = CompAbstraction(\mathcal{C}_i, \vec{t}_i, \vec{c}_i, o_i)$$

. Figure 3.8 shows the trace abstraction $\langle G_1, G_2 \rangle$ of the motivating example’s solution on the input in Figure 3.2. The graphs G_1 and G_2 are the component abstractions of the calls to `gather` and `group_by`, respectively. Note how G_1 and G_2 share certain nodes and edges: this is because they both contain the table abstraction of `gather`’s output.

The trace abstraction captures the relationships between the inputs and output *for each of the constituent component invocations* of the program. However, the graph of user intent

G_{user} captures only relationships between the input and output of the *entire program*. This means that to properly evaluate containment of G_{user} , the final graph abstraction for P must capture relationships between the original input table(s) and final output table directly.

Thus, in the final graph abstraction for a program P , we need to combine the input-output relationships of the individual components into an overall relationship between the input tables of the program and the final output. Consider the union $G_{12} = G_1 \cup G_2$

of graphs G_1 and G_2 in Figure 3.8. The path $\text{CELL} \xrightarrow{=} \text{CELL} \xrightarrow{=} \text{CELL}$ appears in G_{12} . It essentially captures the fact that an input cell's value is equal to the value of a particular cell of the intermediate output table of `gather`, which in turn is equal to a cell of the final output table. We can use the transitivity of equality to conclude that the input cell's value is equal to the value of the final output cell. That is, we can add an edge as in $\text{CELL} \xrightarrow{=} \text{CELL}$.

Similarly, we can simplify the path $\text{CELL} \xrightarrow{=} \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$ in G_{12} to establish that the input cell is directly involved in an averaging operation: $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$.

We formalize this idea of *propagation* of relationships as follows:

Definition 3.2.1 (*PropagatedGraph*($\langle G_1, \dots, G_k \rangle, S_N$)). Let $G_u = G_1 \cup \dots \cup G_k$. The propagation graph of $\langle G_1, \dots, G_k \rangle$ with respect to a set of nodes S_N is a graph G such that

- $N(G) = N(G_u) - S_N$ and
- there is an edge $e \in E(G)$ with $lbl(e) = \ell$ if and only if (a) its end-points are *not* in S_N and (b) there is a path between $src(e)$ and $dst(e)$ through nodes *in* S_N with at most one edge labeled with $\ell \neq "="$ (only one non-equality edge).

Thus, *PropagatedGraph*($\langle \text{CELL} \xrightarrow{\text{DIV}} \mathbf{1} \xrightarrow{=} \text{CELL}^*$, $\text{CELL}^* \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL} \rangle$, $\{\text{CELL}\}$) is the graph $\text{CELL} \xrightarrow{\text{DIV}} \mathbf{1} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$ where the path $\mathbf{1} \xrightarrow{=} \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1}$ leads to the edge $\mathbf{1} \xrightarrow{\text{MEAN}} \mathbf{1}$.

Now, the graph abstraction of a program is simply the propagated graph of the trace abstraction, augmented with the full table abstractions of the inputs and output.

Definition 3.2.2 (*GraphAbstraction*). Let $\langle (\mathcal{C}_1, \vec{t}_1, \vec{c}_1, o_1), \dots, (\mathcal{C}_k, \vec{t}_k, \vec{c}_k, t_o) \rangle$ be the trace of a program P when run on inputs \vec{t}_{in} and output t_o and a corresponding trace abstraction

$\langle G_1, \dots, G_k \rangle$. Then,

$$\begin{aligned}
 \text{GraphAbstraction}(P, \vec{t}_{in}, t_o) &= G_{propagated} \cup G_{table} \\
 \text{where } S_N &= \bigcup_{i=1}^{k-1} N(\text{TableAbstraction}(t_i)) \\
 G_{propagated} &= \text{PropagatedGraph}(\langle G_1, \dots, G_k \rangle, S_N) \\
 G_{table} &= \bigcup_{t \in \vec{t}_{in} \vee t = t_o} \text{TableAbstraction}(t)
 \end{aligned}$$

That is, the graph abstraction contains the abstraction of the inputs and output, as well as the relationships between inputs and output from the propagation graph. The graph abstraction must not contain any intermediate outputs, though it can contain intermediate computation nodes **I**. Hence S_N is the set of nodes belonging to the table abstractions of the intermediate output tables. Figure 3.4 shows the final graph abstraction of the solution program for the motivating example. Note how the user graph in Figure 3.2 is subgraph isomorphic to the graph in Figure 3.4. This observation is at the heart of the problem statement of GAUSS.

3.2.3 Problem Statement

We first formalize the synthesis problem using graph abstractions.

Definition 3.2.3 (Synthesis Problem). Assume a user specification consisting of input tables \vec{t}_{in} , partial output table t_{opart} , and a graph abstraction of the user intent G_{user} (captured automatically via UI). The table abstractions of \vec{t}_{in} and t_{opart} are included in G_{user} i.e. $\forall t \in \vec{t}_{in}. \text{TableAbstraction}(t) \subseteq G_{user}$ and $\text{TableAbstraction}(t_{opart}) \subseteq G_{user}$. The synthesis problem is to find a program P such that:

$$(t_{opart} \text{ is contained in } t_o) \wedge (G_{user} \subseteq \text{GraphAbstraction}(P, \vec{t}_{in}, t_o)) \text{ where } (t_o = P(\vec{t}_{in}))$$

The first clause is in line with a standard problem formulation in example-based synthesis: the output table t_o of program P when executed on \vec{t}_{in} , contains the user-provided partial output table t_{opart} . The second clause enforces a match with the user's *intent*; the graph abstraction of P must be *consistent* with the user-provided graph i.e. G_{user} is *subgraph isomorphic* to G .

3.2.4 Synthesis Algorithm

GAUSS's synthesis algorithm is *enumerative* in nature. That is, it enumerates and checks programs against the specification one by one and stops when it finds a solution or has exhausted all programs. Simply enumerating all programs will be prohibitively expensive as the space of possible programs is very large. The key to GAUSS's performance is how it

exploits the user-provided intent graph to *prune* large parts of this space without explicit enumeration. Next, we provide the intuition for, and formalize the key idea behind, GAUSS’s pruning: graph decompositions.

3.2.4.1 Graph Decompositions

Consider the sequence of graphs $s = \langle \text{CELL} \xrightarrow{=} \text{CELL}^* , \text{CELL}^* \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL} \rangle$. Its propagated graph as per Definition 3.2.1), with respect to the singleton set $\{\text{CELL}\}$, is the graph $G = \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$. We can think of the sequence s as a decomposition of the resulting graph G . Intuitively, s divides the task of aggregation into two parts: first, preserve the value of the cell in another cell and then perform the aggregation on it. We formalize this notion of decomposition below.

Definition 3.2.4 (Graph Decompositions). A sequence of graphs $\langle G_1, \dots, G_k \rangle$ is a *decomposition* of a graph G if there exists a set of nodes $S_N \subseteq N(G_1 \cup \dots \cup G_k)$ such that:

$$G = \text{PropagatedGraph}(\langle G_1, \dots, G_k \rangle, S_N) \cup \bigcup_{x \in \text{Entities}(G)} G[N(G, x)]$$

The second term is a subgraph of G without any *inter-entity* edges: in particular, the union of the subgraphs induced by nodes with the same entity. A decomposition is *minimal* if no edges or nodes can be removed from the constituent graphs without violating the above property.

Combining the problem statement (Definition 3.2.3) and the graph abstraction formulation (Definition 3.2.2) leads us to the following observation, which is core to GAUSS’s pruning strategies.

Observation 3.2.1. *If P solves the synthesis problem $(\vec{t}_{in}, t_{o_{part}}, G_{user})$, then for all subgraphs $G_q \subseteq G_{user}$, including G_{user} itself, there exists a decomposition $\langle G_1, \dots, G_k \rangle$ of G_q such that:*

$$\begin{aligned} & (\forall j \in [1, k]. G_j \subsetneq G'_j) \text{ and } \langle G_1, \dots, G_k \rangle \text{ is minimal.} \\ & \text{where } \langle G'_1, \dots, G'_k \rangle = \text{TraceAbstraction}(\tau(P, \vec{t}_{in})) \end{aligned}$$

Intuitively, $\langle G_1, \dots, G_k \rangle$ can be thought of as a “plan” followed by P to ensure that G_q is present in its overall graph abstraction. This is because each G_i is a subgraph of the corresponding component abstraction G'_i . We say that this decomposition is *realized* by a program P if $\forall j \in [1, k]. G_j \subseteq G'_j$.

Definition 3.2.5 (Realized Decompositions). We say the decomposition $\langle G_1, \dots, G_k \rangle$ is realized by a program P for a given input \vec{t}_{in} , denoted by $\text{Realizes}(\langle G_1, \dots, G_k \rangle, P, \vec{t}_{in})$, if $\forall j \in [1, k]. G_j \subseteq G'_j$ where $\langle G'_1, \dots, G'_k \rangle = \text{TraceAbstraction}(\tau(P, \vec{t}_{in}))$.

Recall the decomposition s at the beginning of this section for $G = \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} = \text{CELL}$ which is a subgraph of G_{user} in Figure 3.2. s is realized by the solution program of the motivating example. The decomposition succinctly captures the plan executed by the program—the `gather` invocation is in charge of preserving the value in its reshaping operation so that the `group_by` invocation can then use it in an averaging operation.

How does Observation 3.2.1 enable pruning? Suppose we want to enumerate and check programs in $Programs(\sigma)$ against the user specification $(\vec{t}_{in}, t_{opart}, G_{user})$, where σ is a skeleton of length k . Given a graph $G_q \subseteq G_{user}$, suppose we have access to a set S_σ of decompositions of G_q satisfying the following property: if there exists a program $P_\sigma \in Programs(\sigma)$ that solves the synthesis problem for the given user spec, then there exists a decomposition in S_σ realized by P_σ . This immediately allows us to implement two straightforward pruning strategies:

1. If S_σ is empty, there does not exist any solution in $Programs(\sigma)$. Thus we can prune the entire family of programs with skeleton σ at one go.
2. If S_σ is non-empty and there is a partial program $P_{partial}$ with the first d constant argument holes of σ filled and the trace abstraction $\langle G_1, \dots, G_d \rangle$, if there is no $\langle G'_1, \dots, G'_k \rangle$ in S_σ such that $G_i = G'_i$ for all $i \in [1, d]$, we can prune all programs in $Programs(P_{partial})$.

One can think of S_σ as a pre-determined set of “plans”, one of which any solution program in $Programs(\sigma)$ must implement. The pruning strategies simply discard programs that clearly diverge from these “plans”. Strategy (1) was motivated in Section 3.1.2, and Strategy (2) in Section 3.1.4.

Before formally developing these pruning strategies, we must first answer two main questions:

1. Which subgraphs $G_q \subseteq G_{user}$ should we use for pruning?
2. Given G_q and a skeleton σ , how do we construct the set of decompositions S_σ ?

How to pick subgraphs $G_q \subseteq G_{user}$?

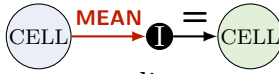
The G_q that will help us prune the search space of programs are ones whose decompositions give us meaningful information. Consider the user graph G_{user} in Figure 3.2. A subgraph G consisting solely of nodes from an input table, like $G = \text{CELL}$, is not useful because its decomposition is the trivial one: $\langle G, K_0, \dots, K_0 \rangle$, where K_0 is the empty graph.

However, subgraphs of G_{user} which relate the input and output, like $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} = \text{CELL}$, will meaningfully decompose and allow us to conduct the pruning steps described above. We formalize this intuition by introducing the concept of *query graphs*:

Definition 3.2.6 (Query). Given a user specification $(t_{in}^{\vec{t}}, t_{o_{part}}, G_{user})$, a *query graph* G_q is a subgraph of G_{user} with at least one node corresponding to every input and the (partial) output, i.e.

$$\forall (t \in t_{in}^{\vec{t}} \vee t = t_{o_{part}}). N(G_q) \cap N(TableAbstraction(t)) \neq \emptyset$$


Additionally, a query must contain at least one path from an input node to an output node. This ensures that queries represent a meaningful fragment of the relationship between input and output.

A *unit query*, like  has *exactly* one node corresponding to every input and *exactly* one node corresponding to output. A *compound query* is simply a non-unit query.

How to determine possible decompositions of G_q given a skeleton σ ?

In constructing S_σ , the set of decompositions of G_q for a skeleton σ , there is a clear tradeoff between the effort spent building S_σ and the pruning power it gives GAUSS. We could, for instance, simply let S_σ be the set of all decompositions for G_q , regardless of the skeleton σ or the user-provided input/partial output. While this is easy to pre-compute, it would not allow GAUSS to do any pruning.

For optimal pruning power, S_σ should only contain decompositions that are realized by a concrete $P \in Programs(\sigma)$ for the current input and output. However computing those decompositions precisely requires enumerating all programs in $Programs(\sigma)$. That is, of course, equivalent to solving the synthesis problem itself and thus does not help in pruning. Instead of either of these extremes, we would like to hit the sweet spot: the decompositions in S_σ are not *unrealizable* and can be computed independently of the synthesis problem.

Consider the decomposition $\langle \text{CELL} \xrightarrow{\text{MEAN}} \text{CELL}^* , \text{CELL}^* \xrightarrow{=} \text{I} \xrightarrow{=} \text{CELL} \rangle$ for the graph . It is a meaningless decomposition as no program in our table transformation domain would ever be able to realize it. This is because of the way we define and implement our component abstractions: in them, a computation edge like MEAN will always end at a computation node I . So, the first element of the decomposition can never appear in a component abstraction.

More concretely, a decomposition $\langle G_1, \dots, G_k \rangle$ is *unrealizable* with respect to a skeleton σ if any graph G_i *cannot* ever occur in the component abstraction of $C_i(\sigma)$ (regardless of the input tables and constant arguments). For now, assume we have an oracle \mathcal{O} that offers a function $Witnessed(G, \mathcal{O}, \mathcal{C})$ that checks this property for us:

Definition 3.2.7 ($Witnessed(G, \mathcal{O}, \mathcal{C})$). $Witnessed(G, \mathcal{O}, \mathcal{C})$ returns true iff there exist inputs \vec{i} and constant arguments \vec{c} s.t. $G \subseteq CompAbstraction(\mathcal{C}, \vec{i}, \vec{c}, o)$ where $o = \mathcal{C}(\vec{i}, \vec{c})$, and false otherwise.

This allows us to formalize the notion of a *unrealizable* decomposition:

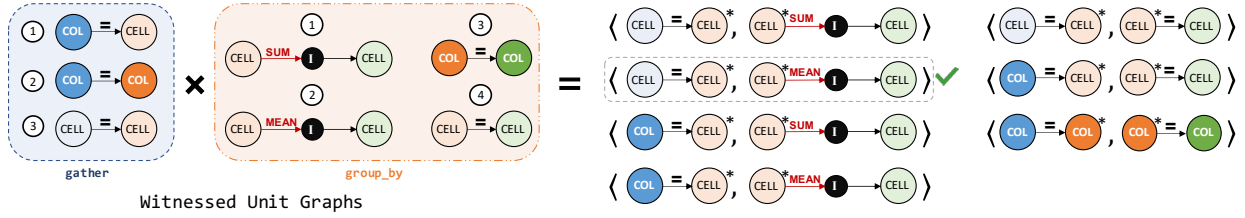


Figure 3.9: Constructing decompositions with respect to skeleton ($\nu_1 = \text{gather}(t_1, \vec{\square}_1)$; $\nu_2 = \text{group_by}(\nu_1, \vec{\square}_2)$)

Definition 3.2.8 (Unrealizable Decomposition). We say that a decomposition $\langle G_1, \dots, G_k \rangle$ of graph G is unrealizable with respect to a skeleton σ if there exists j such that:

$$\neg \text{Witnessed}(G_j, \mathcal{O}, \mathcal{C}_j(\sigma))$$

We define the oracle \mathcal{O} and the details of this *Witnessed* function in Section 3.2.4.6.

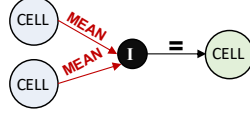
We define the set of possible decompositions of G_q given a skeleton σ as simply the set of all decompositions that are *not* unrealizable with respect to σ . We denote this set as $\text{AllDecompositions}(G_q, \sigma)$ and its construction is described in Algorithm 1.

Algorithm 1 Construction of $\text{AllDecompositions}(G_q, \sigma)$ using oracle \mathcal{O} . Assume σ is of length k .

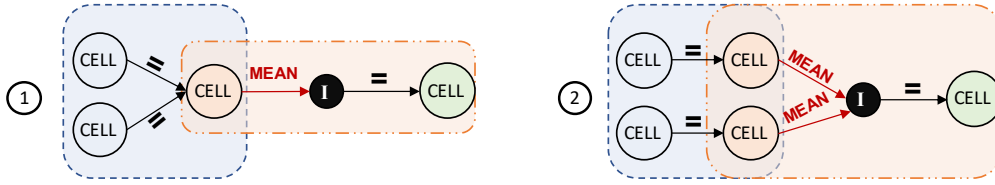
- 1: **procedure** $\text{AllDecompositions}(G_q, \sigma)$
 - 2: **if** G_q is a unit query **then**
 - 3: Construct $\langle S_1, \dots, S_k \rangle$ such that $S_i = \{G \mid G \text{ is a unit graph} \wedge \text{Witnessed}(G, \mathcal{O}, \mathcal{C}_i(\sigma))\}$
 - 4: $D \leftarrow \{\langle G_1, \dots, G_k \rangle \mid \forall i. G_i \in S_i \wedge \langle G_1, \dots, G_k \rangle \text{ is a decomposition of } G_q\}$
 - 5: **else**
 - 6: $D \leftarrow \text{merge } \text{AllDecompositions}(G_u, \sigma)$ for unit queries G_u within G_q
 - 7: **return** D
-

Let us step through the algorithm on an example. Say we want to compute all decompositions of the query graph $G_q = \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$ with respect to skeleton $\sigma = (\nu_1 = \text{gather}(t_1, \vec{\square}_1); \nu_2 = \text{group_by}(\nu_1, \vec{\square}_2))$. Since G_q is a unit query, we first exhaustively enumerate all unit graphs that are witnessed by the components of σ (Line 3). The unit graphs for **gather** and **group_by** are shown in Figure 3.9 in the blue and orange boxes respectively. We then do a combinatorial search for all valid decompositions of G_q (Line 4) by assembling the unit graphs into a sequence. This results in 7 possible sequences, as shown in Figure 3.9. This assembly essentially makes sure that the output nodes of the unit graphs for **gather** align exactly with the input nodes of **group_by**. Only one of these seven is a valid decomposition for G_q and is highlighted in Figure 3.9.

Suppose G_q is a compound query, such as the one below:



To obtain decompositions for G_q with respect to skeleton σ , we first get the decompositions for the constituent unit queries within G_q . There is just the unit query $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{I} \xrightarrow{=} \text{CELL}$. We then search over all “merges” of these unit query decompositions (Line 6). A merge is essentially a component-wise union of the unit query decompositions, but in which different nodes of the constituent unit queries can be merged. This results in two possible decompositions that are shown below.



Armed with this concept of decompositions for query graphs, we are now ready to develop the overall enumerative synthesis algorithm, using decompositions extensively for pruning.

3.2.4.2 Overall Algorithm

Algorithm 2 outlines the GAUSS algorithm. It proceeds as follows. First, GAUSS extracts the set of unit queries from G_{user} in \mathcal{Q} (Line 1). It then prepares a list of skeletons to explore, where each skeleton σ satisfies the property that $AllDecompositions(G_q, \sigma) \neq \emptyset$ for all unit-queries $G_q \in \mathcal{Q}$ (Line 2). This is a direct instantiation of the first pruning strategy discussed in Section 3.2.4.1. Then, the outer loop at Line 3 iterates over the possible skeletons while the inner call to ENUMERATE (Line 5) searches for a solution program with that skeleton.

Before GAUSS enumerates programs with a particular skeleton σ , it calls FEASIBLE (line 4) to perform more checks to determine whether $Programs(\sigma)$ can possibly contain a solution. If not, it *prunes* away the part of the search space corresponding to $Programs(\sigma)$.

Finally, if the call to ENUMERATE fails to find a solution with skeleton σ , GAUSS attempts to identify a *small* subset of queries $\mathcal{Q}_\kappa \subseteq \mathcal{Q}$ (Line 8) that capture the root cause of this failure. The graph union of these queries, G_κ , is a subgraph of G_{user} not contained in the graph abstraction of *any* program in $Programs(\sigma)$. GAUSS then keeps track of this subgraph G_κ to help prune future skeletons earlier (Line 9). A concrete example of this learning was discussed in Section 3.1.3.

3.2.4.3 Enumeration

Algorithm 3 outlines the ENUMERATE procedure, used by GAUSS to populate program arguments. The loop at Line 2 enumerates *partial programs* by *filling* the holes $\vec{\square} = \langle \square_1, \dots, \square_k \rangle$

Algorithm 2 Return a program P satisfying user spec $(\vec{t}_{in}, t_{opart}, G_{user})$, or \perp if no such P exists.

```

  SYNTHESIZE( $\vec{i}, o_{part}, G_{user}$ )
1:  $\mathcal{Q} \leftarrow \text{ExtractUnitQueries}(G_{user})$ 
2:  $\mathcal{S} \leftarrow \{\sigma \mid \text{length of } \sigma \leq \text{MaxLength and } \forall G_q \in \mathcal{Q}. \text{AllDecompositions}(G_q, \sigma) \neq \emptyset\}$ 
3: for each  $\sigma \in \mathcal{S}$  do
4:   if FEASIBLE( $\sigma, \mathcal{Q}, \vec{i}, o_{part}, G_{user}$ ) then
5:      $P, G_\kappa \leftarrow \text{ENUMERATE}(\sigma, \mathcal{Q}, \vec{i}, o_{part}, G_{user})$ 
6:     if  $P \neq \perp$  then
7:       return  $P$ 
8:     else if  $G_\kappa$  is not empty then
9:        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{G_\kappa\}$ 
10: return  $\perp$ 

```

Algorithm 3 For skeleton σ , queries \mathcal{Q} , and spec $(\vec{t}_{in}, t_{opart}, G_{user})$, return solution $P \in \text{Programs}(\sigma)$ if it exists, else the graph union of the smallest set of queries capturing the conflict.

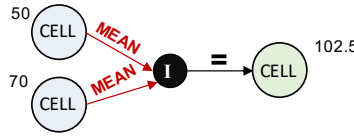
```

  ENUMERATE( $\sigma, \mathcal{Q}, \vec{t}_{in}, t_{opart}, G_{user}$ )
1:  $k \leftarrow \text{length}(\sigma); P \leftarrow \sigma; d \leftarrow 1; \mathcal{F} \leftarrow \emptyset$ 
2: while  $d > 0$  do
3:    $P' \leftarrow \text{FillHoles}(P, d)$ 
4:   if  $P' = \perp$  then
5:      $d \leftarrow d - 1$ 
6:      $\text{Backtrack}(P, d - 1)$ 
7:     continue
8:    $\mathcal{F}_{P'} \leftarrow \{G_q \in \mathcal{Q} \text{ and } P \text{ does not realize any decomp. in } \in \text{AllDecompositions}(G_q, \sigma)\}$ 
9:   if  $\mathcal{F}_{P'} \neq \emptyset$  then ▷ Failure to realize decompositions
10:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{\mathcal{F}_{P'}\}$ 
11:   else if  $d < k$  then ▷ More holes to fill
12:     $d \leftarrow d + 1; P \leftarrow P'$ 
13:   else if  $P'$  satisfies  $(\vec{t}_{in}, t_{opart}, G_{user})$  then ▷ Solution found
14:    return  $P', \emptyset$ 
15:   else
16:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{\emptyset\}$ 
17:  $\mathcal{Q}_\kappa \leftarrow$  smallest subset of  $\mathcal{Q}$  such that  $\forall s \in \mathcal{F}. s \cap \mathcal{Q}_\kappa \neq \emptyset$  and  $\emptyset$  if no such subset exists.
18: return  $\perp, \bigcup_{G_q \in \mathcal{Q}_\kappa}^{graph} G_q$ 

```

one-by-one, via the *FillHoles* function. The variable d captures the *depth* to which the partial program P has been filled. In Lines 8 and 9, we prune any partial program which does not realize any of the available decompositions for a query G_q . This is the second pruning step motivated in Section 3.2.4.1; we gave a concrete example in Section 3.1.4.

If no solution is found, ENUMERATE computes the smallest set of queries, \mathcal{Q}_κ such that no program or partial program explored was able to realize an available decomposition for *at least one* of the queries (Line 17). ENUMERATE returns the graph union of \mathcal{Q}_κ (Line 18), which essentially captures the root cause of failure of enumeration. This is the first step of the learning strategy described in Section 3.1.3. For the skeleton $\sigma = (\nu_1 = \text{group-by}(t_1, \vec{\square}_1); \nu_2 = \text{gather}(\nu_1, \vec{\square}_2))$, this union of conflict queries corresponds to the following subgraph of G_{user} from Figure 3.2:



This graph embodies the fact that no programs for this skeleton were able to involve two cells in the same row (50 and 70) in an aggregation. We use this information in the FEASIBLE function to filter out other skeletons suffering from the same mistake.

3.2.4.4 The FEASIBLE Check

Algorithm 4 Given a skeleton σ , queries \mathcal{Q} , and user-specification $(t_{in}^{\vec{}}, o_{part}, G_{part})$, return **false** if $Programs(\sigma)$ is guaranteed to not contain a solution else **true**.

```

FEASIBLE( $\sigma, \mathcal{Q}, t_{in}^{\vec{}}, t_{o_{part}}, G_{user}$ )
1: for each  $G_q \in \mathcal{Q}$  do
2:   if  $AllDecompositions(G_q, \sigma) = \emptyset$  then return false
3:   for each  $d \in AllDecompositions(G_q, \sigma)$  do
4:     if  $Strengthening(d, \sigma)$  is inconsistent with  $G_{user}$  then
5:       return false
6: return true

```

Algorithm 4 outlines FEASIBLE, used by GAUSS to prune skeletons before enumerating arguments. The first check at Line 2 checks if decompositions are available for every query derived from the user spec G_{user} . Although a similar check is performed in the main algorithm at Line 2 in Algorithm 2, it is repeated because the set of queries can be updated in the main loop of the algorithm. The reasoning behind this check is the same as was outlined for the first pruning step of Section 3.2.4.1.

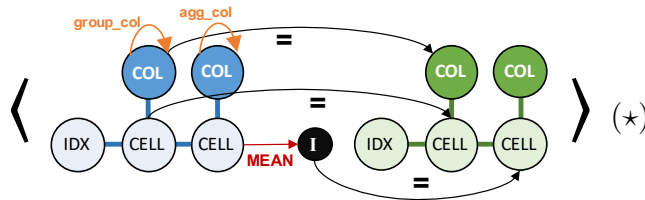
The second check in Line 4 of Algorithm 4 uses the *strengthening* of a decomposition and checks its consistency against the user graph. The next section formalizes the notion

of strengthening with respect to a skeleton. We motivated it in Section 3.1.3: the idea of adding additional nodes and edges to the conflict graph to determine the conditions under which it must occur in the final graph abstraction. At a high-level, FEASIBLE checks in Line 4 if the nodes and edges introduced in the strengthening pertaining to the inputs and final output are consistent with the user-provided graph. If they are not, the skeleton can be skipped.

For example, the strengthening of the decomposition with respect to the skeleton $\sigma = (\nu_1 = \text{group_by}(t_1, \vec{\square}_1); \nu_2 = \text{gather}(\nu_1, \vec{\square}_2))$ of the conflict graph on the left of Figure 3.5d, is shown on the right of Figure 3.5d. This strengthened decomposition captures the fact that cells being aggregated by the first component `group_by` must be in the same column for any program in $Programs(\sigma)$ to produce the correct aggregation. However, this is inconsistent with the user intent graph G_{user} from Figure 3.2 because the two cells corresponding to table values 50 and 70 are in different columns. Thus, FEASIBLE safely concludes that the skeleton cannot contain a solution and returns false.

3.2.4.5 Strengthening Decompositions

We first motivate the idea behind strengthening with a simple example. Consider a skeleton containing a single component `group_by`, i.e. $\sigma = (\nu_1 = \text{group_by}(t_1, \vec{\square}))$. Suppose there exists a program $P \in Programs(\sigma)$ which when executed on some input realizes the decomposition $\langle \text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL} \rangle$ for the query graph $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$. What more can we say about P ? We can prove that P must also realize the following decomposition:



This is because of the properties of tables and the `group_by` function. Specifically, (a) the table structure guarantees that every cell node must have a corresponding column and row node. And (b) every `group_by` operation must have at least one grouping column, and the cell in that column in the same row as the cell being aggregated must be *equal* in value to an output cell in the same row as the cell holding the result of aggregation. Additionally, the columns being grouped and aggregated are designated by self-edges. More formally, we define strengthenings of decompositions as follows:

Definition 3.2.9 (*Strengthening*). A strengthening of a decomposition $d = \langle G_1, \dots, G_k \rangle$ with respect to σ is defined as the decomposition $d' = \langle G'_1, \dots, G'_k \rangle$ such that:

$$\forall P \in Programs(\sigma) \text{ and inputs } \vec{i}. \text{Realizes}(d, P, \vec{i}) \implies \text{Realizes}(d', P, \vec{i})$$

The strengthening operation thus captures additional relationships that must be present between the nodes of the graphs involved in the decomposition. GAUSS exploits strengthened decompositions to more aggressively prune the search space.

Algorithm 5 Fixed-Point Iteration for Strengthening using oracle \mathcal{O} . Assume σ expands to $\nu_1 = \mathcal{C}_1(\vec{p}_1, \vec{\square}_1); \dots; \nu_k = \mathcal{C}_k(\vec{p}_k, \vec{\square}_k)$

STRENGTHENING($\langle G_1, \dots, G_k \rangle, \sigma$)

- 1: $\langle G'_1, \dots, G'_k \rangle \leftarrow$ copy of $\langle G_1, \dots, G_k \rangle$
- 2: **while** $\exists j. G'_j \neq \text{Strengthen}(G'_j, \mathcal{O}, \mathcal{C}_j)$ **do**
- 3: $G'_j \leftarrow \text{Strengthen}(G'_j, \mathcal{O}, \mathcal{C}_j)$
- 4: **for each** (i, ent) such that G'_i shares nodes with entity ent with G'_j **do**
- 5: $G_{ent} \leftarrow$ subgraph induced in G'_j by nodes in G'_i with entity ent
- 6: $G'_i \leftarrow G'_i \cup G_{ent}$
- 7: **return** $\langle G'_1, \dots, G'_k \rangle$

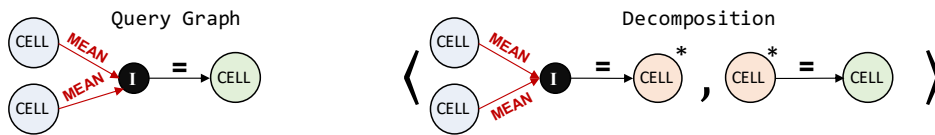
Fixed-point Computation of *Strengthening*

The computation of $\text{Strengthening}(\langle G_1, \dots, G_k \rangle, \sigma)$ relies again on the concept of an oracle \mathcal{O} , which we discussed before.

Definition 3.2.10 ($\text{Strengthen}(G, \mathcal{O}, \mathcal{C})$). $\text{Strengthen}(G, \mathcal{O}, \mathcal{C})$ returns the largest graph G' such that (a) $G \subseteq G'$ (b) for any component abstraction of \mathcal{C} for any arbitrary input, for all isomorphisms G_s of G in the abstraction, there will be an isomorphism G'_s of G' in the abstraction such that $G_s \subseteq G'_s$.

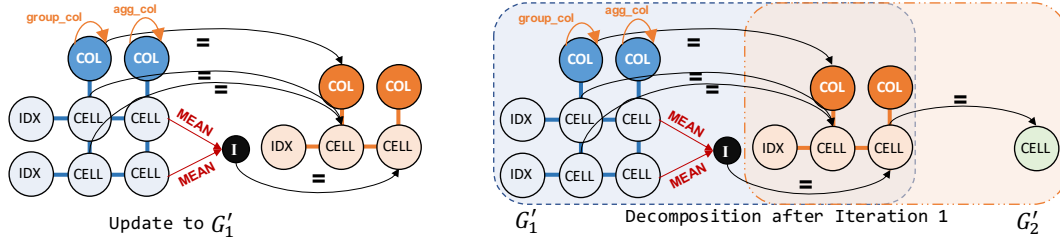
Roughly, the graph result of $\text{Strengthen}(G, \mathcal{O}, \mathcal{C})$ captures additional nodes and edges that must be present for any occurrence of G in a component abstraction of \mathcal{C} for any input. The graph in the strengthened decomposition (\star) above is, in fact, the result of applying Strengthen on $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$ for the component `group_by`.

Algorithm 5 gives the algorithm for computing $\text{Strengthening}(\langle G_1, \dots, G_k \rangle, \sigma)$ for a decomposition $\langle G_1, \dots, G_k \rangle$ with respect to skeleton σ . We walk through the algorithm with the following running example. Suppose, given the query graph below on the left, we want to strengthen its decomposition, on the right, with respect to the skeleton $\sigma = (\nu_1 = \text{group_by}(t_1, \vec{\square}_1); \nu_2 = \text{gather}(\nu_1, \vec{\square}_2))$:

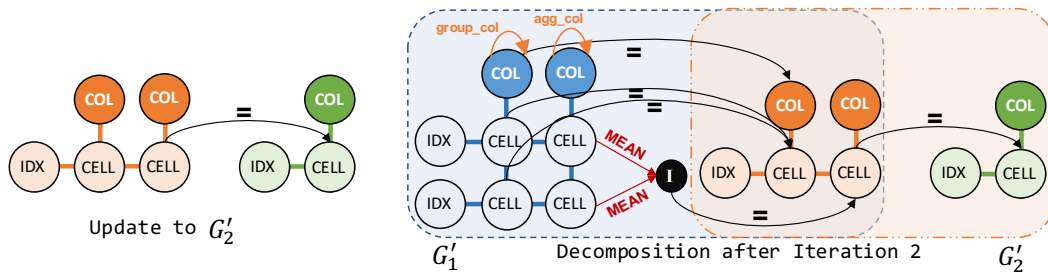


In Lines 2-3, we pick a graph in the decomposition Strengthen . Suppose we pick the first graph (the one for `group_by`). We Strengthen and update it to the graph on the left below. It

captures the property that “Aggregated cells should be in the same column. Their group cells must be in the same row and must be equal”. In Lines 4-6, we update the rest of the graphs in the decomposition with the newly added nodes and edges, if they originally shared nodes with the same entity. We do the update for the graph corresponding to `gather` because the input to `gather` is the output of `group_by`. Hence we add the orange nodes and their edges to obtain the decomposition below on the right:



In the second iteration, we pick the second graph and apply *Strengthen* to obtain the graph on the left below. This one captures the property that all cells have a row and column. This time, we do not need to update the first graph as no new nodes have been added for the input to `gather`. The algorithm finishes, and the final decomposition is shown on the right below:



Use of Strengthening in ENUMERATE Although omitted from Algorithm 3 for brevity, the *FillHoles* internally utilizes the strengthenings of decompositions to reduce the number of possibilities for holes. For example, suppose we have the skeleton $\sigma = (\nu_1 = \text{group_by}(t_1, \text{by} = \square_1, \text{col} = \square_2(\square_3))$ and the query graph $\text{CELL} \xrightarrow{\text{MEAN}} \mathbf{1} \xrightarrow{=} \text{CELL}$ is in \mathcal{Q} . The strengthening of the decomposition of the query graph, shown at the top of this section, contains the self-edges with labels `group_col` and `agg_col`. These self-edges, in turn, help prune the argument space by quickly filtering down the possibilities for the grouping and aggregating columns. Overall, this internal use of strengthening greatly reduces the number of possibilities explored.

3.2.4.6 The Oracle

We now formalize the concept of an *oracle*, used throughout this section to develop GAUSS’s algorithm using the concept of *component examples*.

Component Examples An *example* for a component \mathcal{C} is a tuple $(\mathcal{C}, \vec{i}, \vec{c}, o)$, where $o = \mathcal{C}(\vec{i}, \vec{c})$. Essentially, an example is a set of arbitrary input tables and constant arguments and the component output on those inputs. An oracle \mathcal{O} is simply a set of examples that contains at least one example for each component \mathcal{C} in our table transformation domain.

If \mathcal{O} contains n examples, i.e. $|\mathcal{O}| = n$, we say it is of size n . We use \mathcal{O}_∞ to denote the oracle (of infinite size) containing every possible example. GAUSS's algorithm uses two operations involving the oracle, namely *Witnessed* and *Strengthen*. We slightly modify the original definitions (Definitions 3.2.7 and 3.2.10) to use examples explicitly:

Definition 3.2.11 (*Witnessed*($G, \mathcal{O}, \mathcal{C}$)). *Witnessed*($G, \mathcal{O}, \mathcal{C}$) returns true if there exists an example $(\mathcal{C}, \vec{i}, \vec{c}, o) \in \mathcal{O}$ such that $G \subsetneq \text{CompAbstraction}(\mathcal{C}, \vec{i}, \vec{c}, o)$ and false otherwise.

Definition 3.2.12 (*Strengthen*($G, \mathcal{O}, \mathcal{C}$)). *Strengthen*($G, \mathcal{O}, \mathcal{C}$) returns the largest G' such that (a) $G \subseteq G'$, and (b) for all examples $(\mathcal{C}, \vec{i}, \vec{c}, o) \in \mathcal{O}$, and isomorphisms G_s of $G \subseteq \text{CompAbstraction}(\mathcal{C}, \vec{i}, \vec{c}, o)$, there will be an isomorphism G'_s of G' in $\text{CompAbstraction}(\mathcal{C}, \vec{i}, \vec{c}, o)$ such that $G_s \subseteq G'_s$.

Note that the results of these functions are sensitive to the number of examples used. This relates directly to the issues of soundness and completeness discussed below.

3.2.4.7 Soundness and Completeness

GAUSS's algorithm is trivially sound as it only returns a program if it satisfies the specification. Completeness, on the other hand, depends on the soundness of the pruning strategies. That is, programs pruned must be guaranteed to not solve the specification. The soundness of our pruning strategies has already been discussed in the previous sections, but that discussion assumes that the results of the *Witnessed* and *Strengthen* operations are *correct*, i.e., the oracle used is \mathcal{O}_∞ . But in practice, our oracle \mathcal{O} is finite, containing $|\mathcal{O}| = n$ examples.

Fortunately, we can prove that for a finite set of graphs, there exists a finite oracle such that it is *behaviorally equivalent* to \mathcal{O}_∞ when it comes to the results of the *Witnessed* and *Strengthen* functions. Let us define behavioral equivalence first:

Definition 3.2.13 (Behavioral Equivalence). We say that \mathcal{O}_1 is *behaviorally equivalent* to \mathcal{O}_2 with respect to a (possibly infinite) set of graphs S_G , if $\text{Witnessed}(G, \mathcal{O}_1, \mathcal{C}) = \text{Witnessed}(G, \mathcal{O}_2, \mathcal{C})$ and $\text{Strengthen}(G, \mathcal{O}_1, \mathcal{C}) = \text{Strengthen}(G, \mathcal{O}_2, \mathcal{C})$ for all graphs $G \in S_G$ and for all components \mathcal{C} .

Theorem 3.2.2. *If a set of graphs S_G is finite, there exists \mathcal{O} of finite size $n \in \mathbb{N}$ such that \mathcal{O} is behaviorally equivalent to \mathcal{O}_∞ for S_G .*

Proof. Assuming a finite S_G , we can construct such an \mathcal{O} as follows. Let us consider the case for a single component \mathcal{C} ; we can then repeat this construction procedure for the finite number of other components. For every graph G in S_G , if $\text{Witnessed}(G, \mathcal{O}_\infty, \mathcal{C})$ is true, there is some example in \mathcal{O}_∞ witnessing it. So, we add this example to \mathcal{O} . Since S_G is finite,

say $|S_G| = k$, this adds at most k examples to O . Definition 3.2.12 implies that adding examples to O can only cause a (monotonic) *reduction* in the number of nodes and edges added by $Strengthen(G, O, \mathcal{C})$ to G . Suppose $Strengthen(G, O, \mathcal{C})$ adds n nodes and m edges while $Strengthen(G, O_\infty, \mathcal{C})$ adds n' nodes and m' edges to G respectively. We need to add examples to O to match the behavior; let us pick the examples that cause a reduction of at least 1 in the number of nodes and edges added by $Strengthen$. In the worst case, each example will only remove one node or edge, and we will need to add $(n - n') + (m - m')$ examples to O to get identical behavior on S_G . So, to match O_∞ 's behavior, we have added at most $k + (n - n') + (m - m')$ to O for each component; this proves that O is finite. \square

In our implementation for the domain of table transformations, the set of possible node and edge labels is finite, and we set an upper limit of 2 on the number of inputs for each component. Consequently, the number of possible unit queries is fixed. Further, we only track compound queries at Line 9 in Algorithm 2 if it is obtained by merging *at most* 2 unit queries. Thus the set of graphs for which *Witnessed* and *Strengthen* is invoked is finite, and by Theorem 3.2.2, there exists a finite oracle containing n examples with which we can guarantee completeness. The problem of determining it still remains, however. We determine n empirically by generating random examples till the results of *Witnessed* and *Strengthen* stabilize. We found that 100 random examples for each component were enough for our domain. One can also use a set of benchmark problems with known solutions to determine n , which allows the algorithm to return the correct solution.

3.2.5 User Interface Implementation

We also provide a UI frontend to Gauss that helps users transparently create the intent graph as they interact with the UI to construct a partial output. As shown in Figure 3.3, the user is presented with interactive widgets for the input tables and an empty, editable space for constructing the partial output. The user can simply copy-paste values from the input to the output or use any of the primitive operations exposed by the UI. These operations can be accessed via right-clicking on any arbitrary selection of cells, as shown in Figure 3.3b. Upon selection of the operation, the result is copied into the clipboard, using which the user can paste the value in the partial output. Once the user is satisfied with the partial output provided, they can click **Synthesize** to retrieve solutions from GAUSS. If there are multiple solutions, the user can cycle through them in the UI.

Note that directly editing the partial output is not permitted by the UI. This ensures that values in the partial output are some function of the input cells or columns. If operations need to be chained together, such as taking the ratio of two sums, the UI provides a scratch space to store intermediate computations before pasting the result into the partial output. The publicly available demo at <https://github.com/rbavishi/gauss-oopsla-2021> contains a walkthrough of a problem that can be solved using this scratch space feature.

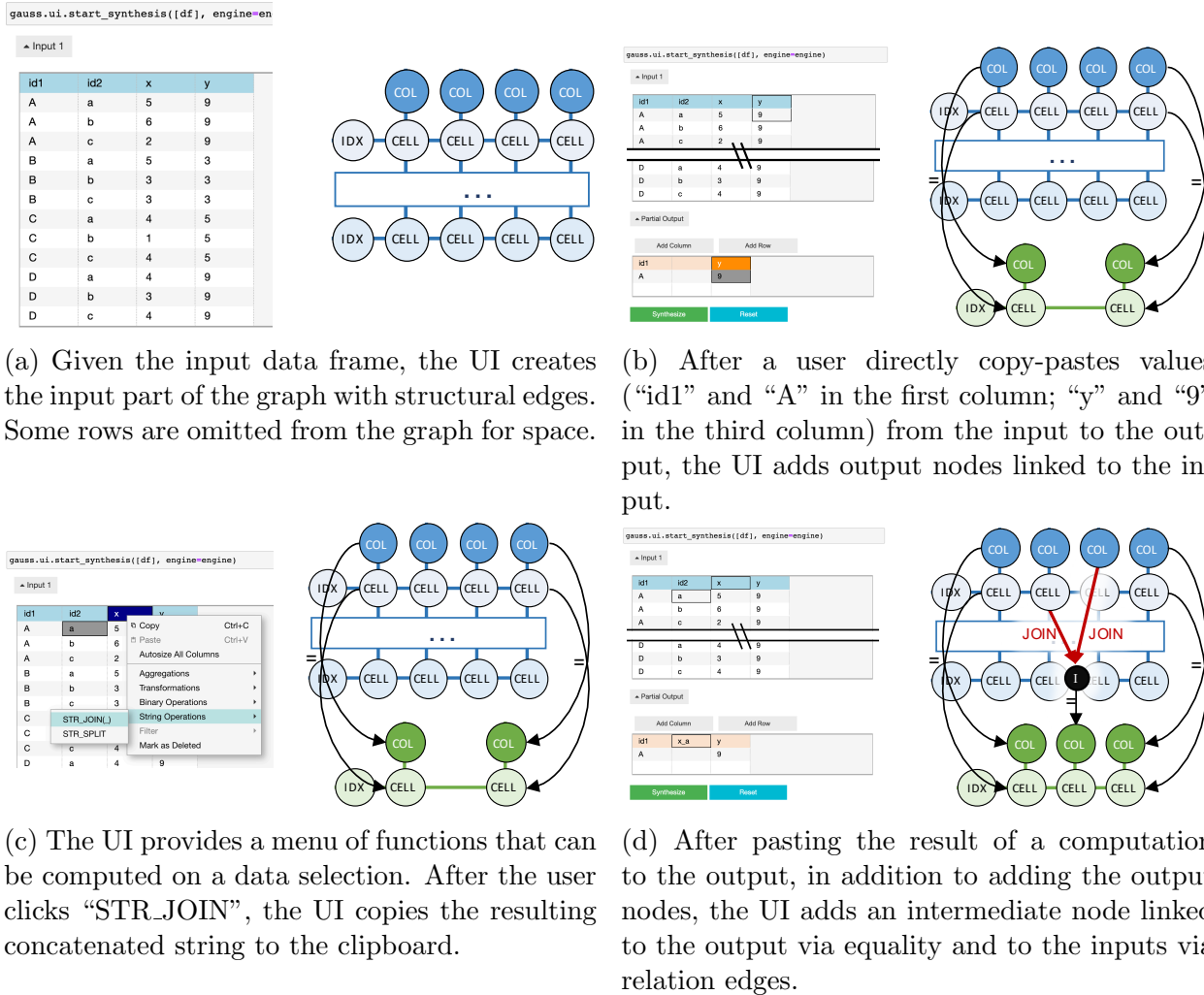


Figure 3.10: Walkthrough of how the UI creates a graph spec. capturing intent as the user constructs the output.

Figure 3.10 shows how the UI records the intent graph when trying to solve the problem posed in the StackOverflow post 62280527². The original dataframe the user provides has two `id` columns and two variable columns (see left of Figure 3.10a). One of the variables, `x`, depends on both `id1` and `id2`, while the second, `y`, only depends on `id1`. The user wants to create a wider table, so that all variables are dependent only on `id1`—by creating new columns that combine the `id2`-dependent variable, `x`, with the different values of `id2`. In Figure 3.10a, the user first loads up the synthesis engine with their input dataframe. At this stage, the UI adds the table abstraction—which captures the structure of the input table, ref. Section 3.2.2—of the input dataframe to its graph specification.

Then, the user copies over a few values that are identical in the input and the output (Figure 3.10b). Specifically, the user copies over (1) the column header “`id1`” and the first value “`A`” of the column to the first column of the output, and (2) the column header “`y`” and the first value in that column, “`9`”, to the third column of the output. After each paste, the UI adds nodes for the new output values and links these to the input nodes via equality edge. The right-hand side of Figure 3.10b shows the graph specification after pasting both (“`id1`”, “`A`”) and (“`y`”, “`9`”) to the output.

Since the user wants new columns that combine `x` with different values of `id2`, they choose “String Operations `>> STR_JOIN(_)`” to concatenate `x` with the first value of the `id2` column. The UI copies the result—“`x_a`”—to the clipboard. When the user pastes the value to the second column header in the output (Figure 3.10d), the UI adds an intermediate node representing the result of the computation to the graph specification. The UI links this intermediate node to the input nodes used in the computation via the `JOIN` relation edges and to the pasted output node via an equality edge.

At this point, the user could paste more values, but GAUSS actually has enough information to synthesize the transformation below:

```
out_1 = gather(input, 'var', 'val', 'x')
out_2 = unite(out_1, 'newvar', 'var', 'id2')
out_3 = spread(out_2, 'newvar', 'val' )
```

This program is equivalent to the accepted answer for the StackOverflow post, which happens to use the newly-added (Sep 2019) `pivot_wider` API function.

3.3 Evaluation

In this section, we present a comprehensive evaluation of GAUSS’s algorithm by answering two primary research questions:

RQ1: What is the upper limit on the pruning power of graph-based reasoning?

Given access to a *total* specification, that is, an input/output example and a user intent graph capturing *all* relationships between the input and output, how does GAUSS compare

²<https://stackoverflow.com/questions/62280527/>

against state-of-the-art pruning-based synthesizers for table transformations that accept the example alone? We evaluate this by measuring the synthesis runtimes and the number of program candidates explored by each tool. As GAUSS can exploit the user intent graph, we expect it to take significantly less time to synthesize a solution and explore far fewer candidates than the baselines.

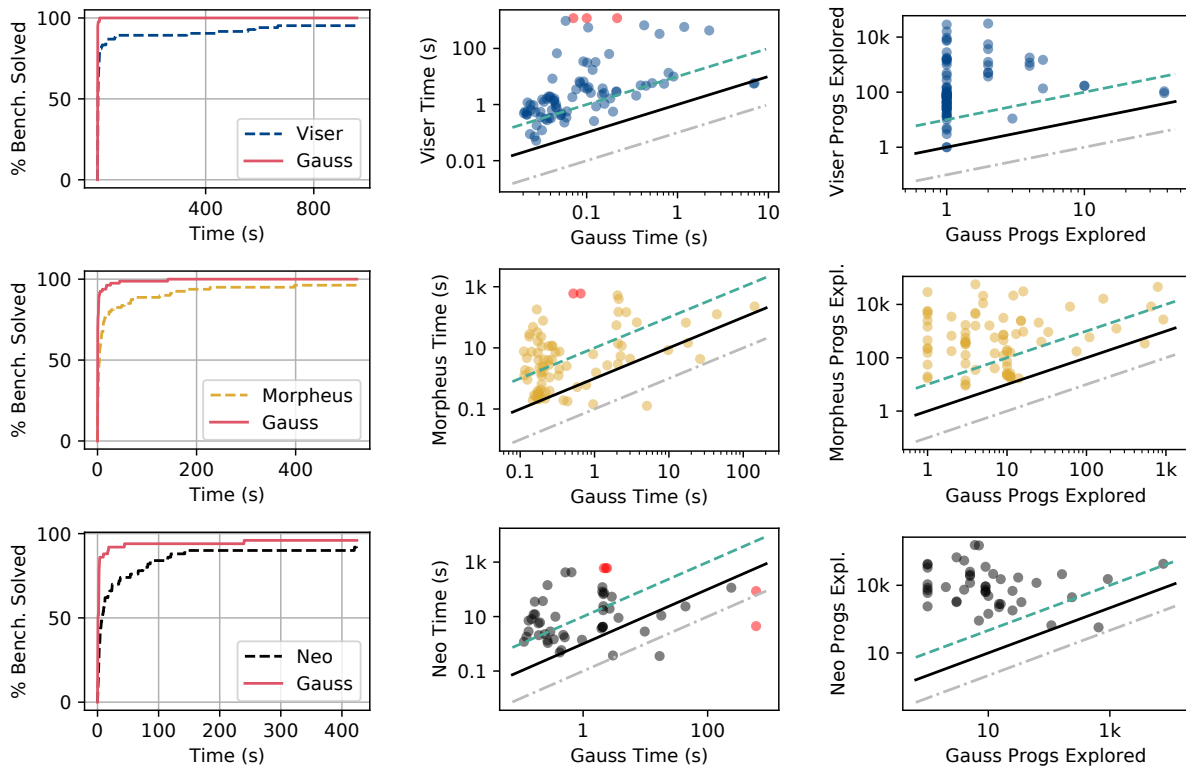
RQ2: Can user intent graphs reduce the size of output specifications? We also evaluate whether user intent graphs can help reduce the burden of specifying a complete output. We minimize the number of output elements—and related user intent graph nodes—in our specifications (Section 3.3.1), until GAUSS can no longer find a solution. This allows us to measure the *achievable* reduction in specification size that user intent graphs enable.

3.3.1 Baselines and Benchmarks and Hardware

We compare GAUSS against three synthesizers: MORPHEUS [34], NEO [34] and VISER [125]. Our benchmark suite contains the 80 benchmarks used in MORPHEUS [34], 50 benchmarks from NEO and 84 benchmarks from VISER [125]. Note that VISER couples the problems of synthesis of table transformation and plotting programs. So, we extract the table outputs inferred during the first step of their algorithm as the output spec for our benchmarks. We also discarded VISER benchmarks that were not solvable by VISER due to the lack of expressiveness of its supported operations. MORPHEUS and NEO benchmarks are *harder* than VISER benchmarks in that their ground-truth solutions use 3-5 API function calls while those for VISER benchmarks use 1-2 function calls. As explained at the beginning of this chapter, we do not directly compare to AUTOPANDAS as GAUSS is meant to augment it for classes of tasks where AUTOPANDAS underperforms and thus covers only a subset of operations.

Since all three tools support slightly different sets of operations and thus cannot solve a portion of each other’s benchmarks, we instantiate and compare GAUSS against them individually. All experiments are performed on a 16-core Intel i9-9900K @ 3.6Ghz machine with 64 GB RAM running Ubuntu 18.04. We use the publicly available implementations for all three tools.

Obtaining Total Specifications for GAUSS A total specification consists of (1) an example containing input and full output tables and (2) a user intent graph capturing all relationships between the input and output. This maps to a scenario where a user uses our UI to provide a complete output table. For our experiments, we obtain the user intent graph programmatically by using the graph abstraction of the ground-truth program. Our graph abstraction does not retain any information about the functions in the program or their arguments and is thus suitable for modeling this scenario.



(a) Solving Time: All. (b) Solving Time: Individual. (c) Progs. explored: Individual.

Figure 3.11: Comparison to VISER, MORPHEUS and NEO. Red dots in (b) indicate timeouts. In (b) and (c), dots above the black line indicate that GAUSS is better, and dots above the teal dotted line indicate that GAUSS is 10x better.

3.3.2 RQ1: Pruning Power

First, we assess the pruning capabilities of graph-based reasoning in GAUSS. We measure this by comparing the synthesis times and the number of programs explored by GAUSS (specification includes user intent graphs) and our baselines (the specification is only input-output examples). We use a timeout of 10 minutes for MORPHEUS and NEO benchmarks and 20 minutes for VISER, both twice the number used in the respective papers. Figure 3.11 shows the results.

Figure 3.11a compares the synthesis times of GAUSS to each of the baselines. The x-axis shows the time budget and the y-axis shows the number of benchmarks that can be solved within that budget. We see that (a) GAUSS is able to solve all 80 MORPHEUS benchmarks with a 2-minute budget while MORPHEUS solves 78 with a 10-minute budget, (b) GAUSS solves 48 out of 50 NEO benchmarks with a budget of 10 minutes, with 47 in under a minute, while NEO solves 45 with a 10-minute budget and (c) GAUSS solves all 84 VISER

benchmarks with a budget of 10 seconds while VISER only solves 81 with a 20-min. budget. Note that we ignore the time VISER spent synthesizing plotting programs.

Figure 3.11b shows a per-benchmark comparison of synthesis time. Dots above the black solid line on the figure are benchmarks where GAUSS is faster than the respective tool, those above the teal dashed lines are benchmarks that indicate at least $10\times$ speedups for GAUSS, and those under the dotted-dashed gray line indicate $10\times$ slowdowns by GAUSS. Red dots along the horizontal and vertical axes represent timeouts for the baselines and GAUSS respectively. We find that GAUSS is faster than MORPHEUS and NEO on most benchmarks (69/80 and 37/50, respectively), and at least $10\times$ faster on 35/80 and 20/50 benchmarks, respectively. This is significant as MORPHEUS is written in C++, and NEO is written in Java, and both parallelize their search by program depth, while GAUSS is a sequential program written entirely in Python. GAUSS is also faster than VISER on all but 2 benchmarks, and over $10\times$ faster on 57 benchmarks.

Figure 3.11c shows a per-benchmark comparison of number of candidates explored. This includes all partial and complete programs encountered during the search. The results reveal the root cause of GAUSS’s performance improvements: the additional graph specifications enable GAUSS to explore significantly fewer candidate programs than all baselines. For many VISER benchmarks, GAUSS only needs to explore one or two candidates before finding a solution. The effect is more pronounced for NEO as its set of benchmarks is the hardest in terms of the size of the solution program. On average across all benchmarks, GAUSS prunes 76% of partial programs encountered in ENUMERATE (Section 3.2.4.3) and 15% of skeletons in FEASIBLE (Section 3.2.4.4).

Overall, the results show that user intent graphs enable orders-of-magnitude reductions in the search space compared to methods using only input-output examples as spec.

3.3.3 RQ2: Reduction in Size of Specifications

Now, we assess whether user intent graphs allow for a substantial reduction in the size of I/O examples provided. In particular, we try to find the smallest output tables (and corresponding user intent graphs) that still allow GAUSS to synthesize the correct programs.

For this experiment, we only consider benchmarks from Viser [125] since it was designed to work on partial input-output examples. For each benchmark, we manually minimized the output and the user intent graph while ensuring that GAUSS still returns the correct solution. In particular, we kept only the output elements (cells, column names, and row indices) which were representative of the transformation. We correspondingly reduced the user intent graph to a graph that includes the full input but only the nodes associated with these output elements. This process enables us to mimic a user inputting only these key output elements in our prototype UI.

Figure 3.12 shows the results with each benchmark as a dot: the x-axis is the number of nodes in the reduced output, and the y-axis is the number of nodes in the full output. Note the *log scale* on both axes. The dots above the green line and grey line indicate benchmarks with $10\times$ and $100\times$ reductions in size. We see that GAUSS synthesizes the correct program

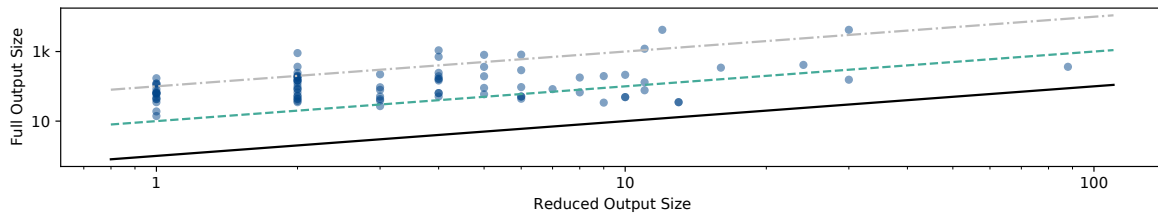


Figure 3.12: Maximal reduction in the number of output nodes such that GAUSS still synthesizes the correct program. Dots above the green line and grey line indicate that reduction is more than 10x and 100x, respectively.

even with orders-of-magnitude less information about the output. On 87% of benchmarks, it finds the correct solution with 10 or fewer output nodes; on 19%, it finds the solution with only a single output node. The impact on runtime is negligible: GAUSS finds the solution $1.2\times$ faster on average with partial outputs.

Overall, we find that the maximal reduction of output size—while retaining GAUSS’s ability to find the solution—is $33\times$ on average. Although the reduction obtained by users in a real deployment of our UI would likely be less, these results suggest that capturing user intent can reduce the burden of output specification.

3.4 Discussion

We elaborate on various topics regarding GAUSS’ design, its limitations, and opportunities.

3.4.1 Necessity of a User Interface

GAUSS returns a solution program if its abstraction contains the user intent graph as a subgraph. This inherently places the restriction that the user intent graphs need to use the same collection of node and edge labels and capture computation in the same way as the graph abstraction. Our use of a UI solves this problem because the UI translates user interaction to a user intent graph. It also frees the user from needing to understand the internals of GAUSS.

3.4.2 Ease of Use

Although we provide a UI in GAUSS, we do not explicitly and formally evaluate its usability. Thus our empirical results, where we programmatically generate the user intent graph, may not reflect real-world usage. For example, although our UI supports the construction of partial outputs in cases where chaining multiple computations is required, such as taking the ratio of two sums—, it may not be easy to use. We feel intelligent UI design that allows

users to input formulas for such cases could help resolve this problem. In a preliminary run-through of the UI, we had two participants solve ten problems each: one was able to solve all 10, while the other participant only solved eight. The experience is discussed more thoroughly in Appendix 3.4.5.

3.4.3 Multiple Possible Representations

There may be multiple possible ways to construct the partial output using our UI. For example, a user could explicitly compute the average by first taking a sum and then dividing it by the count. Our current implementation does not handle this case. This could be handled by either implementing rewrite rules for graphs in the UI or incorporating such alternatives into the component abstractions.

3.4.4 Noise in Demonstrations

Finally, like many synthesis systems, GAUSS is sensitive to noise in the user-provided specifications: GAUSS assumes that the specification is the ground truth and tries to find a program that matches the specification. However, in contrast to other input-output-based systems, GAUSS assumes the output and user intent specification is constructed by the UI. This may reduce the likelihood of users adding certain types of noise (e.g., a typo or error in calculating the mean of some elements). A possible avenue for future work in this space is to identify potentially noisy specifications—especially when the system fails to find a program—and re-run the system with repaired versions of these specifications.

3.4.5 Experience with Real Users

In building our prototype GAUSS UI, we did an informal study of the UI with two computer science graduate students. Both were unaffiliated with the GAUSS project. The first student (henceforth referred to as Participant 1) had some experience using `pandas` to transform tables, while the second student (henceforth referred to as Participant 2) did not.

We constructed a Jupyter notebook with 10 distinct problems for these participants to solve. Each problem contained a natural language description of the desired transformation along with one input-output example to illustrate the transformation. The goal was to find a program performing this transformation. To enable participants to use GAUSS to find the program, we provided a second input that they could use to build an intent-annotated partial input-output example. We spent 15 minutes going over the basic features of the GAUSS UI. Then, for the first 5 problems, we asked the participants to use the UI exclusively to solve the problems: they had to load the second input and construct a partial output that matched the specified transformation. For the remaining 5 problems, we told the participants they could either use the UI or consult any other resources, such as the API documentation or a search engine, to come up with the solution. If a participant took more than 10 minutes to solve a problem, we marked the problem as unsolved by the participant.

Both participants were able to solve the first 5 problems with the UI alone. In fact, Participant 1 solved all 10 problems with the UI, taking 1-3 minutes for each problem, including interaction with the UI, as well as validating the solution. On all 10 problems, they only provided partial outputs before coming to the correct solution. On 4 problems, they had to add more cells to the output—the first partial output they provided did not have enough information for GAUSS to find the correct solution. Overall, though, Participant 1 only provided 24% of the output in their partial output.

Participant 2 timed out on 2 of the last 5 problems—they were unable to solve these problems with the UI and were unable to find external resources (i.e., via a search engine) to solve the problem. On the 8 problems they did solve, they used the UI exclusively, taking 1.5-5.5 minutes for each problem. Unlike Participant 1, Participant 2 provided full outputs on two of the problems.

There was one problem Participant 2 was unable to solve that could be solved in two different ways: (a) by adding three columns or (b) by subtracting a single column from another existing column that contained the total of the remaining columns. Participant 1 took approach (b) and was able to solve the problem. GAUSS, surprisingly, was unable to solve the problem when Participant 2 used approach (a). This led to a number of bug fixes, and currently, GAUSS synthesizes a correct solution for both approaches.

Overall, this experience suggested that the UI approach to constructing user intent graphs was certainly viable. However, there is also evidence that in practice, users may not always provide sufficiently expressive user intent graphs. An in-depth study of this problem will be key to the practicality of a GAUSS-like approach in a deployment setting.

3.5 Summary

In this chapter, we introduced GAUSS, a synthesis system for generating data transformation code that accepts partial input-output examples along with a demonstration of the construction of the output via a specially designed UI. GAUSS includes a novel conflict-resolution reasoning algorithm over graphs that enables it to learn from mistakes made during the search and use that knowledge to explore the space of programs even faster. It also ensures the final program is consistent with the user intent specification, reducing overfitting.

We compare GAUSS to three state-of-the-art synthesizers that accept only input-output examples. We find that it is able to reduce the search space by $56\times$, $73\times$ and $664\times$ on average, resulting in $7\times$, $26\times$ and $7\times$ speedups in synthesis times on average, respectively.

We also conduct an informal user study to gauge the usability of the new UI. Overall, we see promising results on the viability of the UI — the participants were able to solve the majority of the problems, but the partial examples provided were not always sufficiently expressive to get GAUSS to return the right result. There is a research opportunity here in terms of designing an interaction mechanism that can identify the problematic points and ask targeted questions to the user.

Finally, while both `AUTOPANDAS` (Chapter 2) and `GAUSS` target a useful subset of data transformations, there are still a number of wrangling and transformation tasks that are not fully suited to an input-output example or demonstration setting. Consider the task of computing the top ten correlated columns along with the correlation values. It is unreasonable to expect users to provide examples in this setting as computing the values is extremely laborious. Such tasks, however, are arguably simpler and concise to describe in natural language. The two systems `VIZSMITH` and `DATANA`, discussed in Chapters 4 and 5 respectively, precisely tackle this class of problems.

Chapter 4

VizSmith

The last two chapters introduced `AUTOPANDAS` and `GAUSS`, assistants for generating data transformation code from input-output examples, and interactively specified partial input-output examples, respectively. However, not all data processing or visualization tasks are amenable to such a form of specification. Consider the task of visualizing the top-10 correlated columns with respect to a target column as a heatmap. It is difficult, if not infeasible to provide an input-output example for such a task. Even techniques such as `VISER` are inapplicable here as the correlation operation is not supported. Direct manipulation tools for visualizations such as Tableau [115] may also prove to be tricky to use since the data is not structured in a way such that it can be directly mapped to visual channels. In fact, a data join operation needs to be performed using a separate UI [102] in order to make a correlation matrix in Tableau.

Natural language, however, is a convenient specification for such visualization tasks that require the composition of multiple data processing and visualization operations. Being simply able to write “heatmap of top-10 correlated columns with Sales” is a lot more convenient than all the other options presented above. However, code generation from natural language presents an inherently harder problem — unlike the settings of `AUTOPANDAS` and `GAUSS`, candidate programs cannot be automatically evaluated for correctness, as the ambiguity of natural language prevents the design of a mechanized check. Building an *expressive* assistant for generating visualizations from natural language requires a complete rethink of the search space and algorithm. Search becomes more of a ranking problem, and the search space needs to *align* code with natural language, either explicitly in the form of grammars as used in semantic parsing [68, 138], or implicitly as exemplified by machine learning models. How do we capture and represent a diverse space of aligned code and natural language?

In this chapter, we present an approach to automatically mine a corpus of aligned code and natural language and utilize it for the generation of visualizations from natural language. The resulting assistant is called `VIZSMITH`. We leverage the fact that machine learning platforms such as Kaggle [116] host scores of executable data science notebooks that also include the raw dataset. We use program analysis to automatically process these notebooks and mine a knowledge base of *visualization functions*, which are Python functions that take

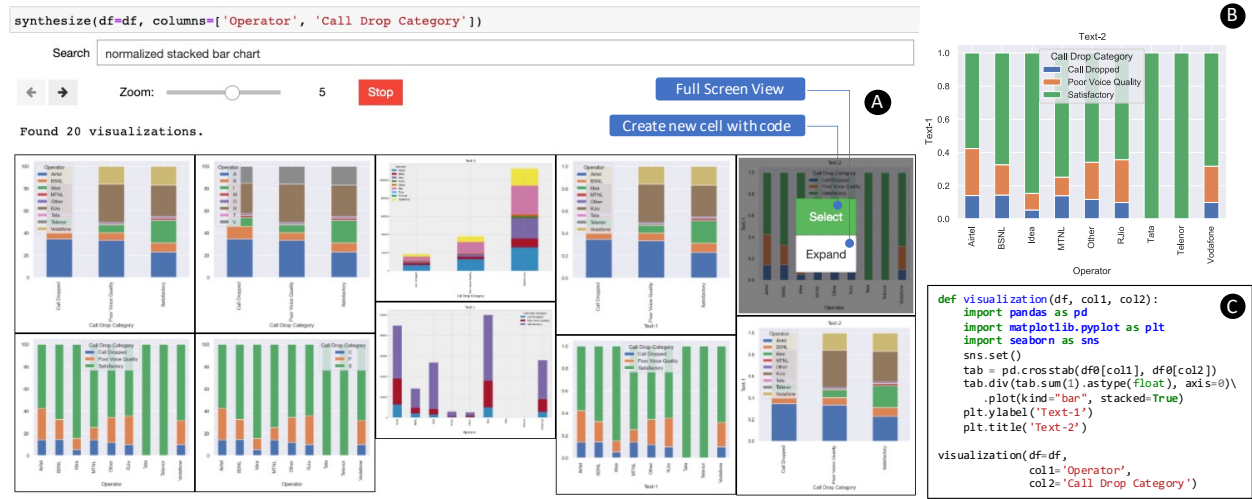


Figure 4.1: VizSMITH’s Jupyter notebook frontend. VizSMITH is provided with a table as a Pandas dataframe along with columns to visualize as input. It has a search bar to input text queries. (A) shows how Alice uses VizSMITH to search for normalized stacked bar charts for her call quality dataset. (B) and (C) show the visualization selected by Alice and its code respectively.

an input table and the set of columns to visualize as input and produce a visualization as output. VizSMITH provides a frontend where users can provide a dataframe and the columns to visualize along with a text query. VizSMITH finds, ranks, and executes the functions best matching the query, and displays the synthesized *bespoke* visualizations.

At the heart of the mining approach inside VizSMITH lies a novel analysis for determining the quality or *reusability* of a mined visualization function. The analysis allows it to discard low-quality code upfront, which greatly helps in improving both the quality and speed of code generation. To the best of our knowledge, VizSMITH represents the first system to provide a precise conceptual definition of *reusability* in the context of visualization code. We also develop a novel approach based on metamorphic testing that approximates this definition for automatically evaluating the reusability of any arbitrary visualization function.

As before, we walk through an example to illustrate the workings of VizSMITH.

4.1 Overview

Alice is a researcher working on a project on analyzing the voice call quality dataset released by the Indian government [121] containing customer ratings. As part of her project, Alice needs to build a visual dashboard that updates every time new data comes in. She has heard about rich data transformation and visualization libraries in Python such as `pandas` and `matplotlib` and decides to use them for this purpose.

In her dashboard, Alice wants to include a visualization of the distribution of customer ratings for every network operator individually, normalized by the number of records for every operator. She decides that a *normalized stacked bar chart* with a bar for every operator would be appropriate for this purpose.

Alice promptly writes code to load the dataset into a `pandas` dataframe. Unsure about how to create a stacked bar chart, she visits the `matplotlib` gallery entry for this chart [110] only to find it insufficient for her needs. She is also uncertain about exactly how to *transform* her dataframe in order to create the bar chart. She turns to StackOverflow for help and browses the results for the query “`matplotlib pandas normalized stacked bar chart`”.

The top result [49] contains a visualization close to what Alice needs, but she has trouble understanding the code, let alone adapting it for her data. This experience is in line with the findings of previous work [134].

Figure 4.1 demonstrates how Alice uses VIZSMITH to find the visualization of her choice along with code to produce it. First, Alice brings up the frontend of VIZSMITH, which is implemented as a Jupyter notebook [61] widget. Alice provides VIZSMITH with her dataframe as well as the columns she wants to visualize. VIZSMITH then presents a search bar where Alice provides the same query as before.

VIZSMITH then consults its knowledge base of *visualization functions* that it has mined from the machine learning notebooks written by data scientists on Kaggle. VIZSMITH utilizes dynamic program analysis and metamorphic testing to construct these functions. These visualization functions are regular Python functions that take a dataframe as an argument along with column arguments and produce a visualization after performing any necessary dataframe transformations. VIZSMITH indexes these functions using the names of the API functions and their keyword arguments, along with the natural language comments found in the Kaggle notebooks. Given Alice’s keywords, VIZSMITH finds the best matching functions, runs them and presents the resulting visualizations in a gallery view as shown in Figure 4.1. VIZSMITH allows Alice to expand a particular visualization to a full-screen view as well as study the code for the visualization.

Alice finds her desired visualization in this list right away, shown in (B) in Figure 4.1. VIZSMITH also produces many similar visualizations with small styling variations. The code for the visualization is shown in (C) and illustrates the inherent complexity of the task as it needs a combination of three `pandas` functions, namely `crosstab`, `div` and `sum` followed by the call to `plot`. Alice copies the code into her workflow, and adjusts the title and y-axis labels.

Thus, VIZSMITH is automatically able to retrieve visualization code available online and, more importantly, automatically *adapt* it to work for the user’s data. This last step is the crucial distinction between VIZSMITH and the usage of search engines to find matching code wherein the developer or analyst is forced to manually adapt it for use in their respective context. VIZSMITH’s corpus is not constrained in any way with respect to the functionality covered, and thus affords VIZSMITH unbounded expressivity.

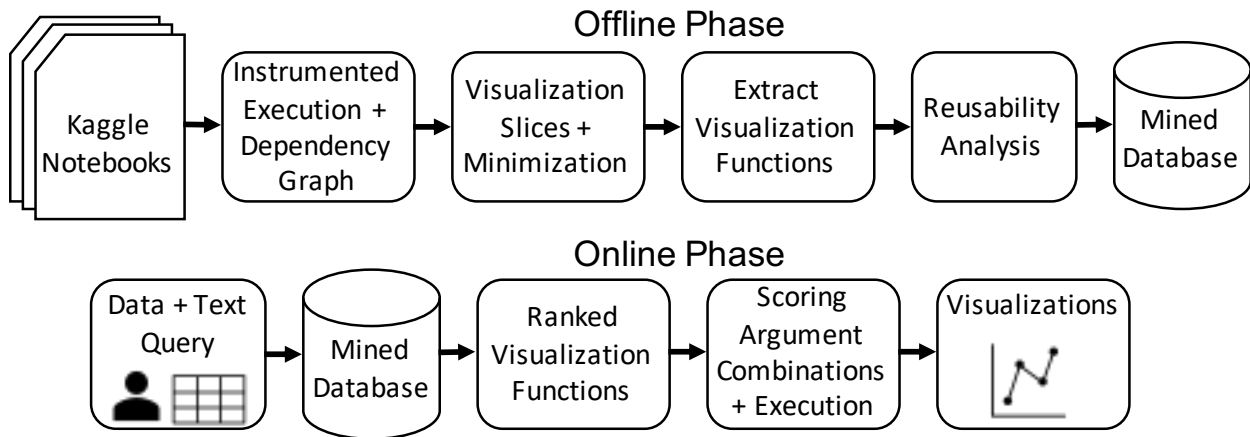


Figure 4.2: Overview of VIZSMITH.

4.2 Technique

In this section, we describe the technique powering VIZSMITH.

4.2.1 Architecture of VIZSMITH

Figure 4.2 presents a high-level overview of VIZSMITH’s architecture. In the offline phase, VIZSMITH collects and mines visualization functions from Python notebooks hosted on the machine learning platform Kaggle [116] (Section 4.2.2). VIZSMITH also analyzes the functions using a novel metamorphic testing scheme (Section 4.2.4) to discard functions ill-suited for code generation. In the online phase, VIZSMITH receives from the user, a dataframe as well as the set of columns in the dataframe that participate in the desired visualization along with a search query. VIZSMITH first uses the query to collect a ranked list of functions to explore (Section 4.2.5.1). Then it finds appropriate arguments to the parameters of each function, and executes them all. Finally, VIZSMITH collects and displays the generated visualizations in a Jupyter notebook user interface.

4.2.2 Mining

We first describe the component of our system responsible for collecting notebooks from Kaggle, replaying them and harvesting visualization code from the notebook runs.

4.2.2.1 Collecting and Replaying Notebooks

We sort the list of competitions on Kaggle by the number of teams participating in the competition. From the top-50 such competitions, we pick those where the dataset corresponds

to a *single* csv/tsv file. We additionally include the `titanic` and `house-prices` competitions as they are the most well-known classification and regression tasks on Kaggle respectively, resulting in a total of 10 competitions (Table 4.2).

Within these competitions, we only select kernels that have an associated Docker image ID which can be downloaded from Kaggle’s GCR repository¹. To conserve resources, we ignore GPU-based kernels and impose a timeout of 10 minutes on each kernel run. For competitions with large datasets (>50k rows), we take a sample of the dataset in order to reduce the execution time.

4.2.2.2 Instrumentation and Execution

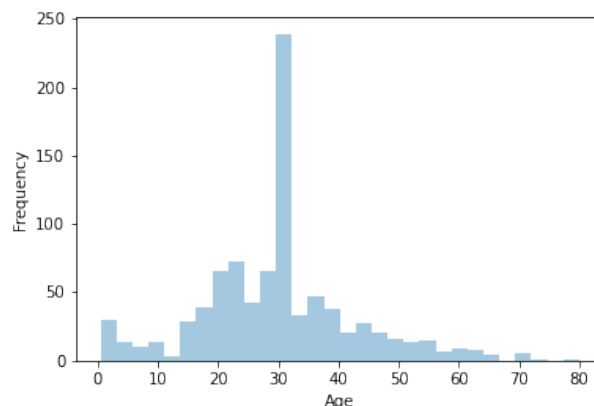
We perform source-level instrumentation of the scripts collected before execution to facilitate the construction of the *dependency graph*. We define the dependency graph of a program P as a graph G such that the nodes correspond to the simple statements in P . A dependency edge exists between nodes n_1 and n_2 if the statement corresponding to n_2 is data-dependent or control-dependent on the statement at n_1 . Data-dependence implies that n_2 uses some variables or data defined or modified at n_1 . Control-dependence means that if n_1 determines whether n_2 executes or not, which is the case when n_1 is an if-statement or a looping statement.

Our source-level instrumentation adds wrapper functions to record essential runtime information such as variable reads and writes, as well as types and memory locations of objects. This information helps us construct the dependency graph. Note that we do not instrument code corresponding to built-in or third-party libraries. Therefore, to capture library dependencies correctly, we construct a separate database of *function specs* with one entry for each built-in and API function. For every function, we determine if it has side-effects, based on the arguments to the function. We write such specs for methods of inbuilt types such as lists, sets and dictionaries as well as API functions from popular data science libraries, namely `pandas`, `matplotlib`, `seaborn`, `numpy` and `scikit-learn`. These specs are quite coarse — given the function call `df.drop(columns=["Low"], inplace=True)`, our spec for `drop` only records that the dataframe `df` is modified, instead of the precise column “Low” that was updated. This keeps our implementation simple at the cost of spurious dependency edges.

4.2.2.3 Visualization Objects and Visualization Slices

Over the course of execution of a program P , we collect the Python objects corresponding to individual visualizations. In our implementation, we focus on the `matplotlib` library as well as its wrapper library `seaborn`, so we track all unique Python objects of the type `matplotlib.pyplot.Figure`. We call such an object a *visualization object*, or simply *visualization*. We say that visualizations ν_1 and ν_2 are the same if the corresponding images obtained after serialization/rendering are a pixel-by-pixel match. For `matplotlib`, this corresponds to the output of the `matplotlib.pyplot.Figure.savefig` API function.

¹<https://gcr.io/kaggle-images/python>



```

1 import pandas as pd
2 import seaborn as sns
3 sns.set_style('white')
4 df_train = pd.read_csv("../input/train.csv")
5 df_train.fillna(df_train.mean(), inplace=True)
6 df = df_train[['Age']]
7 ax = sns.distplot(df['Age'], kde=False)
8 ax.set(xlabel='Age', ylabel='Frequency')

```

Figure 4.3: Example of a visualization and a corresponding slice extracted from Kaggle.

For every visualization ν seen over the execution of program P , we construct a *visualization slice* defined as follows:

Definition 4.2.1 (Visualization Slice). We define the visualization slice of a program P with respect to a visualization object ν , denoted as $VizSlice(P, \nu)$, as a program P' that can be obtained by removing statements from P such that, when executed, P' produces the same visualization ν and *only* ν .

Thus, a visualization slice contains all the statements in a program necessary for recreating a particular visualization. Figure 4.3 contains a slice of the linked Kaggle notebook for the shown visualization. Furthermore, the slice should only produce a single visualization.

We use standard dynamic program slicing [3] to obtain a visualization slice. Specifically, we remove all statements in P that are *not reachable* via a backward-traversal of the dependency graph of P starting from any of the statements in the set $VizStmts(P, \nu)$ defined below:

Definition 4.2.2 ($VizStmts(P, \nu)$). $VizStmts(P, \nu)$ is the set of all statements in the program P that *directly* create/modify the visualization object ν .

In Figure 4.3, the statements in lines 5-6 correspond to the set returned by $VizStmts$ for the program corresponding to the parent notebook and the visualization object being the

```

1 import pandas as pd
2 import seaborn as sns
3 df_train = pd.read_csv("../input/train.csv")
4 df_train.fillna(df_train.mean(), inplace=True)
5 ax = sns.distplot(df_train['Age'], kde=False)
6 ax.set(xlabel='Age', ylabel='Frequency')

```

Figure 4.4: Minimized version of visualization slice in Figure 4.3.

actual plot at the top of Figure 4.3. The first creates the distribution plot, while the second sets the labels of the axes. The remaining statements in Figure 4.3 modify the style, load the dataframe and modify it before visualization and are hence included in the slice.

4.2.2.4 Minimizing Visualization Slices

Recall that our dependency graph construction is not precise as we use coarse specifications for third-party libraries. As a result, the visualization slice obtained via dynamic program slicing may still contain irrelevant statements whose removal will not affect the visualization. Consider the slice in Figure 4.3. The call to `set_style` in line 3 is unnecessary as the style is "white" by default. It is included in the slice because it writes to an internal styling dictionary which is then read in the call to `distplot` thereby establishing a dependency. Taking the subset of columns in line 6 is also unnecessary as `distplot` only receives the target column anyway. We can remove both these operations to yield a simpler, *minimized* visualization slice, as shown in Figure 4.4.

How do we obtain the minimized visualization slice in Figure 4.4 from the slice in Figure 4.3? Note that it is not enough to simply remove or delete code as one might do if they were using delta-debugging [140]; removing lines 3 and 6 in Figure 4.3 would lead to an undefined variable error for `df`. Essentially, we need transformations that go beyond code removal.

We instantiate the generalized syntax-guided program reduction framework developed in PERSES[113] to enable this minimization. In particular, we use standard statement-level delta-debugging to remove top-level statements whose removal does not change the generated visualization. Additionally, we use a transformation where we replace a usage of a variable holding a dataframe with the usage of a previously defined variable, also holding a dataframe. We keep alternating between these two transformations until the slice cannot be minimized further without altering the visualization. Alternation helps here because one transformation may introduce minimization opportunities for another. Algorithm 6 describes this procedure.

We walk through how the algorithm minimizes the slice in Figure 4.3. In the first iteration, delta-debugging (line 5) would remove the call to `set_style` in line 3, Figure 4.3. Then we iterate over variants returned by `DFVARREPLACE`. `DFVARREPLACE` replaces a use of a variable holding a dataframe by a use of another previously defined variable holding a

Algorithm 6 Minimization Algorithm Pseudocode

```

1: function MINIMIZE( $P_\nu$ )
2:   current  $\leftarrow P_\nu$  ; change  $\leftarrow$  true
3:   while change is true do
4:     change  $\leftarrow$  false
5:     variant  $\leftarrow$  DELTADeBUG(current)
6:     if variant  $\neq$  current then
7:       current  $\leftarrow$  variant; change  $\leftarrow$  true
8:     for variant in DFVARREPLACE(current) do
9:       if variant produces same visualization then
10:        current  $\leftarrow$  variant; change  $\leftarrow$  true
11:      break
12:    $P_\nu^{min} \leftarrow$  current
13:   return  $P_\nu^{min}$ 

```

different dataframe. If there are many possibilities, *DfVarReplace* explores variants in the descending order of the *gap* between the original and replacing definitions of the variables, measured in the number of statements. The variant where `df` is replaced with `df.train` is retained. Then line 6 in Figure 4.3 gets removed in the second iteration, and the algorithm exits after the third iteration as no further minimization occurred, successfully returning the desired slice in Figure 4.4.

The reasons behind selecting these two transformations are two-fold. First, data-science code has minimal control flow. Hence, focusing on top-level statements is sufficient. Secondly, data-transformation logic almost always involves applying API functions on variables holding the data (dataframes). Since visualization slices can be slow to execute as they use heavyweight libraries, our restricted set of transformations strike a balance between scalability and quality of minimization.

4.2.3 Extracting Visualization Functions

In this section, we describe how VIZSMITH creates *visualization functions* from a visualization slice. Visualization functions form the basic unit of VIZSMITH’s mined database which it uses for code generation. Throughout this section, whenever we refer to a visualization slice, we assume it is minimized.

A visualization function is formally defined as follows:

Definition 4.2.3 (Visualization Functions). A visualization function f is a Python function with a single dataframe parameter df and m column parameters col_1, \dots, col_m that produces a visualization.

Note that while the above definition restricts a visualization function to a single dataframe parameter, our technique has no such inherent restriction. We adopt this definition to

```

1 import pandas as pd
2 import seaborn as sns
3 df_train = pd.read_csv("../input/train.csv")
4 df_train.fillna(df_train.mean(), inplace=True)
5 ax = sns.distplot(df_train['Age'], kde=False)
6 ax.set(xlabel='Age', ylabel='Frequency')

```

Figure 4.5: Dependencies between top-level statements for code in Figure 4.3. Edges labeled 1, 3, 4, 5 and 6 capture dependency between the use and definition of a variable (`df_train`, `df_train`, `ax`, `sns` and `pd` respectively) while 2 captures the dependency between attribute reads and writes of an object (the dataframe in `df_train`).

```

1 def visualization(df, col1):
2     import seaborn as sns
3     df.fillna(df.mean(), inplace=True)
4     ax = sns.distplot(df[col1], kde=False)
5     ax.set(xlabel=col1, ylabel='Frequency')

```

(a) Visualization function using `b=3` and variable as `df_train`.

```

1 def visualization(df, col1):
2     import seaborn as sns
3     ax = sns.distplot(df[col1], kde=False)
4     ax.set(xlabel=col1, ylabel='Frequency')

```

(b) Visualization function using `b=4` and variable as `df_train`.

Figure 4.6: Visualization functions extracted from slice in Figure 4.3.

simplify the discussion and the notation used throughout the chapter.

At a high level, visualization functions can be extracted from a visualization slice by converting variables holding references to dataframes into parameters and abstracting concrete references to columns into column parameters. The body of the function contains only the statements from the slice required to reproduce its visualization given the new dataframe argument. Figure 4.6 shows two visualization functions from the visualization slice in Figure 4.3. Each of them has a single column parameter `col1`. Both produce a visualization containing the distribution plot of the supplied column, with the function in Figure 4.6a performing an extra imputation step to replace missing values by the mean of their respective columns. We call the slice P_ν from which a visualization function f is obtained as the *parent slice* of f .

Algorithm 7 formalizes the idea. Given a visualization slice P_ν producing visualization

ν and its dependency graph G , for every program point b between the top-level statements of the slice (line 4), and every variable var holding a reference to a dataframe object val_{df} that is in scope at b (line 6), we extract a visualization function as follows. We set the body of the function to be a subset of the statements in P_ν , with the variable var renamed to df (the dataframe parameter). This subset is the smallest such that if the function is executed with the initial value of df as val_{df} in the exact same state it was at program point b in the slice, the resulting visualization is the same as ν . This subset is obtained using backward slicing (lines 7-10), but on a subgraph G_r of G . G_r has the same set of nodes as G , but does not contain any dependency edges in G that originate before the boundary and that arise because of the use of the variable var or the dataframe val_{df} . This helps us pick only the statements necessary to reproduce visualization ν if var is already assigned to val_{df} to begin with.

Algorithm 7 Extracting Visualization Functions

GETVARDFS(P_ν, b) returns the set of dataframe variables in scope at program point b in P_ν along with their values. ISDATAFRAMEEDGE(e, var, df_{var}) returns true if the edge e is a data-dependency edge resulting from the use of variable var or dataframe val_{df} . REACHABLE($s_i, G_r, root$) checks if s_i is reachable from root via a backwards traversal of G_r .

- 1: **function** EXTRACTVIZFUNCTIONS(P_ν, ν, G)
- 2: $\langle s_1, \dots, s_k \rangle \leftarrow$ top-level statements in P_ν
- 3: funcs $\leftarrow \emptyset$
- 4: **for each** program point $b \in [1, k]$ **do**
- 5: $S_b \leftarrow \{s_1, \dots, s_b\}$
- 6: **for each** $(var, df_{var}) \in$ GETVARDFS(P_ν, b) **do**
- 7: $E_r \leftarrow \{e \mid e \in \text{EDGES}(G) \wedge \text{SRC}(e) \in S_b$
 $\wedge \text{ISDATAFRAMEEDGE}(e, var, df_{var})\}$
- 8: $G_r \leftarrow$ induced subgraph of G by removing edges in E_r
- 9: root \leftarrow VIZSTMTS(P_ν, ν)
- 10: body $\leftarrow \{s_i \mid s_i \in \{s_1, \dots, s_k\}$
 $\wedge \text{REACHABLE}(s_i, G_r, \text{root})\}$
- 11: $S_{\text{forbid}} \leftarrow \{s \mid s \in S_b \wedge var \text{ is used in } s\}$
- 12: **if** $S_{\text{forbid}} \cap \text{body} = \emptyset$ **then**
- 13: $f.\text{df_param} \leftarrow df$
- 14: $f.\text{body} \leftarrow$ RENAMEVAR(body, var, df)
- 15: $f.\text{col_params}, f.\text{body} \leftarrow$ INFERCOLPARAMS(f, df_{var})
- 16: **if** VERIFY(f) **then**
- 17: funcs \leftarrow funcs $\cup \{f\}$
- 18: **return** funcs

For example, suppose $b=3$ and $var=df_train$ and the slice under consideration is the one in Figure 4.3. The graph G_r would *not* contain the edges ① and ③ in Figure 4.5 as they originate right after the statement at line 3 (before b), and arise due to the use of the dataframe variable `df_train`. The edge ② is *included* as it originates *after* b .

```

1  def visualization(df):
2      import seaborn as sns
3      sns.heatmap(df.corr())

```

Figure 4.7: A visualization function taking no arguments.

Lines 11-12 confirm that the selected statements which appear before the selected program point b do not involve the use of variable var . This prevents any dependency on a possibly *stale* version of val_{df} . Then, we infer column parameters by simply replacing all string constants that correspond to a column name in val_{df} with parameter variables (line 15). In Figure 4.3, this corresponds to the string "Age" in lines 5 and 6. We also rewrite attribute based column-accesses of dataframes, such as `df.Column` as `df["Column"]` prior to applying this procedure. We denote the mapping from these column parameters to the string constants as $\text{ORIGCOLS}(f)$. We refer to the selection of val_{df} as $\text{ORIGDF}(f)$.

Finally, in line 16, we verify if running the visualization function with val_{df} , that is, $\text{ORIGDFs}(f)$ and $\text{ORIGCOLS}(f)$ reproduces the visualization from the parent visualization slice. Figure 4.6 contains the two visualization functions extracted from the visualization slice in Figure 4.3. Observe that no choices for a dataframe variable would be available if we pick the program point b as either 1 or 2.

In this way we are able to obtain 9740 visualization functions across 1188 Kaggle notebooks. Additionally, for each visualization function, we also have access to the *original* dataframe and column arguments needed to reproduce the visualization as seen in the parent notebook via ORIGDF and ORIGCOLS . We utilize this information heavily when analyzing these functions and using them for synthesizing visualizations in the next two sections.

4.2.3.1 Participating Columns vs. Column Parameters

Visualization functions have dataframe and column parameters. It is important to note that column parameters do not necessarily correspond to the exact subset of columns that actually *participate* in the visualization. For example, the function in Figure 4.7 accepts no column arguments, but produces a correlation heatmap of all the numeric columns in the passed dataframe. We call such columns *implicitly* participating columns. Consequently, we call a column as *explicitly* participating if it is passed as a column argument.

We can decide if a column is implicitly participating using a simple mutation-based strategy—for every column c in $\text{ORIGDF}(f)$ that is not mapped in $\text{ORIGDFs}(f)$, we drop c from $\text{ORIGDF}(f)$ and check if the visualization is the same after executing the function. If it is not, the column c is implicitly participating.

We denote the set of columns visualized (explicit or implicit) by f for the dataframe $\text{ORIGDF}(f)$ as $\text{ORIGPARTICIPATINGCOLS}(f)$. This notion of participation is at the heart of the reusability analysis as well as visualization code generation, as we shall see next.

```

1 def visualization(df, col1):
2     import matplotlib.pyplot as plt
3     counts = df[col1].value_counts()
4     porct =counts/1460*100
5     label = []
6     for i in range(len(counts)):
7         label.append(counts.index[i] + " "+ '{0:.2f}'
8     sizes = [1141, 286, 13, 11, 7, 2]
9     colors = ['steelblue', 'skyblue', 'navy',
10             'blue', 'red', 'green']
11     fig, ax = plt.subplots()
12     ax.pie(sizes, colors=colors, shadow=False,
13           startangle=0)
14     ax.axis('equal')
15     ax.legend(label, shadow=True)

```

Figure 4.8: A visualization function with hard-coded values.

4.2.4 Analysis of Mined Visualization Functions

Before we use the generated visualization functions to service user queries, we need to assess their *quality*. What makes a mined visualization function “good” (or “bad”) in the context of programming assistance? Since code generation, by its very nature, involves the construction of visualizations for an *unseen* dataframe, a visualization function should be considered “good” or *reusable* if, given appropriate assignments to column parameters, it produces *meaningful* visualizations for a *broad class* of dataframes, and “bad” or *non-reusable* otherwise. We illustrate our notions of *meaningful* and *broad* using examples.

Consider the visualization function in Figure 4.8. Note that the data values passed to `ax.pie` in line 12 are hard-coded in the function. That is, regardless of the dataframe and categorical column passed to the function, the produced visualization will be exactly the same. The produced visualization is thus not meaningful. If this function is used in a visualization code generation setting, its resulting visualization would most likely make no sense to the user and could undermine trust in the system. Thus we deem this function to be *non-reusable*. This also illustrates why a successful execution of a function does not necessarily entail a meaningful visualization.

In contrast, we consider the function in Figure 4.7 as “good” or *reusable*. It will correctly produce a correlation heatmap for the class of dataframes that have at least one numeric column. This class clearly includes a wide variety of dataframes and hence we consider this function *reusable*.

Figure 4.9 presents a much more subtle scenario. The function plots a histogram of the values in `co12`, but only considers the rows where the value corresponding to `co11` is 1. This filtering criteria is quite arbitrary and only meaningful for dataframes that contain a 1. We thus deem this function non-reusable..

```

1 def visualization(df, col1, col2):
2     import seaborn as sns
3     sns.set(font_scale=2.5)
4     df[df[col1] == 1][col2].hist()

```

Figure 4.9: A visualization function using a specialized predicate.

4.2.4.1 Defining Reusability

We consolidate the ideas developed in the above discussion in the following definition of *reusability*:

Definition 4.2.4 (Reusable Visualization Function). We consider a visualization function f *reusable* if there exists a set S_{df} of dataframes such that:

1. f produces a meaningful visualization for every dataframe df in S_{df} , given an appropriate assignment of df 's columns to f 's parameters. A meaningful visualization is non-empty and represents all the information in df or a filtered view of df where the filtering criterion is independent of the concrete data values in df . By the representation of information, we mean that some attribute of a visual element (a line, a dot, an axis, etc.) is directly or indirectly influenced by every data point.
2. S_{df} can be characterized using high-level properties of a dataframe and its columns, including types of columns and types of data values, but excluding properties relying on arbitrary constants or values in the data.

Note that a meaningful visualization need not follow best visualization design practices that would make it “meaningful” for an end-user. With reusability, we are only concerned about its relationship to the data and the visualization function code.

Ideally, we would like to be able to *automatically* classify our mined visualization functions as reusable and non-reusable and discard the non-reusable functions. However, it is hard to automatically check if a visualization function is reusable according to Definition 4.2.4 as we do not have access to S_{df} . Essentially, we are faced with the problem of a missing *test-oracle* [129]. We present a novel approach of using metamorphic-testing [22] to alleviate this issue.

4.2.4.2 Metamorphic Testing for Checking Reusability

Metamorphic testing relies on a *metamorphic relation* (MR): a property that must be satisfied by the outputs of a function for different inputs. Our choice of this property for a visualization function f is defined as follows:

Definition 4.2.5 (MR for Approximating Reusability). Visualizations produced by f on mutated copies of its original dataframe i.e. $\text{ORIGDF}(f)$ must all be different from each other as well as the original visualization of f .

These mutated dataframes are produced using column-level *type-aware* mutation operators. Definition 4.2.5 along with these mutation operators approximates the concept of reusability in Definition 4.2.4 in two ways. First, these operators only modify one column and take the column type (categorical, quantitative, etc.) into account. This helps increase the likelihood of staying within the class of dataframes f is appropriate for. It also ensures that this class is characterizable using simple properties like column types. Second, the mutations applied are large enough to warrant a change in the visualization if f truly produces a visualization that represents all the information in the dataframe or a meaningful subset of it. This helps catch cases like Figure 4.8 and Figure 4.9

Algorithm 8 Checking Reusability using Metamorphic Testing

```

1: function ISREUSABLE( $f$ )
2:    $df_{orig} \leftarrow \text{ORIGDF}(f)$ ;  $\nu_{orig} \leftarrow \text{ORIGVIZ}(f)$ 
3:   for each  $c \in \text{ORIGPARTICIPATINGCOLS}(f)$  do
4:     success  $\leftarrow$  false
5:     for each mutation operator  $m$  for  $\text{COLTYPE}(c, df_{orig})$  do
6:        $s \leftarrow$  initialize  $m$ 
7:       if  $\text{GUARD}(m, df_{orig}, c, s)$  then
8:          $df_1, \dots, df_k \leftarrow m(df_{orig}, c, s)$ 
9:          $\nu_1, \dots, \nu_k \leftarrow$  viz produced by  $f$  on  $df_1, \dots, df_k$ 
10:        if  $(\forall i. \nu_i \neq \nu_{orig}) \wedge (\forall i \neq j. \nu_i \neq \nu_j)$  then
11:          success  $\leftarrow$  true
12:          break
13:        if success is false then
14:          return false
15:  return true

```

Algorithm 8 formalizes our metamorphic testing strategy. For every *visualized* column $c \in \text{ORIGPARTICIPATINGCOLS}(f)$, we check if there exists a mutation operator for which the metamorphic relation is satisfied for the mutated dataframes it generates. Every mutation operator has a guard that must be true for it to be applicable (line 7).

Our mutation operators for columns take the type of the column into account and are listed in Table 4.1. We recognize four distinct types of columns, namely categorical, quantitative, ID and nominal. At a high level, for each type of column, we design a mutation operator for each of the different ways in which a column of that type may participate in a visualization. We walk through the operators for the two most common types of columns: categorical and quantitative.

Categorical Columns The visualization may be a function of either (1) the individual category labels in the column, or (2) the count distribution of categories, or (3) whether a value is a NaN (missing value). Note that the visualization may represent a *function* of these properties, which may not necessarily be the identity function. The first operator in Table 4.1

Table 4.1: Column Mutation Operators

Column Type	Mutation Operator	Guard
Categorical	Replacing a <i>fixed</i> subset of values with new categories	-
Quantitative	Shifting and Scaling Values + Gaussian Noise	-
Categorical / Quantitative	Replacing a <i>fixed</i> subset of values with missing values	Replacing the same subset with arbitrary values does not change visualization
ID	Random Permutation	-
Nominal	Replace a subset of values with a sample from the remaining values	-
Nominal	Replacing a <i>fixed</i> subset of values with missing values	Replacing the same subset with values sampled from the column does not change visualization

selects a fixed subset of values and replaces them with one or more unseen categories. Thus, if a function relies on hard-coded values or too arbitrary a filtering process, the resulting visualizations should be the same and thus fail the check. The second operator enables the check of whether the visualization is sensitive to whether values are NaNs or not, rather than their concrete values themselves. It also has a guard which checks whether substituting the same missing values with different categories yields the same result as the original. This ensures that cases like Figure 4.9 do not pass the check.

Quantitative Columns The visualization may be a function of either (1) the values, (2) a statistical function of those values, or (3) whether a value is a NaN. The first operator shifts and scales the data by different amounts and adds some Gaussian noise, thus testing (1) and (2). We add noise because some statistical functions, such as Pearson correlation, are robust to uniform scaling and shifting. The magnitude of the shift is at least as large as the range of values to ensure zero overlap with the original range of values. Null values are handled similarly as in categorical columns.

The mutation operators for ID and nominal columns are designed using similar principles. VIZSMITH is able to discard 26% of mined functions by classifying them as non-reusable via this approach. We evaluate how well the metamorphic testing approach approximates the main definition of reusability in Section 4.3.2.

4.2.5 Visualization Code Generation

VIZSMITH accepts a user specification comprising dataframes, a list of columns in each dataframe that need to *participate* in the visualization, and a search query. VIZSMITH uses the search query to get a ranked list of visualization functions from the database obtained using the mining and analysis components from Sections 4.2.2 and 4.2.4. Then for each function, VIZSMITH determines the best possible assignments to the dataframe and column arguments, runs the function, collects the visualizations generated, and presents them to the user after deduplication.

4.2.5.1 Search

Algorithm 9 Instantiating Visualization Functions

```

1: function INSTANTIATE( $f, df, vcols$ )
2:    $V \leftarrow \emptyset$ ;  $params \leftarrow COLPARAMS(f)$ 
3:    $M \leftarrow$  set of all injective maps from  $params$  to  $vcols$ 
       $\triangleright$  Score is non-zero and every col in  $vcol$  is mapped to a param or potentially implicit
4:    $M_{valid} \leftarrow \{m | m \in M \wedge SCORE(f, df, m) > 0 \wedge \forall c \in vcols. (ISIMPLICITCAND(f, df, c) \vee \exists p \in$ 
       $params. m[p] = c)\}$ 
5:   for each  $m$  in  $RANK(M_{valid}, SCORE)$  do
6:      $\nu \leftarrow f(df, m)$ 
7:     if  $\nu$  is valid then
8:        $V \leftarrow V \cup \{\nu\}$ 
9:   return  $V$ 

10: function SCORE( $f, df, m$ )
11:    $df_{orig} \leftarrow ORIGDF(f)$ ;  $m_{orig} \leftarrow ORIGCOLS(f)$ ;  $score \leftarrow 0$ 
12:   for each  $p \in COLPARAMS(f)$  do
13:      $c_m \leftarrow m[p]$ ;  $c_{orig} \leftarrow m_{orig}[p]$ 
       $\triangleright$  Column-types must match for the mapping to be valid
14:     if  $COLTYPE(df_{orig}, c_{orig}) \neq COLTYPE(df, c_m)$  then
15:       return 0
16:      $d_{orig} \leftarrow DTYPES(df_{orig}, c_{orig})$ ;  $d_m \leftarrow DTYPES(df, c_m)$ 
17:      $score \leftarrow score + (|d_{orig} \cap d_m| / |d_{orig} \cup d_m|)$ 
18:     if  $HASNULLS(df_{orig}, c_{orig}) = HASNULLS(df, c_m)$  then
19:        $score \leftarrow score + 1$ 
20:   return  $score$ 

```

VIZSMITH associates each mined visualization function with a text document that contains (a) the natural language comments around the visualization statements in the parent notebook, (b) the text in the title and axis labels of the visualization in the parent notebook, (c) the names of the API functions used and (d) the API documentation of the API functions used in the visualization function. We collect comments from the notebook under the assumption that authors often attach meaningful comments describing the logic in and before/after cells, although this may not always be true.

Given a search query, we rank documents according to their similarity with the search query using BM25 [9]. To obtain a ranked list of visualization functions, we simply map the documents back to their respective visualization functions.

4.2.5.2 Generating Visualizations

VIZSMITH adapts the ranked visualization functions to the user-provided dataframe using the INSTANTIATE function in Algorithm 9. It takes as input the mined visualization function

f , the user-supplied dataframe df and the columns that must participate in the visualization $vcols$.

In the first phase (lines 3-4), the set of mappings from the column parameters of f to a subset of $vcols$ is computed. A mapping is valid if (a) it has a non-zero score, and (b) the columns in $vcols$ that have not been assigned to a parameter as per the mapping are eligible to be visualized *implicitly*.

The SCORE function computes the score of a mapping m for a visualization function f by comparing m to ORIGCOLS(f). Recall that ORIGCOLS(f) is the mapping column parameters to the string values in the parent visualization slice of f . Essentially SCORE checks the compatibility between the columns using high-level properties such as column, data types, and presence of null values.

We consider a column eligible to participate implicitly (ISIMPLICITCAND) if there exists a column in the original set of implicitly participating columns of f that has the same column-type and data-types. The rationale is that if a column participates implicitly, the criterion determining its participation is most often a function of the column and data types.

In the second phase (lines 5-9), the mappings are tried one by one, highest-score first. All the unique valid (non-empty) visualizations collected are returned at the end.

4.3 Evaluation

We focus on three main research questions to evaluate VIZSMITH:

RQ1: How diverse is the collective functionality of all visualization functions Specifically, we explore the distributions over the size of the functions, the APIs explored, and whether a function performs data pre-processing.

RQ2: How accurate is our metamorphic testing approach? We perform a manual study comparing ground-truth reusability results to those obtained automatically via our metamorphic testing approach to gauge its efficacy.

RQ3: What is the end-to-end code generation performance of VIZSMITH? We evaluate the efficacy of VIZSMITH’s code generation approach via the use of an automatically mined corpus.

4.3.1 RQ1: Diversity of Functionality in Mined Corpus

As it is infeasible to manually examine each function and classify its functionality, we approximate it as the set of API functions used in the body of the visualization function. Note that we only consider functions classified as reusable by VIZSMITH. We find that all mined functions collectively exercise a total of 289 API functions across 12 third-party libraries. We further bucket each API function manually into four categories using simple criteria, namely

Table 4.2: Competition Statistics. # notebooks is the number of notebooks eligible for execution. \checkmark , \emptyset , \top , \times indicate that at least one viz was mined, no visualizations mined, timeout and error respectively. # viz. funcs is the number of visualization functions mined with reusable count in brackets.

competition	# kernels (\checkmark / \emptyset / \top / \times)	# viz. funcs (reusable)
LANL-Earthquake-Prediction	219 (46/159/0/14)	90 (86)
covid19-global-forecasting-week-1	223 (78/121/16/8)	397 (212)
house-prices	1025 (437/161/108/319)	3832 (2772)
mercari-price-suggestion-challenge	286 (29/227/24/6)	120 (67)
mercedes-benz-greener-manufacturing	44 (18/8/6/12)	95 (64)
otto-group-product-classification	77 (29/33/5/10)	68 (68)
santander-customer-satisfaction	63 (19/32/10/2)	39 (23)
santander-value-prediction-challenge	213 (16/72/14/111)	64 (47)
titanic	1015 (486/239/45/245)	4745 (3604)
tmdb-box-office-prediction	208 (30/30/62/86)	290 (233)
total	3373 (1188/1082/290/813)	9740 (7176)

Table 4.3: Top-10 API functions in each category, and the number of reusable viz. functions that use the API.

plotting	transform	computation	styling
sns.heatmap (1064)	pd.groupby (367)	pd.corr (687)	mpl.title (948)
sns.countplot (1019)	pd.drop (316)	pd.isnull (352)	sns.set (882)
sns.distplot (827)	pd.fillna (308)	pd.mean (241)	mpl.ylabel (707)
sns.barplot (629)	pd.sort_values (264)	pd.sum (221)	mpl.xlabel (565)
sns.boxplot (576)	pd.dropna (235)	pd.value_counts (217)	mpl.xticks (377)
sns.factorplot (370)	pd.concat (72)	pd.replace (126)	sns.set_style (344)
mpl.scatter (304)	pd.reset_index (63)	pd.isna (80)	mpl.set_title (302)
sns.scatterplot (213)	pd.pivot_table (53)	pd.median (67)	mpl.legend (252)
mpl.hist (212)	pd.get_dummies (52)	pd.nlargest (63)	sns.add_legend (207)
sns.catplot (180)	pd.head (35)	pd.count (61)	mpl.set_ylabel (176)

Table 4.4: Characterization of misclassifications by our metamorphic testing approach. FP and FN stand for false positive and false negative respectively

ID	Category	Num. Cases
A	Arbitrary Filtering using Multiple Columns (FP)	3
B	Undetected Over-Specialization (FP)	4
C	Visualization Design Choices (Bucketing/Axis-Limits) (FN)	4
D	Overaggressive Mutation (FN)	14
E	Adequately General Filtering Criterion (FN)	4

(a) “plotting” if it draws a visualization, (b) “transformation” if it involves reshaping or filtering operations such as transpose, groupby, and dropping null rows, (c) “computation” if it involves mathematical operations such as correlation and skew and (d) “styling” if it only modifies the cosmetic attributes of a visualization or the text inside it.

We find that 100%, 27%, 38% and 80% of visualization functions use APIs in categories (a), (b), (c) and (d) respectively. The top-10 API functions in each category with respect to the number of visualization functions using the API are listed in Table 4.3. Evidently, VIZSMITH’s database covers a wide variety of plotting, styling and transformation operations.

4.3.2 RQ2: Accuracy of Metamorphic Testing

Section 4.2.4 introduced the conceptual definition of reusability of visualizations. We also proposed an approach using metamorphic testing where the metamorphic relation approximated this concept of reusability. In this RQ we measure the accuracy, precision, and recall of this metamorphic testing approach with respect to a ground truth obtained via manual inspection of the visualization functions using the conceptual definition.

We sampled 50 reusable and 50 non-reusable visualization functions as judged by our metamorphic testing approach. We then designed an interface that displays these 100 functions one-by-one in a random order. Three of the authors labeled each function as reusable or non-reusable as per Definition 4.2.4. We computed the ground-truth label via majority vote. In particular, the authors try to assess the intent of the visualization, the class of dataframes where a similar visualization would be meaningful, and whether the implementation would be able to produce that visualization without any modifications.

We find the accuracy of the metamorphic approach to be 71%, with a precision of 73% and recall of 71%. There were 7 false positives (ground-truth non-reusable, classified reusable) and 22 false negatives. We categorized these cases in Table 4.4. The category column summarizes the reason for the misclassification of the metamorphic testing approach. Examples of these categories are shown in Figure 4.10.

The 7 false positives occur because our mutation operators are only applied on one column at a time (category A), or the code performs overly specific transforms that are not triggered

<p style="text-align: center;">Ⓐ</p> <pre>def visualization(df, col1, col2): import matplotlib.pyplot as plt train_df = df.drop(df[(df[col1]>4e3) & (df[col2]<3e5)].index) plt.scatter(train_df[col1], train_df[col2])</pre>	<p style="text-align: center;">Ⓑ</p> <pre>def visualization(df, col1): import seaborn as sns df[col1]=df[col1].fillna("S") df[col1]=df[col1].map({"S":0,"C":1,"Q":2}) sns.heatmap(df.corr(), annot=True)</pre> <p style="text-align: center;">Ⓒ</p> <pre>def visualization(df, col1): import matplotlib.pyplot as plt df[col1].hist(bins=5, grid=False) plt.xlabel(col1)</pre>	<p style="text-align: center;">Ⓓ</p> <pre>def visualization(df, col1, col2): import numpy as np import matplotlib.pyplot as plt df[col1]=np.log(df[col1]) plt.scatter(df[col1],df[col2])</pre> <p style="text-align: center;">Ⓔ</p> <pre>def visualization(df, col1): import seaborn as sns ms = df[df[col1] > 0] sns.barplot(ms.index, ms[col1])</pre>
---	--	--

Figure 4.10: Examples of each category in Table 4.4.

by mutations (category B), and hence pass the metamorphic testing check. The majority of the false negatives occur because of over-aggressive mutation (category D). The example in Figure 4.10 uses a log function that throws an error when our mutation introduces negative values. In 4 cases, the design choice of using bucketing or changing the axis limits led to the same visualizations being produced despite the mutations (category C). Finally, there were 4 cases in Category E where the filtering was not arbitrary (all positive values), but was judged to be the case by our approach. All categories except E can be handled by a more sophisticated mutation scheme or finer-grained operators. Category E would require a pre-defined notion of what is an adequately general filtering criterion.

4.3.3 RQ3: Code Generation Performance

Finally, we evaluate the end-to-end code-generation performance of VIZSMITH. We reuse the Kaggle notebooks utilized for mining to create benchmarks. For every visualization slice we extracted in Section 4.2.2.3, we select a visualization function and create a benchmark where the dataframe corresponds to the original dataframe i.e. $\text{ORIGDF}(f)$ and the columns to visualize are $\text{ORIGPARTICIPATINGCOLS}(f)$. The natural language query is set to the text document associated with f as described in Section 4.2.5.1. We select the largest visualization function, in terms of statements, whose statements all come from the same cell in the parent notebook. The rationale is that this simulates a real usage scenario for VIZSMITH as notebook cells often correspond to a single semantic unit of work. We also only consider reusable functions as benchmarks. This yields 3284 benchmarks in total.

For each benchmark, we create an instantiation of VIZSMITH using only visualization functions mined from competitions other than the one corresponding to the benchmark (leave-one-out cross-project). We then run VIZSMITH as well as a baseline version of VIZSMITH called VIZSMITH_{ALL} that searches over all visualization functions, including non-reusable functions on each benchmark till they generate 10 visualizations or timeout after 60 seconds, whichever is earlier.

We find that both VIZSMITH and VIZSMITH_{ALL} have a very low top-10 accuracy of 5%. That is, both have an *exactly* matching visualization in the top-10 for only 5% of the cases.

Through manual inspection, we determined two broad reasons for this low performance: (a) the quality of the natural language query is poor, and (b) styling variations such as color schemes, rotation of tick labels and legend positions will fail the matching visualization test. In particular, with respect to a sample of 100 visualization functions, we found only 17% to actually describe the kind of plot and the columns being visualized. Thus (a) is a distinct possibility. To mitigate the effects of (b), we sample 50 benchmarks and examine the results of both tools manually. In particular, we ignore stylistic variations such as color schemes, rotations of tick labels, legend positions, etc. while comparing the visualizations with the ground truth. Note that the goal of VIZSMITH is not to correctly handle queries with precise styling information — this is best left to direct manipulation technology (see Section 4.4.4).

The top-10 accuracy, when ignoring stylistic changes, is 56% and 46% for VIZSMITH and VIZSMITH_{ALL} respectively. Although the numbers are close, the difference lies in the number of functions explored. VIZSMITH explores 50% fewer visualization functions than VIZSMITH_{ALL} while still getting slightly better accuracy as it only searches over reusable functions which we hypothesized to be more useful during code generation than their non-reusable counterparts. Hence, we demonstrate the utility of reusability analysis to improve end-to-end performance.

4.4 Discussion

We elaborate on the main taking points behind VIZSMITH’s design and its limitations, and highlight opportunities for improvement

4.4.1 Real-World Usage

We have not performed an explicit user study to gauge the performance of users using VIZSMITH on real-world visualization authoring tasks. Hence, the results in Section 4.3.3 may not apply to real use cases. Note that performing such a study would require careful experimental design to decouple the techniques behind VIZSMITH from the quality of the mined code as well as the associated natural language comments, which are often imprecise or even irrelevant. To enable external assessment, we have released a fully functioning prototype of VIZSMITH along with a simple UI at <https://github.com/rbavishi/vizsmith-demo>.

4.4.2 Code Licensing and Security

VIZSMITH’s database is populated using code written by data scientists and machine learning practitioners that is publicly available on the Internet. As such, code snippets returned by VIZSMITH may not be appropriate for use in certain contexts due to the license of the parent notebook containing the code snippet. This can be mitigated by passing an appropriately vetted corpus to VIZSMITH. Security may also be a concern since VIZSMITH executes every function in its database as part of its metamorphic testing phase. We could mitigate this by

adding extra checks to filter out functions with excessive resource consumption, unauthorized file system access, or network requests.

4.4.3 Construct Validity

All three research questions involve manual analysis and thus have a subjective component. For RQ1, we classified the functions manually. To reduce the effect of subjectivity, we provided simple and easily reproducible criteria for arriving at this classification. For RQ3, we analyzed the generated visualizations manually because it is hard to automatically identify stylistic variations in a reliable manner. We precisely listed down the classes of stylistic variations that we ignore while comparing two visualizations. Judging reusability as per Def. 4.2.4 involves manual inspection of the code, the data, and the visualization. Thus, RQ2 has a higher risk of imprecision than RQ1 and RQ3. We mitigated this by having three reviewers independently judge reusability and taking the majority vote. We also assessed the misclassifications qualitatively and characterized the failure cases.

4.4.4 Integrating Direct Manipulation

VIZSMITH failed to produce an exact-match visualization on most queries as matching against stylistic variations without explicit information is difficult. That being said, VIZSMITH is suitable for complex visualization tasks that combine data processing steps to reshape the data into the right format before plotting. Stylistic improvements are best suited for direct manipulation [5], and thus integration of VIZSMITH with direct manipulation capabilities for touching up the produced visualizations can be potentially useful.

4.5 Summary

In this chapter, we presented VIZSMITH, a programming assistant for generating visualization code given keyword-like natural language queries. With VIZSMITH, we introduce a novel approach to search space design for aligned code and natural language, utilizing program slicing to automatically mine diverse visualization code along with code comments from Kaggle. We additionally present techniques to solve the resulting challenges from utilizing code online: code quality and reuse. Not all publicly available code is equally good, and we leverage simple yet effective metamorphic relations tailored for visualizations to discard low-quality code snippets. Data analysis code often explicitly refers to elements of the underlying data it is operating on, such as column names or specific values. Consequently, the code must be suitably modified before it is applied to another compatible dataset. We also use sophisticated program analysis to lift the code into reusable templates applicable to a much wider variety of data. We integrated our prototype of VIZSMITH within Jupyter notebook, and a demo is available publicly at <https://github.com/rbavishi/vizsmith-demo>.

VIZSMITH also introduces unsolved challenges. Building aligned code and natural language corpora using human-written code comments, while convenient, comes with the risk of poor natural language quality. Indeed, the evaluation results suggest that VIZSMITH is strongly affected by the poor quality of descriptions with only 17% of the natural language descriptions accurately describing its paired code snippet in the VIZSMITH’s corpus. The final assistant contributed by this dissertation, namely DATANA described in the next chapter, tackles exactly this problem.

Chapter 5

Datana: Leveraging Automatic Code Summarization for Code Generation

In the last chapter, we introduced VIZSMITH, a tool that accepts the data to visualize along with a keyword-based natural language description to generate visualization code recommendations. It services such queries by performing lookups on an indexed corpus of aligned visualization code and natural language description pairs. Our key insight and contribution in VIZSMITH were to leverage the abundance of open-source visualization code available online on platforms such as Kaggle. We used dynamic program slicing to collect visualization code from computational notebooks hosted on Kaggle and used program analysis along with metamorphic testing to create reusable visualization code templates. More importantly, we use a heuristic to collect accompanying natural language descriptions — we rely on the assumption that analysts and scientists often include code comments and rich markdown descriptions explaining their workflow in the notebook. Thus, we extracted the code comments and markdown content surrounding the mined visualization code.

However, we found a sizeable proportion of descriptions obtained in this manner to be irrelevant or imprecise. This is primarily because (1) analysts and scientists often do not include detailed descriptions of every single analysis step, and (2) they often only describe their findings or conclusions from a visualization rather than describe the visualization itself. Thus, VIZSMITH does not do well on more precise queries that specify lower-level details such as mark types, color schemes, titles, font styles and sizes, and legend placements.

It is thus natural to ask: can we eliminate the need to rely on human-written, possibly imprecise, and/or irrelevant natural language descriptions for code? This chapter explores the idea of automatically generating high-quality natural language descriptions for a corpus of code using machine learning and then using it for code generation.

Using Large Language Models (LLMs) for Code Summarization

Recent advances in large-scale, task-agnostic, pre-training to generate language representations have unlocked a wide spectrum of abilities, most notably in-context learning in language

models [16]. In-context learning for language models refers to a class of tasks where instruction for a task is provided as a natural language prompt, optionally along with a handful of examples. The model is then expected to complete the tasks for unseen examples by simply predicting the text that comes after. In other words, the model is expected to adapt to a new task on the fly without any modifications to its weights. Empirical scaling laws, or the performance trend of models, indicate relatively smooth trends of improvement in performance as model sizes increase on text-synthesis tasks [57] as well as in-context learning tasks [16]. This has led to major efforts in training very large language models with hundreds of billions of parameters such as GPT-3 [16], and PaLM [25] on vast amounts of data obtained by principled crawling of the internet. Models such as Codex [21] have been trained on specially-crafted corpora of code and text and have demonstrated remarkable ability at a variety of code-related tasks [21, 10]. Thus, we can exploit these advances to use models such as Codex, specifically their in-shot learning capability with examples or few-shot learning, to generate natural language descriptions of code snippets automatically.

Why Not Use LLMs Directly for Code Generation?

If we can use Codex to generate code summaries, why not use it directly for code generation from natural language queries provided by the user? Models such as Codex indeed show remarkable performance on general-purpose programming benchmarks [21, 10, 78]. Copilot [40], a code completion tool powered by Codex, has been a commercial success.

However, these models are not perfect. A recent study by Vaithilingam et al. [119] reported that participants were often bogged down by erroneous code produced by Copilot and had to spend significant time debugging. This is especially problematic when users are not familiar with how to solve a particular task with a given set of APIs, a scenario that is of core interest in this thesis. Thus, there is a clear need to build upon and improve the performance of this technology.

Retrieval-Based LLM Augmentation for Code Generation

In this chapter, we present an approach, and the resulting system DATANA, that combines the benefits of an automatically assembled corpus of aligned natural language and code pairs with the code-generation ability of large language models. Specifically, given a code context and a natural language query, DATANA uses the query to retrieve the best-matching pairs of code snippets and natural language descriptions from the corpus. Then for each retrieved example, it creates a prompt containing the retrieved example, the original context, and finally, the query. Then DATANA simply returns the completions generated by Codex for these prompts as suggestions.

Essentially, DATANA breaks down the code-generation process into two stages: retrieval and generation. The retrieval stage fetches relevant examples from a corpus which is then used as a reference in the generation stage alongside the target query. The retrieved example provides additional help, such as hints about the syntax and semantics of the functions to

use or even which algorithm to use to solve a particular task. This breakdown is analogous to the two stages of the coding process employed by human developers and analysts — first, a high-level plan of operations is devised that can be mapped to the abstractions available, which is then translated into code. While analysts are able to do the first step owing to their programming background, it is the second step of translation that requires them to look up documentation or search online. In the same way, retrieved examples act as a reference for easing the burden of the translation step for Codex.

This insight is also shared by Parvez et al. and Drain et al. in their respective retrieval-augmented code generation systems [84, 30]. However, they rely on corpora created by scraping functions and their documentation from Github or StackOverflow [51]. DATANA’s approach, which is also its key contribution, is to automatically generate such corpora by utilizing large language models to automatically summarize mined code snippets. To ensure a high degree of fidelity of a description generated in this manner to the corresponding snippet, we introduce the novel notion of bidirectional consistency. Simply put, bidirectional consistency ensures that the target code is reproducible from the generated description. In other words, bidirectional consistency ensures that the generated description contains enough information about the snippet to enable its reproduction. Furthermore, REDCODER and Drain et al. perform explicit training of models for both the retrieval and generation tasks. Thanks to the few-shot learning ability of LLMs, we are able to use Codex itself for the generation task and only need to train a relatively much smaller CodeBERT model to perform retrieval of examples from the corpus using sentence embeddings [97].

Overall, we find that this approach improves Codex’s accuracy by 26% on a large set of real-world tasks involving the generation of dataframe-manipulating programs from natural language. Furthermore, compared to the state-of-the-art approach JIGSAW, which applies sophisticated program analysis to repair an erroneous Codex solution using input-output examples, DATANA improves performance by 4%. This is especially remarkable as DATANA does not require the use of any input-output examples or domain-specific post-processing strategies as employed in JIGSAW.

In the next section, we walk through classes of scenarios where Codex returns erroneous suggestions and how DATANA’s retrieval approach helps Codex correct its mistakes.

5.1 Motivation

In this section, we walk through classes of scenarios where large language models like Codex can underperform. These classes highlight the different ways in which DATANA’s approach of retrieving examples from a large, automatically-mined corpus can help Codex generate the desired code snippet by providing additional context in the form of retrieved examples.

5.1.1 Correcting Algorithmic or Functional Errors

This scenario illustrates how DATANA helps Codex correct algorithmic or functional errors. This includes errors such as using the wrong API functions or wrong combinations of them, missing a crucial operation, or misinterpreting the intent altogether.

The code snippet below contains a context where a dataframe is initialized and stored into a variable. The intent of the analyst is recorded as a comment at the end, and the italicized and highlighted assignment to `dfout` corresponds to the incorrect suggestion from Codex. While the suggestion seems okay at a glance, it actually returns an empty dataframe. The reason is that the inner expression `dfin[dfin == "?"]` simply evaluates to a dataframe where the "?" with NaNs or null values. The index is thus exactly the same as `dfin`. Thus dropping the index of this new dataframe from the original `dfin` is equivalent to dropping all the rows. Essentially, Codex missed a `dropna` step right after `dfin[dfin == "?"]` which would drop all the rows containing nulls.

```
dfin = pd.DataFrame({
'A': {0: 1, 1: 1, 2: 2, 3: 0, 4: 0, 5: 1, 6: 0},
'B': {0: 0, 1: 0, 2: '?', 3: 1, 4: 1, 5: 0, 6: 1},
'C': {0: 0, 1: 7.5, 2: 1.5, 3: 3.0, 4: '?', 5: 4.5, 6: 6.0},
'D': {0: 0, 1: 2, 2: '?', 3: 1, 4: 0, 5: 1, 6: 2}
})

# drop rows in 'dfin' where values are '?'
dfout = dfin.drop(df[dfin == '?'].index) # incorrect
```

Recall that given a query, DATANA first fetches relevant examples from its automatically assembled corpus of aligned code and natural language descriptions. One of the examples retrieved by DATANA for the query above is shown below. While it does not exactly fit the scenario of dropping rows from an entire dataframe, it does contain the algorithm for dropping rows containing zeros in a series corresponding to a specific column.

```
# Return a series containing the "population" column in "df" with 0 values replaced by NaN and
rows with NaN values dropped
df['population'].replace(0, np.nan).dropna()
```

Providing this example to Codex (details in Section 5.2.6) helps Codex adapt the algorithm and make it work for the target query:

```
dfin.replace('?', np.nan).dropna() # correct
```

It is worth noting that this is not a trivial code adaptation involving a simple renaming of variables or constants exercise for Codex. It has to recognize that the original snippet is targeting a specific column while the goal is to apply the operation across an entire dataframe. One of the baselines in our experiments uses a much smaller model than Codex and trains it directly on DATANA's corpus. We found that it very much struggles with such tasks, highlighting the need for models trained at a scale similar to Codex.

5.1.2 Highlighting Important Query Fragments

In this following scenario, we show how Codex can sometimes miss important aspects of the query and how DATANA can help out. Similar to the previous scenario, the code snippet below contains the context (initialization of a dataframe variable) and the query as a natural language comment. Codex's suggestion (italicized and highlighted) is subtly incorrect. In particular, it does the replacement correctly but performs the replacement across the entire dataframe while the query specifically requests the replacement to be confined to the "city" and "country" columns. In other words, Codex misses acting upon an important fragment of the query.

```
dfin = pd.DataFrame({
  'country': {0: 'USA', 1: 'FR', 2: 'UK', 3: 'India', 4: 'India'},
  'city': {0: 'LA', 1: 'PAR', 2: 'London', 3: 'MUM', 4: 'DEL'},
  'random': {0: 'India', 1: 'Mumbai', 2: 'DU', 3: 'London', 4: 'IN'}
})

# Replace 'London' in column 'city' with 'LON', and 'India' in column 'country' with 'IN'
dfin.replace(to_replace=['London', 'India'], value=['LON', 'IN'], inplace=True) # incorrect
```

One of the best matches in the corpus, as deemed by DATANA's retrieval algorithm (Section 5.2.6.1), is shown below.

```
# Replace values "Ms" and "Mme" in the "Title" column with "Miss" and "Mrs" respectively
df['Title'].replace('Ms', 'Miss').replace('Mme', 'Mrs')
```

The example deals with replacing two values in a single column and doing so in a manner that is quite different and arguably irrelevant to the target query. However, the example serves as a clear illustration of confining the replacement to a specific column. Accordingly, Codex picks up on this connection between the example's description and corresponding code and returns the correct solution, as shown below.

```
dfin.replace({'city': 'London', 'country': 'India'}, {'city': 'LON', 'country': 'IN'}) # correct
```

The code correctly restricts the replacement of "London" with "LON" to the "city" column, and "India" with "IN" to the "country" column.

5.1.3 Providing Solutions with Low Adaptation Overhead

This final scenario illustrates how DATANA can assume the burden of code generation altogether when the retrieved example's functionality *exactly* satisfies the target query's requirements. As before, the context below contains the initialization of the dataframe and its subsequent assignment into a variable. Codex's suggestion for the target query, which is supplied as a comment, is shown at the end in italics and with a yellow highlight. The suggestion is a bit off, as the query clearly requests a single number that corresponds to the

number of duplicated rows, but the suggestion simply returns the duplicated rows *without* de-duplication. Note that the count of these rows would additionally include the first occurrence for every duplicated row. While the natural language is certainly ambiguous, counting duplicates more commonly does not include the first occurrence.

```
dfin = pd.DataFrame({
  'inp1': {0: 5, 1: 5, 2: 15, 3: 3, 4: 3, 5: 33, 6: 3},
  'inp2': {0: 12, 1: 12, 2: -5, 3: 7, 4: 7, 5: 14, 6: 7},
  'inp3': {0: 17, 1: 17, 2: 4, 3: 9, 4: 9, 5: 17, 6: 9},
  'target': {0: 0, 1: 0, 2: 1, 3: 0, 4: 0, 5: 1, 6: 0}
})

# count number of duplicate rows in 'dfin'
dup_rows = dfin[dfin.duplicated(keep=False)] # incorrect
```

The example retrieved by DATANA *exactly* corresponds to the desired functionality. All that is needed to be done is to replace the variable `df` with `dfin`.

```
# Return the number of duplicate rows in "df"
df.duplicated().sum()
```

This task of variable replacement is relatively easy for Codex, and it expectedly comes up with the right solution as shown below.

```
dfin.duplicated().sum() # correct
```

These three scenarios illustrated the various ways in which DATANA helps Codex arrive at the right solution. It is worth noting that the examples retrieved by DATANA are created using Codex itself. The key insight in DATANA is that instead of asking Codex to solve a task in a single step, we break it down into two steps: (1) creating a knowledge bank offline that helps capture low-level syntactic, semantic, and algorithmic details, and (2) piecing together the information from the examples as well as the target query online to solve the task.

5.2 Technique

This section describes the technical details behind DATANA. We first give a high-level overview of DATANA's architecture, followed by a detailed and formal treatment of every individual component.

5.2.1 Architecture Overview

Figure 5.1 gives an overview of the architecture of DATANA, for both the offline and the online stages. In the offline stage, we prepare the corpus of aligned natural language and code snippets related to data analysis. We use static program analysis, specifically type inference

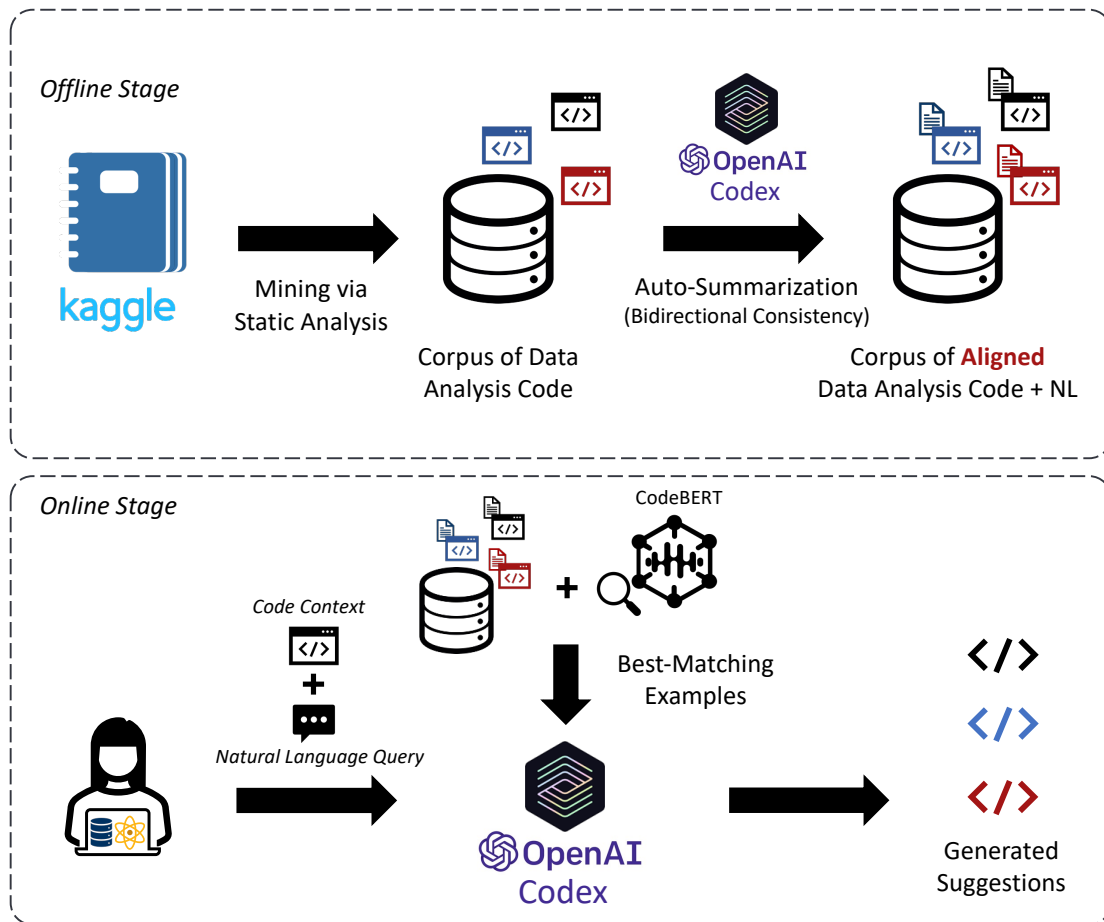


Figure 5.1: Overview of DATANA architecture

(Section 5.2.2), to process $\sim 400k$ notebooks from the data science platform Kaggle [116]. Then, for each of the snippets mined, we use Codex to automatically generate descriptions of varying styles for the mined snippets (Sections 5.2.3, 5.2.4, and 5.2.5). To ensure a high degree of fidelity of the generated description to the corresponding snippet, we introduce the notion of *bidirectional consistency*. Simply put, bidirectional consistency ensures that the target code is reproducible from the generated description. Thus, we obtain a corpus of aligned code and high-quality natural language pairs.

In the online stage, DATANA accepts the current code context and natural language query as input from the user. This context corresponds to the contents of the current file being edited by the user in an IDE of their choice. The natural language query can be simply supplied as a code comment. Then, examples from the corpus best matching the query are retrieved. The retrieval algorithm in DATANA (Section 5.2.6.1) uses a CodeBERT model [38] to generate neural vector representations for all the descriptions in the corpus. The model

$$\begin{array}{c}
\text{TYPE} \\
\frac{e \in \mathcal{E}_{all} \wedge \text{type}(e) \in \{\text{DataFrame}, \text{Series}, \text{Groupby}\}}{e \in \mathcal{E}_{pandas}} \\
\\
\text{INIT} \\
\frac{e \in \mathcal{E}_{pandas}}{e \in \mathcal{E}_{mined}} \\
\\
\text{SUBSCRIPT} \\
\frac{e \in \mathcal{E}_{all} \wedge e \equiv v[i] \wedge v \in \mathcal{E}_{mined}}{e \in \mathcal{E}_{mined}} \\
\\
\text{ACCESSOR} \\
\frac{e \in \mathcal{E}_{all} \wedge (e \equiv v.\text{acc}[i] \vee e \equiv v.\text{acc}.v') \wedge v \in \mathcal{E}_{pandas} \wedge \text{acc} \in \text{KnownPandasAccessors}}{e \in \mathcal{E}_{mined}} \\
\\
\text{ATTRIBUTE} \\
\frac{e \in \mathcal{E}_{all} \wedge e \equiv v.\text{attr} \wedge v \in \mathcal{E}_{pandas} \wedge \text{attr} \in \text{KnownPandasAttributes}}{e \in \mathcal{E}_{mined}} \\
\\
\text{PANDAS-API} \\
\frac{e \in \mathcal{E}_{all} \wedge e \equiv v.\text{fn}(a_1, \dots, a_n) \wedge v \in \mathcal{E}_{pandas}}{e \in \mathcal{E}_{mined}} \\
\\
\text{REGULAR-FUNC} \\
\frac{e \in \mathcal{E}_{all} \wedge e \equiv \text{fn}(a_1, \dots, a_n) \wedge \exists i. a_i \in \mathcal{E}_{mined}}{e \in \mathcal{E}_{mined}}
\end{array}$$

Figure 5.2: Inference rules, applied till fixed-point, for mining expressions from a computational notebook. \mathcal{E}_{all} is the set of all Python expressions in the notebook, \mathcal{E}_{pandas} is the set of expressions with types related to pandas, and \mathcal{E}_{mined} is the final set of expressions mined.

is trained to generate vectors that are closer together for similar pairs of natural language descriptions and farther apart for dissimilar pairs. The examples are then combined with the user’s code context and query into a single text prompt for Codex (Section 5.2.6.2), and the top-5 completion suggestions are returned.

We now describe each of the components inside DATANA in detail.

5.2.2 Mining Pandas Expressions

Given a Kaggle notebook, we use standard static type inference to deduce the types of all the expressions in the notebook. Specifically, we use the Mypy [117] optional static type checker along with specialized type stubs for the Pandas and Numpy libraries. We obtain these stubs by building upon third-party stubs [62]. Then, using this type information, we repeatedly apply the rules described in Figure 5.2 till fixed-point is achieved to obtain the final set of mined expressions from the notebook, denoted by \mathcal{E}_{mined} .

The TYPE rule in Figure 5.2 simply collects all expressions with inferred types being one of the core pandas [118] types — dataframe, series, or a groupby expression, into the set \mathcal{E}_{pandas} . The INIT rule states that any expression with a pandas-related type is part of the mined set \mathcal{E}_{mined} . All the other rules build off of these core set of expressions to mine more complex ones.

The SUBSCRIPT rule mines expressions that represent a subscript (array) access on an expression that is already part of the mined set. This helps mine expressions such as `df["col"]` used to access a particular column in a dataframe. The ACCESSOR rule helps collect expressions exercising *accessors* in pandas such as `df["Date"].dt.weekday` where the `dt` accessor converts the dates in the "Date" column into a format from which weekdays can be extracted. We maintain a set of known accessors denoted by *KnownPandasAccessors*. Similarly, the ATTRIBUTE rule helps mine expressions such as `df.shape` which evaluates to the shape of the dataframe. We maintain a set of available attributes denoted by *KnownPandasAttributes*. Next, the PANDAS-API rule allows mining of expressions that correspond to Pandas API calls called on a pandas object such as `df.head(5)`. Finally, the REGULAR-FUNC rule helps mine function calls that has at least one argument that is already considered as part of the mined set. An example is `sns.heatmap(df.corr())` which creates a heatmap of column correlations in the dataframe using the `seaborn` library.

We designed these rules in an iterative manner, with the overall aim to capture a majority of one-liner expressions involving pandas objects that we deemed common and with potential value. Using these rules, from ~400k Kaggle notebooks, we built a mined corpus containing 2.5 million expressions, out of which ~760k were syntactically unique.

5.2.3 Auto Code Summarization using Large Language Models

After the corpus of expressions is mined, we automatically generate natural language descriptions for each of the expressions or snippets using a large language model, specifically Codex [21] from OpenAI. Codex contains 11 billion parameters and has been pre-trained on vast amounts of code and text. Specifically, we use the few-shot learning ability of models such as Codex [16] to generate these natural language descriptions.

A few-shot example for summarization is simply a tuple $\langle c, d \rangle$ where c is a code snippet or expression, and d is its corresponding natural language description. We manually create a set of five few-shot examples by randomly selecting five snippets from the mined corpus and writing down their descriptions.

Codex is an autoregressive language model — it is trained to predict the next token given a context of previous tokens. Therefore, to generate descriptions for a target code snippet, we create a prompt containing (1) a description of the task, followed by (2) the few-shot examples, and finally (3) the target code snippet itself. The prompt is passed to Codex, which is then expected to *complete* the prompt up until a pre-determined *stop token*. In other words, the model is expected to pick up the pattern and generate the description of the target code snippet. Figure 5.3 shows an example of such a prompt.

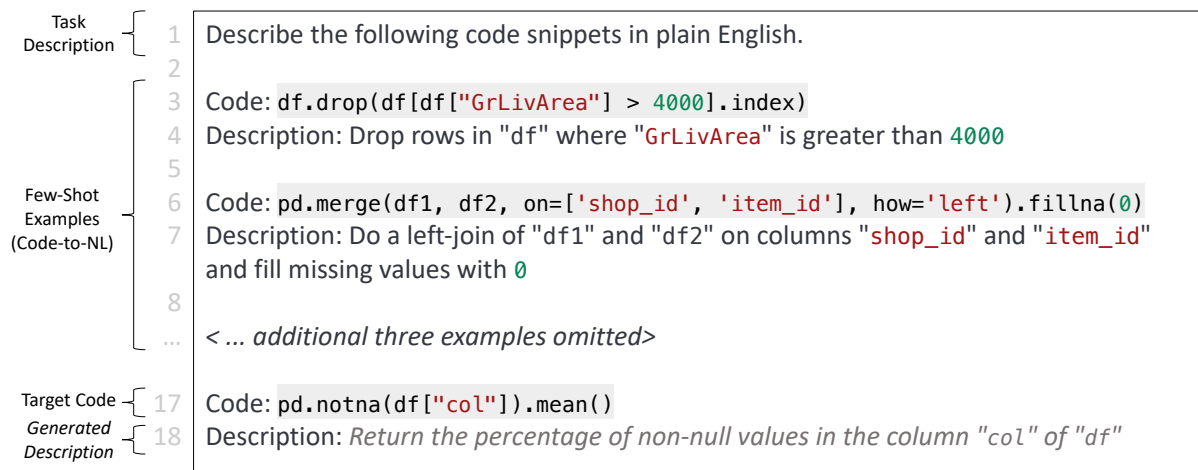


Figure 5.3: Prompt supplied to Codex for generating descriptions of code snippets. The gray, italicized text at the end corresponds to the Codex-generated description.

5.2.4 Bidirectional Consistency

Simply generating descriptions using the previously outlined approach does not guarantee the description's correctness. How do we know if the generated description precisely and completely describes the code snippet? Given the size of the corpus, it is infeasible to devise a manual or semi-manual verification approach. Our key insight here is that this concept of correctness can be operationalized as follows.

Definition 5.2.1 (Description Correctness). We say a description d for a code snippet c is correct if it is *precise* and *complete*. That is, the information contained in the description is necessary and sufficient for an oracle that generates code from natural language to reproduce the semantics of the code snippet c .

Definition 5.2.1 assumes the availability of an oracle that faithfully translates any arbitrary natural language description into corresponding code whenever possible. While a perfect oracle of this nature is impossible to obtain, we approximate it by exploiting the few-shot learning ability of the same large language model, Codex, that we used for generating the descriptions in the first place. We use the same prompting methodology as before, simply flipping the natural language and code positions for the few-shot examples and providing the target description to generate code for at the end. Figure 5.4 shows an example of a prompt to generate code back from natural language. We formalize this use of prompting to generate natural language descriptions from code as well as code from natural language as the two functions $CodeToNL$ and $NLToCode$ respectively.

Definition 5.2.2 ($CodeToNL(c, FS)$). The function $CodeToNL$ returns a set of strings corresponding to the natural language description of the target code snippet c using a large

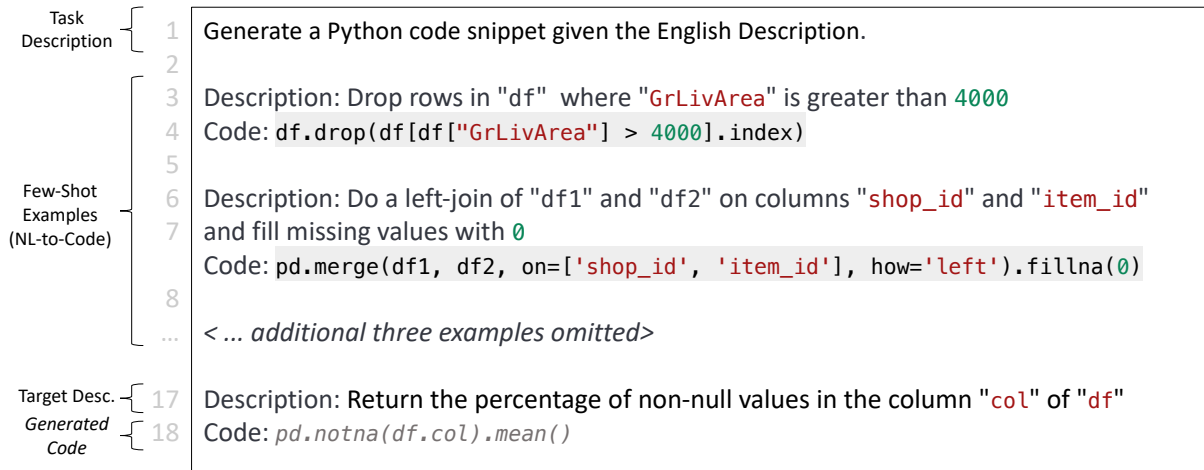


Figure 5.4: Prompt supplied to Codex for generating code from natural language descriptions. The gray, italicized text at the end corresponds to the Codex-generated code.

language model (LLM) and the set of few-shot examples FS .

Definition 5.2.3 ($NLToCode(d, FS)$). The function $NLToCode$ returns a set of generated code snippets from the given description d using a large language model (LLM) and the set of few-shot examples FS .

Definition 5.2.1 also uses the concept of *semantic equivalence* in that it allows the oracle to generate a code snippet that is syntactically different but preserves the semantics of the original code snippet. Checking equivalence of programs is undecidable in general, and approximations such as observational equivalence on a finite set of inputs are not applicable in our setting as our mining procedure does not extract inputs to the mined snippets. Thus, we further approximate this check as syntactic equivalence modulo code normalization. That is, the oracle (model) is expected to *exactly* reproduce the original code from the description. We use code normalization to eliminate superficial syntactic and simple semantic differences, such as normalizing keyword argument orders, and replacing attribute-based column access with subscript-based access, and code formatting. For example, the generated code in Figure 5.4 after normalization will be `pd.notna(df["col"]).mean()`. Note that syntactic equivalence is an underapproximation of semantic equivalence — every pair of syntactically equivalent code snippets are also semantically equivalent, but the reverse does not hold.

We call the correctness check with these oracle and equivalence approximations, *bidirectional consistency*, as we use the same model for generating as well as verifying descriptions.

Definition 5.2.4 (Bidirectional Consistency, $c \xleftrightarrow{FS} d$). We say that a code snippet c is bidirectionally consistent with a description d (and vice-versa) with respect to a set of few-shot examples FS , also denoted by $c \xleftrightarrow{FS} d$, if the following holds:

```
Code: df.drop(df[df["GrLivArea"] > 4000].index)
Description:
* Drop rows in "df" where "GrLivArea" is greater than 4000
* Use the DataFrame.drop function and explicitly use the index

Code: pd.merge(df1, df2, on=['shop_id', 'item_id'], how='left').fillna(0)
Description:
* Do a left-join of "df1" and "df2" on columns "shop_id" and "item_id" and fill missing values with 0
* Use the pandas.merge and DataFrame.fillna functions and explicitly use the "on" and "how" keyword args
```

Figure 5.5: Examples of helper descriptions for snippets, italicized with a yellow highlight.

$$c \xleftrightarrow{FS} d \equiv (d \in \text{CodeToNL}(c, FS)) \wedge (\exists c' \in \text{NLToCode}(d, FS). \text{SNorm}(c) = \text{SNorm}(c'))$$

where SNorm is a function that applies syntax normalization, which includes operations such as normalizing keyword order, unifying variable names, and code formatting.

We use bidirectional equivalence to obtain as many descriptions as possible for a code snippet within a budget. Since Codex has request and token rate limits in place, we generate 10 descriptions for a snippet and then only retain the ones passing the bidirectional consistency check. Having multiple descriptions for a snippet helps in matching as well as training the retrieval models (Section 5.2.6.1).

5.2.4.1 Better Approximating Semantic Equivalence

Using syntactic equivalence modulo normalization as described above is a sound approximation to semantic equivalence. However, in practice, we found it to be too strict. The pandas library has a large number of function aliases (`pd.merge` and `df.merge`, `pd.melt` and `df.melt`, `df.fillna(method="ffill", ...)` and `df.ffill(...)`), equivalent argument combinations (both `df.drop(["B", "C"], axis=1)` and `df.drop(columns=["B", "C"])` are equivalent), and multiple ways of performing a set of operations in general. We found that enforcing syntactic equivalence in such a scenario led to poor coverage of the mined corpus in terms of the proportion of snippets for which bidirectionally consistent snippets could be generated.

Therefore, we augment the natural language descriptions with auxiliary or helper descriptions. Helper descriptions contain information about the functions and the arguments invoked but do not give away information about how to combine the functions or what values the arguments should take. In other words, these helper descriptions contain just enough information to tackle the aliasing and multiple-approaches problem described above. Figure 5.5 gives an example of a helper description alongside the main description. How do we generate these helper descriptions? We simply use Codex again by manually writing and

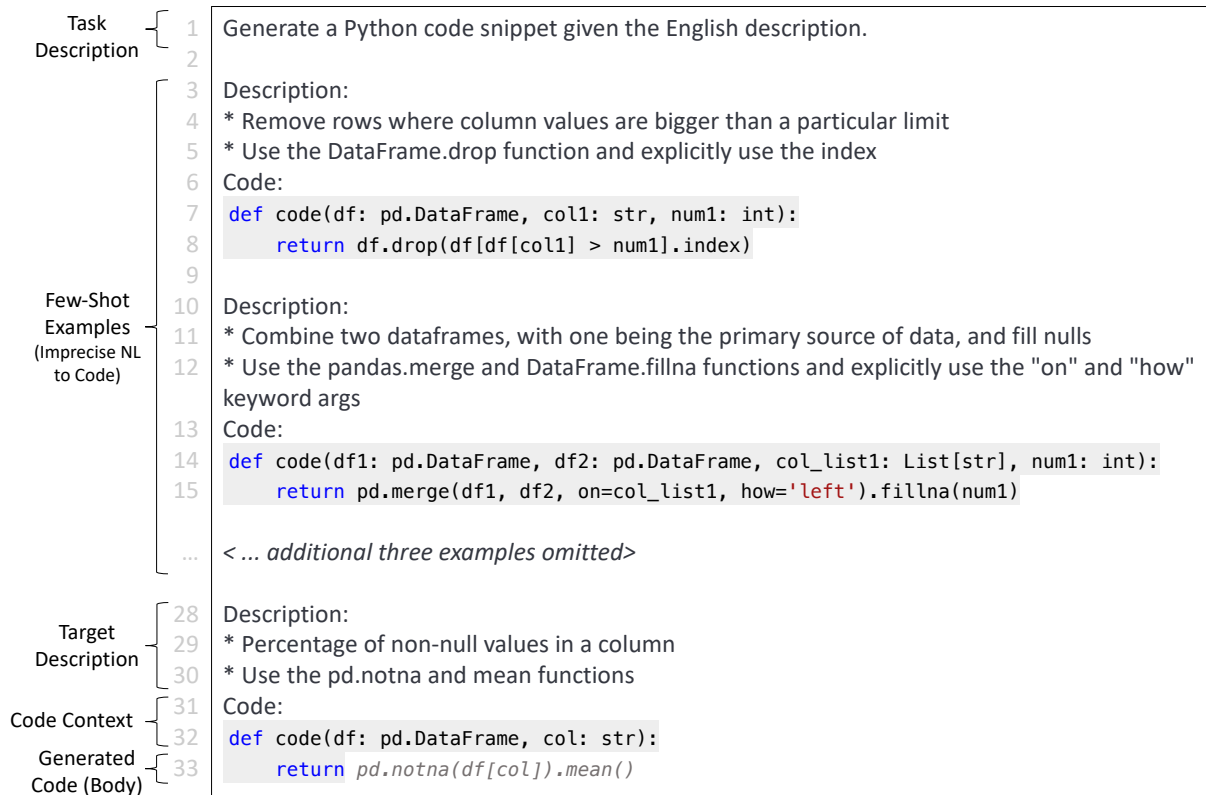


Figure 5.6: Prompt supplied to Codex for generating code from possibly imprecise or incomplete descriptions as part of the bidirectional consistency check. Additional code context is provided to fill the information gap in the description. The gray, italicized text towards the end corresponds to the Codex-generated code.

including helper descriptions in the few-shot examples and then using them in prompts in both directions to both generate them and use them for verification.

5.2.5 Generating Imprecise and Incomplete Descriptions

The style of descriptions generated in the previous steps, as exemplified by the descriptions in Figure 5.3, is arguably *not* representative of how a large fraction of analysts and developers might state their intent. In particular, it is written in an imperative style and explicitly states references, constants, and the variables used. Such details are often omitted from intent specifications in natural language, as evidenced by recent studies [136]. For example, for the target code in Figure 5.3, a user might very well put down the description as “Reading a non-comma CSV with specific index column”. In fact, this better resembles how people search on Google/StackOverflow — abstract concepts and the details are filled out or edited later within the code.

Ideally, we would like to be able to generate descriptions of this flavor. However, such descriptions are arguably imprecise and incomplete, which would preclude a bidirectional consistency check as the model will not have all the information to be able to reproduce the code back from the description.

The key idea behind still being able to perform a bidirectional consistency is to provide the model with additional context when generating code back from a description. In particular, we ask the model to complete the body of a function, where the parameters exactly correspond to the variable and constant references that are absent from a description. Figure 5.4 shows the prompt to generate the code from imprecise descriptions. The function signature provided at the end of the prompt helps the model fill in the details missing from the description. Note that to generate these descriptions, we use the prompt obtained by simply flipping the order of descriptions and code in the few-shot examples, as illustrated in Figures 5.4 and 5.3.

But a question remains - how do we obtain this context to provide to the model in the first place? We again use the few-shot learning ability of the language model to parameterize code snippets as functions and use the function signatures as the additional context.

5.2.5.1 Automatic Parameterization of Code Snippets

We first formally define a parameterization of a mined code expression or snippet.

Definition 5.2.5 (*Parameterization*($c, d, FS_{bd}, FS_{param}$)). Given a code snippet c and its description d such that $c \xleftrightarrow{FS_{bd}} d$, the parameterization of c with respect to the set of few-shot examples FS_{param} is a function, with $\rho = \{p_1, \dots, p_n\}$ as the set of n parameters and b as the body, satisfying the following:

$$\exists M : \rho \rightarrow \mathcal{E}_c \text{ such that } b[a_1/M[a_1]][\dots][a_n/M[a_n]] = c$$

where \mathcal{E}_c is the set of all sub-expressions in c .

Informally, Definition 5.2.5 simply states that for the parameterization to be valid, there must exist some assignment of its parameters to sub-expressions of c such that replacing the parameter variables in b with those expressions yields c itself. To automatically generate such parameterizations, we manually write down few-shot examples denoted by FS_{param} . Note that we use the same code snippets as the ones used as part of few-shot examples for generating descriptions. Figure 5.7 shows the prompt (and the few-shot examples) used for generating parameterizations. We check whether the output of Codex satisfies the constraints in Definition 5.2.5 using standard program analysis.

We use the approach described in Sections 5.2.3, 5.2.4, and 5.2.5 to generate descriptions for all the mined code expressions. Note that we attempt to generate multiple bidirectionally consistent descriptions for a single snippet by setting 0.5 as the temperature for Codex when generating descriptions. Setting a non-zero temperature increases the diversity of completions returned by Codex.

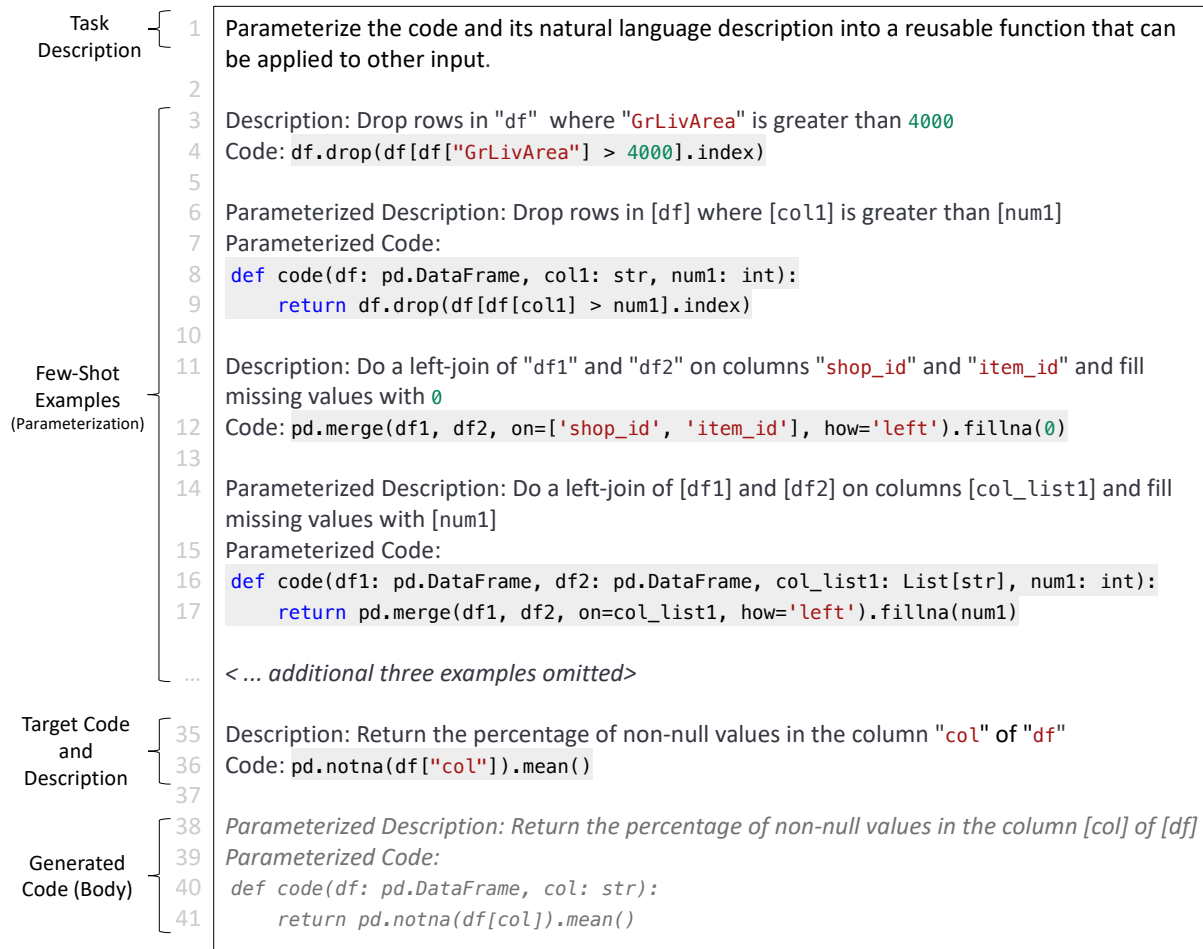


Figure 5.7: Prompt supplied to Codex for generating parameterizations of code snippets and their bidirectionally consistent descriptions.

5.2.6 Augmenting Codex for Improved Code Generation

Now that we have a corpus of code snippets aligned with their bidirectionally-consistent descriptions, we are ready to service online queries from the user in order to generate code from natural language. Recall that our goal in DATANA is to augment Codex when given a natural language query by providing the best matches from the corpus as *guides* for generating the correct code result. Thus, two problems need to be solved to enable augmentation of this sort: (1) retrieving best matches from the corpus given a natural language query, and (2) *engineering* the prompt to expose the results to Codex.

```
# Use the following code example as a guide to write code for the comment below

# Return a series containing the "population" column in "df" with 0 values replaced by
NaN and rows with NaN values dropped
df['population'].replace(0, np.nan).dropna()

dfin = pd.DataFrame({
'A': {0: 1, 1: 1, 2: 2, 3: 0, 4: 0, 5: 1, 6: 0},
'B': {0: 0, 1: 0, 2: '?', 3: 1, 4: 1, 5: 0, 6: 1},
'C': {0: 0, 1: 7.5, 2: 1.5, 3: 3.0, 4: '?', 5: 4.5, 6: 6.0},
'D': {0: 0, 1: 2, 2: '?', 3: 1, 4: 0, 5: 1, 6: 2}
})

# drop rows in 'dfin' where values are '?'
dfin.replace('?', np.nan).dropna()
```

Figure 5.8: Prompt supplied to Codex for code generation in the presence of retrieved examples. The gray, italicized text towards the end corresponds to the Codex-generated code.

5.2.6.1 Retrieval

We use the Sentence-BERT framework [98] to train a CodeBERT-base [38] model for producing embeddings of natural language descriptions that are closer together (cosine similarity close to 1) for descriptions that are semantically similar and further apart (cosine similarity close to 0) for dissimilar descriptions. Then given a natural language query, we use the model to produce an embedding of the query and find the top-k closest natural language descriptions from the aligned corpus. The code snippets associated with these descriptions constitute the desired matches.

Training such a model requires access to pairs of descriptions that are known to be similar (and dissimilar). We create similar pairs by picking two different bidirectionally consistent descriptions for the same code snippet. To create dissimilar pairs, we simply pair up descriptions for two randomly chosen code snippets. We further ensure that the training dataset contains an equal number of similar and dissimilar pairs to avoid class imbalance.

5.2.6.2 Prompt Engineering for Augmenting Codex

The vanilla prompt supplied to Codex for generating code from natural language consists of the code context followed by the natural language description as a code comment. The model is then expected to complete the prompt by generating code following the comment. To utilize retrieval results, we simply add a retrieved example towards the beginning of the

prompt, as shown in Figure 5.8, with explicit instruction for using the example as a guide. We create such a prompt for *every* retrieved snippet and collect Codex completions for all of them individually. The top- k results for DATANA then correspond to the completions for the top- $(k-1)$ retrieval results, plus the original completion from Codex, that is, the completion obtained without using any examples. We set $k = 5$ for the main set of experiments as it strikes a balance between giving the tool some leeway while not placing too large a cognitive burden on the users.

A natural alternative to our augmented prompt approach is to combine all retrieval results into a single prompt. However, we observed poor performance when using this strategy. Our hypothesis is that this is primarily because too many examples distract from the code context and make it unclear how to use them.

5.3 Evaluation

In this section, we evaluate DATANA using four main research questions:

RQ1: Does DATANA complement Codex and improve end-to-end performance on real-world benchmarks? We use a comprehensive benchmark suite (described in the next section) to measure the difference in accuracy between DATANA and vanilla Codex, among other baselines.

RQ2: How does DATANA compare against a state-of-the-art code generation tool? We compare against JIGSAW, a state-of-the-art approach for *repairing* incorrect solutions from Codex using sophisticated post-processing powered by program analysis.

RQ3: Does DATANA complement core improvements in large language models? Do fundamental architecture and/or training improvements subsume the benefits of DATANA?

RQ4: What is the contribution of different components of DATANA? We measure the significance of the major low-level design decisions in DATANA.

In order to answer these questions, we first describe the experimental setup — the benchmarks and the baselines.

5.3.1 Benchmarks

We use the same two datasets used in the evaluation of the Jigsaw system by Jain et al. [52]. Both datasets contain code generation tasks for data transformations using the `pandas` library, containing a natural language description and one or more input-output examples. The examples are primarily used to check the correctness of the generated programs if any.

Table 5.1: Performance (% Accuracy) on JIGSAW benchmarks. Results marked (*) are reported directly from JIGSAW evaluation results [52].

	Codex Version	% of Benchmarks Solved	
		PandasEval1	PandasEval2
DATANA Only-Generational (<i>Corpus Baseline-1</i>)	N/A	13.2	5.3
DATANA Generational+Search (<i>Corpus Baseline-2</i>)	N/A	14.7	4.3
JIGSAW Var-Renaming*	001	54.9	51.0
DATANA Only-Var-Context (<i>Codex Baseline</i>)	001	59.7	50.3
JIGSAW Few-Shot+Var-Renaming* (<i>Simple-Corpus Baseline</i>)	001	63.7	67.5
JIGSAW Full-Offline*	001	66.7	72.2
DATANA (top-5)	001	70.1	76.7
DATANA Only-Var-Context (<i>Codex Baseline</i>)	002	71.6	60.4
DATANA (top-5)	002	79.1	81.1

The first dataset, called PandasEval1, consists of 68 tasks collected from StackOverflow, with the natural language description being written by the authors themselves. The second dataset, called PandasEval2, contains 21 tasks for which the accompanying natural language descriptions were collected from 25 users as part of a hackathon. Since each task can have multiple associated queries from different users, this dataset comprises a total of 725 benchmarks.

Since PandasEval1 is constructed by the authors, the natural language descriptions resemble the precise descriptions we create using the approach in Section 5.2.4. That is, they are mostly complete and explicitly state the variables and constants to use. In contrast, since PandasEval2 is crowdsourced, the queries cover a wider spectrum and may not be as precise or complete.

Table 5.1 contains the results for DATANA, JIGSAW, and a number of baselines which we describe next.

5.3.2 Baselines and Other Systems

We develop four baselines to illustrate the benefits of combining search results and language model generation within DATANA.

5.3.2.1 Corpus Baselines

These baselines only use the mined corpus to solve the tasks. The first baseline, DATANA Only-Generational is a CodeBERT-base [38] model trained on our entire mined and aligned corpus to generate code from natural language descriptions. Note that for this baseline, we only use the *precise* description and code pairs from our corpus. The second baseline, DATANA Generational+Search is a hybrid baseline where a CodeBERT-base model is trained to only solve the instantiation problem — given a search result and the actual query, the

model simply adapts the search result to the query by replacing variable names or constants. Thus, these two baselines measure the effectiveness of the mined corpus in isolation for solving natural language to code tasks.

5.3.2.2 Codex Baseline

This baseline, DATANA Only-Var-Context, is used to measure the effectiveness of Codex in isolation for solving the tasks. The prompt provided to Codex in this setting is simply the variable context comprising the available dataframes, followed by the target natural language description as a code comment. This prompt exactly corresponds to the augmentation prompt in Figure 5.8 minus the retrieved example. We believe providing this context is important as it better simulates real-world scenarios where the data is presumably loaded into a variable before performing any analyses on top.

5.3.2.3 Simple Corpus Baseline

We also consider a baseline that uses a simpler and limited corpus of examples manually collected from forums such as StackOverflow and `pandas` documentation. We adapt the entry in the JIGSAW paper [52] corresponding to a version of their system that uses retrieved examples from their hand-collected corpus to provide additional context to Codex. We also choose the version with variable renaming enabled, which is closer to our setting for Codex where we provide the variable context.

5.3.2.4 Jigsaw Results

JIGSAW is a proprietary system hence we could not obtain the source. Since we use the same set of benchmarks, we report the results stated in the paper directly.

5.3.3 Codex Versions

A new version of Codex (`code-davinci-002`) was recently released with training data up until June 2021. While we are not aware of any further technical differences, we empirically see better performance with this version as opposed to the previous version (`code-davinci-001`). Since JIGSAW was published when only the old version (`code-davinci-001`) was available, only results using this model are directly comparable to JIGSAW. The improvements across the models form the basis for the third research question.

5.3.4 RQ1: End-to-End Performance Improvement Over Codex and Search Baselines

The results in Table 5.1 clearly demonstrate that DATANA’s approach of combining the strengths of Codex and retrieval significantly boosts end-to-end performance. DATANA, when

using the 001 version of Codex, correctly solves 70.1% and 76.7% of benchmarks in PandasEval1 and PandasEval2 respectively. *Codex Baseline* is only able to solve 59.7% and 50.3% of benchmarks in PandasEval1 and PandasEval2, respectively. Only using the mined corpus (*Corpus Baseline-1* and *Corpus Baseline-2*) leads to poor performance on both datasets. This confirms our hypothesis that search results need to be significantly altered to adapt to a new context and that this capability is difficult to learn solely from the mined corpus. Similar findings have been reported by developers when it comes to adapting StackOverflow examples for use in their own workflow [134]. At the same time, the search results contain enough useful information to help Codex use the right set of functions or pick up on information it previously missed.

The benefits of using a large, mined corpus as opposed to a simpler, limited, and hand-collected corpus are also clear. *Simple-Corpus Baseline* only solves 63.7% and 67.5% of benchmarks from PandasEval1 and PandasEval2 respectively. This is a relatively smaller improvement over *Codex Baseline* which is expected since many tasks are multi-step operations, examples for which are hard to collect just from documentation and a handful of StackOverflow posts.

5.3.5 RQ2: Comparison With Jigsaw

JIGSAW uses sophisticated semantic post-processing techniques, including variable renaming, argument transformations, AST-to-AST transformations, and enumerative search to *repair* results from Codex, additionally using the input-output examples. As shown in Table 5.1, this allows JIGSAW to solve 66.7% and 72.2% of PandasEval1 and PandasEval2 benchmarks.

While this form of post-processing is general in principle, it is still restricted by the logic and functionality expressed in the implementation. For example, an approach like JIGSAW would not be able to correct Codex’s output using an enumerative search if the repair, if any, is not expressible in the search space. DATANA simply augments Codex by providing retrieved examples in the prompt, thus retaining the full generality of the Codex’s code-generation capabilities. It is important to note that this approach, unlike JIGSAW, allows DATANA to work without the presence of input-output examples.

5.3.6 RQ3: Complementarity with Core LLM Improvements

Table 5.1 clearly shows a significant boost in performance when using the 002 version of Codex. Specifically, there is a jump of 11.9 and 10.1 percentage points on PandasEval1 and PandasEval2 between the Codex baselines. We are not aware of any differences between the two version apart from updated training data for 002. While one hypothesis may simply be that the 002 version may have JIGSAW benchmarks as part of the training data, we still find that DATANA boosts the performance by roughly the same amount. For 001, DATANA improves performance over the Codex baseline by 10.4% and 26.4% on PandasEval1 and PandasEval2 respectively. The gains are 8.5% and 20.7%, respectively, for the version 002.

Table 5.2: Performance (% accuracy) of DATANA on JIGSAW benchmarks for different top-k.

	Codex Version	% of Benchmarks Solved	
		PandasEval1	PandasEval2
DATANA (top-5)	001	70.1	76.7
DATANA (top-10)	001	77.6	80.7
DATANA (top-20)	001	79.1	82.5
DATANA (top-5)	002	79.1	81.1
DATANA (top-10)	002	80.6	83.9
DATANA (top-20)	002	85.1	85.8

Thus, these results provide promising evidence that DATANA’s benefits do not necessarily go away when using models trained with a better architecture or more comprehensive data.

5.3.7 RQ4: Ablation Study

We first study the effect of varying k in DATANA which controls the number of suggestions returned. Then we separately study the benefits of training a retrieval model as well as diversifying description styles in our corpus.

5.3.7.1 Effect of Varying Top-k

We choose k as 5 in the main set of experiments in Table 5.1. Table 5.2 shows the improvement in performance as we increase k . For both models, the boost in performance is significant. This is not surprising as increasing the number of retrieval examples improves the chances of fetching more relevant examples. These results also signal an opportunity to improve the search mechanism. In our current implementation of DATANA, we use existing training approaches on off-the-shelf models to form the retrieval component. There is room for innovation here to rank the more relevant results higher.

5.3.7.2 Ablations for Retrieval Architecture and Description Styles

In this final research question, we evaluate the benefits of lower-level design decisions in DATANA. Specifically, we measure the benefits of training the CodeBERT model for search as opposed to using the vanilla CodeBERT model since it has already been trained on a large corpus of natural language and code. We also measure the benefits of diversifying the style of descriptions in our corpus (Section 5.2.5). Table 5.3 presents the results for these ablations on PandasEval1 and PandasEval2. Vanilla CodeBERT corresponds to the setting where the model is not trained for search, and No-Add-Desc is the setting when additional diverse descriptions are ignored during training as well as retrieval.

Table 5.3: Ablation Study for JIGSAW benchmarks. DATANA is compared against using vanilla CodeBERT as the search engine (Vanilla CodeBERT) and without using additional descriptions in the search corpus (No-Add-Desc)

	Codex Version	Percentage of Benchmarks Solved	
		PandasEval1	PandasEval2
Vanilla CodeBERT	001	70.1	73.2
No-Add-Desc	001	73.1	72.1
DATANA	001	70.1	76.7
Vanilla CodeBERT	002	77.6	77.9
No-Add-Desc	002	82.1	79.6
DATANA	002	79.1	81.1

Although small, we do see improvements from these two components, especially on PandasEval2. We hypothesize this is the case because PandasEval2 contains more diversity in terms of the descriptions owing to its crowdsourced nature. Thus, having a variety of descriptions for the same code snippet to match against and training a model to pick up on these differences should, in principle, lead to improvements. On PandasEval2, we see considerable improvement when *not* using additional descriptions. A possible explanation is that PandasEval1 contains more precise and imperative descriptions; thus, focusing only on such styles of descriptions leads to better performance. This also indicates room for improvement on the retrieval component of DATANA.

5.4 Discussion

We elaborate on the limitations of DATANA’s approach and opportunities therewith.

5.4.1 Disambiguation and Ranking

Our primary evaluation results use the top-5 suggestions from DATANA to compute the accuracy on the benchmark sets. This is made possible with the availability of input-output examples in the JIGSAW benchmarks as they allow lightweight verification of a candidate solution. This may not however reflect real-world usage; users cannot be expected to provide an example alongside a description every time when using DATANA. Thus, with the current usage scenario, we expect the users to inspect the suggestions manually, hence informing the choice of $k = 5$ to balance performance and the cognitive burden on the user.

However, such a manual inspection can be prohibitively difficult if the user is not well-versed with `pandas`. These users are very much the target audience for the assistants presented in this dissertation, including DATANA. Thus, there is a need to devise approaches to help users understand the results in an efficient manner. Options include disambiguation either on the code space or the input space [74], or visualization of the operations, whenever applicable [59]. There is also the possibility of innovations in ranking solutions, either by training additional, better models or incorporating domain-specific heuristics as a post-processing step.

5.4.2 Contextual Discoverability

Users may not always know how to best describe a particular operation and may benefit from natural language auto-completion suggestions that inform them about the set of possible operations [104]. Although the current implementation of DATANA does not have any auto-completion feature, DATANA’s underlying corpus of aligned natural language and code pairs seems like a natural solution for enabling basic auto-completion. However, this opens up doors for innovation on *contextual* auto-completion: a natural language auto-completion system that takes the current code or data context into account while showing or completing queries. Such a mixed-initiative approach could also help with the disambiguation problem above, as users can use the completed queries as a means of choosing between options immediately. In contrast, in the current implementation, users have to wait for DATANA to come back with suggestions given a single query crafted from scratch.

5.5 Summary

In this chapter, we introduced DATANA, a tool for generating data analysis code from natural language queries. We expanded upon the core idea developed in VIZSMITH in Chapter 4 of using an automatically mined corpus of aligned natural language and code pairs to retrieve

and adapt code. In particular, we identified the utility of such a corpus as a knowledge bank of a wide spectrum of analysis code, which can help break down the code-generation task into two stages: (1) retrieving a small set of relevant examples, and (2) combining insight from the examples to solve the task by either adapting one or more examples directly or stitching together useful components. This closely resembles workflows reported by developers when using forums such as StackOverflow — they often search for similar snippets, understand them, and adapt them suitably for use in their development context.

VIZSMITH relies on a heuristic to extract human-written descriptions for the mined code snippets along with a simple keyword-based search to retrieve examples. However, the downside is the low quality of such descriptions in terms of precision and relevance. This precludes precision searching for visualizations with specific characteristics. In this chapter, we eliminate the need of relying on human-written descriptions by using the few-shot learning ability of large language models such as Codex to automatically generate precise descriptions for the mined snippets. To verify the quality of these descriptions, we also develop a novel notion of bidirectional consistency. We also eliminate the need for sophisticated program analysis and testing as employed in VIZSMITH by using Codex again for the code generation stage.

Our experiments show that this combination of retrieval and Codex’s code-generation ability significantly improves upon the state-of-the-art, increasing accuracy by 4%, which can go as high as 10% with further improvements in retrieval. We also show that DATANA’s performance expectedly improves when the underlying language models are improved. More importantly, however, the benefits of DATANA’s retrieval-based augmentation are retained in the face of these model improvements. Overall, DATANA serves as a strong example of the promise of augmenting powerful machine learning approaches with carefully assembled knowledge banks using program analysis. We believe this line of research can lead to practically usable assistants in the near future.

Chapter 6

Related Work

This chapter discusses related work on program synthesis and program recommendation techniques generating code from a variety of input specifications, as well as other relevant flavors of programming assistants for data analysis powered by alternate technologies.

6.1 Code Generation

In this section, we survey and compare against a broad body of work on code generation techniques. Note that we use code generation as an umbrella term for program synthesis approaches where there is a checkable specification such as logical formulas or input-output examples, as well as recommendation or parsing approaches that generate code from non-checkable specifications such as natural language.

6.1.1 Intent Specifications

We first survey various intent specifications utilized by code generation systems.

6.1.1.1 Logical Specifications

Manna and Waldinger [72] proposed DEDALUS, one of the earliest program synthesis systems. Their inspiration came from the successful use of logical specifications in program verification. DEDALUS accepts specifications in a high-level language akin to first-order logic that *precisely* and *completely* captures the behavior of the desired program. In other words, the specification describes the behavior of the program on all possible (valid) inputs. Since such specifications provide rich information about the target program, powerful search algorithms based on deduction can be designed. However, this comes at the cost of ease of use; coming up with such a logical specification typically solicits a high degree of expertise from the user. For more complex domains that go beyond simple numbers and lists, coming up with such a specification may be as hard as writing the program itself. Subsequent research

has thus targeted specifications that are weaker (less precise and complete) but easier to write.

6.1.1.2 Input-Output Examples

Synthesizing programs from input-output examples, or *programming-by-example* (PbE), has been a popular research bet, buoyed by the commercial success of the seminal work FlashFill by Gulwani [42]. Since then, numerous systems have been developed, targeting a variety of domains such as string processing [42, 83, 29], data wrangling [35, 37, 64], data processing [108, 137], SQL queries [122, 138], syntax transformations [100], learning repair strategies for static analysis violations [15, 11], and bit-vector manipulations [53] among others.

6.1.1.3 Natural Language

Natural language has also been the specification of choice for many code recommendation systems targeting visualizations [41, 103], database queries [138, 141], repetitive text editing [28]. All these systems either only allow a restricted subset of the language or exhibit good performance only when queries are posed a certain way. That is, they are not suitable for arbitrary queries and thus implicitly impose a learning curve to using the tool. While supporting such arbitrary queries are long been intractable, advances in natural language processing, particularly large language models such as GPT-3 [16] have enabled a whole suite of programming-related tasks, including editing, code-generation, and summarizing.

6.1.1.4 Demonstrations

The programming-by-demonstration (PbD) [27] paradigm, where demonstrations are used as a specification, has been applied to a number of application domains. A prominent one is web-scraping, where systems such as Helena [18], and Rousillon [19] generalize a single demonstration of how to collect data from a specific web resource into a fully-fledged scraping program. While demonstrations, in general, can be taxing to provide [43], they capture the user’s intent better than input-output examples, which helps scale synthesis in difficult domains like these. Recall how GAUSS also exploits this fact to deliver superior performance.

6.1.1.5 Combinations of Specifications

Combinations of different specifications have also been explored. Along with input-output examples, SCYTHER [122] expects a bag of constants to be used in the target SQL query. MARS [23] also exploits keywords from natural language descriptions and short snippets from forums such as StackOverflow on top of input-output examples. TF-Coder [106] also uses natural language descriptions to speed up the search for a tensor manipulation program given input-output examples. Here, natural language is used as an auxiliary source of information to speed up the search while still using input-output examples as the primary specification.

6.1.2 Algorithms

Modern synthesis and recommendation search algorithms use either one or a combination of three strategies. The first strategy is to *prune* the search space — portions of the search space are systematically discarded based on the results of checking a candidate program against the specification. The second is to *bias* the exploration of the search space wherein manually-written heuristics or statistical approaches are used to impose an exploration order with the assumption that the desired program is higher up in the order. And third, using statistical models to predict whole candidate programs directly — the search space is implicit in this setting. In `AUTOPANDAS`, we exclusively rely on the second strategy as we use neural-backed generators to guide the search but do not eliminate any part of the search space without checking the candidates explicitly. The same holds for `VIZSMITH` as we use simple TF-IDF document similarity matching to rank mined visualization templates. `GAUSS`, however, is an example of a system utilizing both the first and second strategies together. We use graph-based reasoning and previous candidate failures to prune the search space while also using lightweight machine learning models to explore more likely function sequences first. Finally, `DATANA` is an example of the third strategy as we augment large language models to generate data processing and visualization code directly from natural language without an explicit representation of the search space, which in turn is arbitrary text.

In the next few sections, we present a categorization of prior work based on the classification of the primary search algorithm used per the taxonomy above and further drill down into the high-level approach developed.

6.1.2.1 Pruning via Logical Reasoning.

At a high level, approaches using logical reasoning either encode the synthesis problem as a constraint-solving problem and use SAT/SMT solvers to generate valid solutions to the constraints [53], or use logical specifications to prune the search space. These specifications can be manually specified [35, 91] or learned as lemmas during synthesis [37, 127]. These approaches are best suited for target domains that are amenable to logical specification, such as string processing and numerical domains.

6.1.2.2 Pruning via Domain-Specific Inductive Synthesis

This class of approaches involves the design of a special-purpose DSL tailored towards the target domain such as string processing [42], table data extraction [63], and learning code transformations [15]. Each DSL operator is backed by an algorithm called a *witness function* that prunes away impossible candidates for the given I/O example. Such approaches are highly efficient and can solve a large number of tasks provided their solutions can be expressed in the DSL. However, scalability issues arise if the DSL is too large or poorly designed.

6.1.2.3 Pruning via Abstractions

Abstractions have also been leveraged to simplify reasoning for complex search spaces. BLAZE [126] and ATLAS [128] use abstract semantics of the DSL components to construct a compact representation of all candidate programs to reason about them simultaneously. ATLAS learns these abstract semantics from a separate training set of I/O examples. SYNQUID [90] and MORPHEUS [34] prune invalid programs using refinement-types and first-order logic specifications respectively. MORPHEUS uses linear relationships between table attributes as specifications for components. Singh et al. [107] use I/O examples in the abstract domain of shapes, termed *storyboards*, to synthesize low-level data-structure manipulations. The shapes discard irrelevant details about the data structure. Our approach in GAUSS essentially combines the story-board and component-level specifications approach. GAUSS accepts partial input-output examples along with graph-based specifications of intent and searches the space of programs efficiently using the graph abstractions of components written by us.

6.1.2.4 Pruning via Type Information

Given a type specification of the target program SYPET [33] uses type specifications of individual API components to synthesize the target program as a composition of these components. The *TDE* system [46] mines a large corpus of API methods along with the raw arguments taken by those methods using static analysis, à la VIZSMITH. Then, given a new I/O example, it searches through this corpus using a mix of statistical and static analysis techniques. However, these methods rely on the availability of sufficiently rich type information about the API as well as the I/O example, something which is not available for popular Python libraries for data analysis such as pandas and matplotlib.

6.1.2.5 Biasing via Machine Learning

A number of systems exploit lightweight machine learning models to bias the search towards more likely candidates. MORPHEUS [34] and NEO [36] train n-gram models on sequences of R functions mined from public code repositories to explore candidates utilizing more likely sequences first. Chen et al. [23] train a seq2seq [114] model to predict likely function sequences from natural language.

Akin to AUTOPANDAS, DEEPCODER [13] trains a model using random data to predict likely components or functions to use given an input-output example. However, AUTOPANDAS goes a step further and uses models to predict the arguments for the functions.

This class of approaches uses probabilistic models to rank program candidates generated by the synthesizer [37, 96, 66]. These models are trained on data extracted from large open repositories of code such as Github and StackOverflow.

6.1.2.6 Generation via Machine Learning

Advances in deep learning have helped spur research in systems entirely based on machine learning models. These models take in a representation of the intent specification and directly predict the candidate program. A number of systems accept input-output examples and target domains such as string-processing and lists of bounded integers [14, 29, 54] where machine learning models such as cross-correlational networks, LSTMs with attention are applicable. However, these models cannot target dataframes of arbitrary shapes and values. This also precludes models like CNNs accepting a fixed-size grid as leveraged in the Karel domain [17].

Newer network architectures based on attention [12] such as the transformer [120] have led to tremendous advances in natural language processing, in particular large language models [16, 21]. These models are trained with the simple objective of predicting the next token given a context of previous tokens at scale. Models such as GPT-3 [16] and Codex [21] are trained on vast amounts of text and code and have shown unprecedented performance on difficult general-purpose programming benchmarks such as HumanEval [21]. Codex has also been integrated into the popular in-IDE programming assistant Copilot [40]. However, these models make mistakes [52], which are often subtle and can hamper productivity [119]. In DATANA we propose a retrieval-augmented approach that augments these models with an automatically generated knowledge base which significantly boosts correctness.

Similar to DATANA, REDCODER [84] and Drain et al.’s system [30] also augment language models using retrieval by fetching relevant examples from a corpus. The primary difference with GAUSS is how the corpus is obtained. The two systems use function-documentation pairs mined from Github or question-answer pairs from StackOverflow as the basis for their corpus. VIZSMITH uses the same idea, and thus, the imprecision and irrelevance problems mentioned in VIZSMITH are applicable to these systems as well. In contrast, DATANA automatically generate descriptions.

6.2 Alternate Programming-Based Data Analysis Assistants

While interactive interfaces often provide an easy-to-use interface to suggest interesting statistics and visualizations of data [132, 115, 112, 56, 95], they are not widely used by scientists and analysts with programming backgrounds, with a major factor being the prohibitive cost of switching between these tools and their preferred text and code editing platforms such as notebooks [130].

In response, tooling offering high-level abstractions for producing rich statistical summaries and visualizations of dataframes have been developed [1, 32, 81]. LUX [65] propose *always-on* visualization recommendations. Whenever a dataframe is printed to the console, LUX offers a quick overview of the data along with visualizations for interesting trends with the objective of guiding further exploration. LUX also features a rich domain-specific lan-

guage that users can employ to specify target visualizations on demand. Drawing on the need for visualization automation for exploration [101], LUX’s language also allows fuzzy specifications with wildcards to generate multiple visualizations for different attributes at once.

There has also been work on directly bringing the benefits of interaction and direct manipulation to coding environments [2, 31, 133]. The challenge lies in ensuring minimal friction while switching between the two modes. WREX [31] and B2 [133] propose specially-designed widgets in Jupyter notebooks [61] that maintain a tight integration between code and operations performed on the widget. WREX uses program synthesis to infer data transformation programs in an internal DSL, which is then translated to human-readable Python code. B2 tracks operations on dataframes and accordingly suggests visualizations. Interactions such as filtering and sorting on the visualizations are captured back in code. This helps keep the interaction reusable and reproducible.

The tools and techniques developed in this dissertation share the broad motivation of meeting the needs of data scientists and analysts with a programming background, to help them author complex analyses quickly or get started with an unfamiliar tool faster. As such, all four tools, namely AUTO-PANDAS, GAUSS, VIZSMITH, and DATANA, seek to complement the aforementioned systems by filling the *authoring gap*. While interaction and direct manipulation are important tools to help analysts be more productive in understanding and exploring their data, our tools aim to tackle the problem of authoring complex reshaping, processing, and visualization programs that may not be easily expressible in such forms. Examples include correlation heatmaps, rolling window calculations, min-max normalization, etc. This is facilitated by the use of high-level specifications suitable for different tasks — input-output examples for reshaping tasks, demonstrations for reshaping combined with computational tasks, and natural language for tasks involving conditionals or a complex composition of multiple operations.

Chapter 7

Conclusion

This dissertation presented four major techniques and prototypes of programming assistants for data analysis, with the end goal of improving the productivity of data scientists and analysts who prefer to use programming-based data analysis libraries such as pandas, matplotlib, and scikit-learn among others. These assistants accept high-level specifications such as input-output examples, demonstrations, and natural language and automatically generate suitable data analysis code.

Given the choice of specifications, the design of these assistants can then be understood from the lens of the two underlying dimensions: the search space, which controls the expressivity of the assistant, and the search algorithm, which controls the performance and/or accuracy of the assistant. Clearly, there is a tension between the two — making the search space larger will, in turn, make search that much harder. This then translates to a trade-off between expressivity and performance/accuracy. In this dissertation, we introduced new approaches to push the limits on this trade-off barrier by drawing upon insights from recent advances in program synthesis, testing, analysis, and machine learning.

In Chapter 2, we introduced AUTOPANDAS, an assistant that accepts input-output data frames and automatically generates programs that satisfy the example. That is if run on the input, the returned solutions produce the target output. The primary concern in AUTOPANDAS was the need to support a significant fraction of pandas functionality, whose size and complexity meant that the traditional approach of designing a compact DSL or a limited grammar of transformations would be insufficient. Our key inspiration in solving this problem of representing a complex search space came from random testing, specifically QuickCheck-style random generators. Generators present a natural, programmatic way of capturing a complex input space — arbitrary program logic, unrestricted by context-free grammars or any similar restrictive structures, can be used to precisely capture the parameter spaces. The choosing of parameters is delegated to simple random operators in the same programming environment. We used generators to instead capture the space of pandas programs. To guide the search, we replaced random operators with their *smart* counterparts using modern machine learning methods. The result is an enumerative search that tries out program candidates one by one, ordered by their likelihood of satisfying an example.

In Chapter 3, we identified a key issue with the use of input-output examples to specify dataframe transformations — they lose valuable information when the transformations involve computation and aggregation. However, this information is readily available; users know the precise relationship between the individual input and output cells when constructing the output. This information loss, quite needlessly, increases the burden on the assistant, and an enumerative approach like `AUTOPANDAS` devolves into inefficient trial-and-error. With `GAUSS`, we explored the possibility of using dedicated interfaces to preserve this information and developed a novel reasoning algorithm that led to significant speed-ups in search when such information could be exploited.

We then switched our focus to natural language specifications. While examples can be penned for a wide class of useful transformations, when it comes to complex statistical calculations or making visualizations, they can be limiting. Our goal behind enabling natural language is to give users a quick way to get started in code for their tasks, be it recalling the syntax of a particular operation or creating quick and simple visualizations that are still quite tedious to write from scratch. This shift also entailed a change in our approach to code-generation altogether, stemming from the inherent ambiguity of natural language specifications. For starters, it is not possible to adopt the enumerative approach of `AUTOPANDAS` and `GAUSS`, as, unlike examples, candidates cannot be *checked* against the query. In other words, the synthesis problem is turned into a recommendation problem. Fundamentally, the loss of the ability to iteratively *prune* the search space using a smart search algorithm shifts part of the generation burden onto the search space itself. The search space needs to be designed in a manner that facilitates retrieval or instantiation of a small pool of candidates given the target query, which can then be ordered by a dedicated ranking component for presentation to the user.

In Chapter 4, we took a new approach to search space design. We applied dynamic program analysis for *mining* a search corpus of reusable visualization code automatically from publicly available computational notebooks hosted on the data science platform Kaggle. Additionally, we extracted surrounding code comments and markdown descriptions to align the extracted code with their natural language descriptions. Thus, we reduced code generation to a retrieval problem: code snippets from the corpus whose aligned descriptions best match the query are returned. We argued that this approach has two main benefits: (1) being crowdsourced, the mined corpus is diverse and expressive enough, and (2) the need for manual design is eliminated, and the corpus can be continuously updated automatically. However, one major issue was the quality of extracted descriptions. Since we used a heuristic, there were no guarantees around the relevance or precision of these descriptions.

Finally, we presented a solution to this issue in Chapter 5 and the resulting tool `DATANA`. We fully eliminated the need to rely on human-written descriptions by exploiting the few-shot learning ability of large language models to generate descriptions for snippets in a mined corpus. The code generation component was also replaced with the same large language model. Intuitively, we split the code generation task into two stages: the creation of a knowledge bank and subsequent retrieval, extracting relevant information from the retrieved examples to solve the task. We drew parallels to how human developers and analysts solve

these tasks in a similar way — they gather additional resources and help online, and then adapt and generalize from the gathered information to come up with the right code.

Future Work

This dissertation primarily innovates on the technical side of the overall problem of building useful assistants, contributing techniques and algorithms to improve the expressivity and performance of these assistants. There is a significant opportunity for innovation in improving the usability of these assistants.

Humans are not perfect, and thus the specifications they provide may also be erroneous from time to time. While `AUTOPANDAS` and `GAUSS` simply won't return a correct program if there is any error in the example or demonstration, `VIZSMITH` and `DATANA` offer some leeway due to the use of statistical models for generating code, which is inherently more robust to errors than symbolic methods. Still, it is important to highlight what parts of the query might be wrong and ask clarification from the user if necessary.

While `AUTOPANDAS` and `GAUSS` employ some form of ranking to order the generated code solutions, helping the users disambiguate and choose between multiple solutions is an open problem. Approaches such as generating distinguishing inputs to help understand the differences between solutions have been proposed. The challenge, however, lies in generating inputs that do not impose a cognitive burden on the user. Multi-modal specifications such as combining natural language and input-output examples, although not explored specifically in this dissertation, are another alternative for filtering unwanted solutions and have been successful in practice. Still, supporting more interesting multi-modal specifications, such as those involving an iterative mixing of direct manipulation or interaction alongside natural language or examples, is an open problem and is critical for supporting a wider variety of analyses in a useful manner.

Software development is inherently an iterative process, and programming assistants need to support an iterative workflow. Enabling users to provide feedback, especially in settings such as `VIZSMITH` and `DATANA` involving the use of natural language, is important for making assistants more useful. Note that our notion of feedback goes beyond simple positive or negative votes indicating whether the assistant did the right thing. Feedback must be leveraged in an *active* manner where the solutions are updated on the fly after entering into a dialogue with the user. As an example, imagine a user giving iterative feedback to incrementally construct a target visualization in `matplotlib`. Recent advances in natural language processing and in-context learning can help in realizing this active feedback loop.

It is equally important to give recommendations to help users who may be stuck. Auto-completion suggestions that inform users about the set of possible operations [104] can be useful. Going a step further, such auto-completion can also take existing code context into account. Such a mixed-initiative approach could also help with the disambiguation problem above, as users can use the completed queries as a means of choosing between options

immediately. In contrast, in the current implementation, users have to wait for DATANA to come back with suggestions given a single query crafted from scratch.

We are optimistic that programming assistants with a well-designed collection of high-level specifications, combined with smart algorithms and good interface design, can help raise the level of abstractions available to analysts, ultimately improving their productivity. We hope that the techniques and ideas contributed in this dissertation seed further research on building assistants with these qualities.

Bibliography

- [1] adamerose. *pandasgui*. URL: <https://github.com/adamerose/pandasgui>.
- [2] Eytan Adar. “GUESS: a language and interface for graph exploration”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006).
- [3] Hiralal Agrawal and Joseph R. Horgan. “Dynamic Program Slicing”. In: *SIGPLAN Not.* 25.6 (June 1990), pp. 246–256. ISSN: 0362-1340. DOI: 10.1145/93548.93576. URL: <https://doi.org/10.1145/93548.93576>.
- [4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. “Learning to Represent Programs with Graphs”. In: *International Conference on Learning Representations*. 2018. URL: <https://openreview.net/forum?id=BJOFETxR->.
- [5] Sara Alspaugh et al. “Futzing and Moseying: Interviews with Professional Data Analysts on Exploration Practices”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019), pp. 22–31. ISSN: 1077-2626. DOI: 10.1109/TVCG.2018.2865040. URL: <https://doi.org/10.1109/TVCG.2018.2865040>.
- [6] Alteryx Trifacta. *Trifacta Software*. URL: <https://trifacta.com>.
- [7] R. Alur et al. “Syntax-guided synthesis”. In: *Formal Methods in Computer-Aided Design*. Cham: Springer, Oct. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.
- [8] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. “Scaling enumerative program synthesis via divide and conquer”. In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 2017, pp. 319–336.
- [9] Giambattista Amati. “BM25”. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 257–260. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_921. URL: https://doi.org/10.1007/978-0-387-39940-9_921.
- [10] Jacob Austin et al. “Program synthesis with large language models”. In: *arXiv preprint arXiv:2108.07732* (2021).
- [11] Johannes Bader et al. “Getafix: Learning to Fix Bugs Automatically”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360585. URL: <https://doi.org/10.1145/3360585>.

- [12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: <http://arxiv.org/abs/1409.0473>.
- [13] Matej Balog et al. “DeepCoder: Learning to Write Programs”. In: *CoRR* abs/1611.01989 (2016). arXiv: 1611.01989. URL: <http://arxiv.org/abs/1611.01989>.
- [14] Matej Balog et al. “DeepCoder: Learning to Write Programs”. In: *CoRR* abs/1611.01989 (2016). arXiv: 1611.01989. URL: <http://arxiv.org/abs/1611.01989>.
- [15] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. “Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 613–624. ISBN: 9781450355728. DOI: 10.1145/3338906.3338952. URL: <https://doi.org/10.1145/3338906.3338952>.
- [16] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf>.
- [17] Rudy Bunel et al. “Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis”. In: *CoRR* abs/1805.04276 (2018). arXiv: 1805.04276. URL: <http://arxiv.org/abs/1805.04276>.
- [18] Sarah Chasins and Rastislav Bodik. “Skip Blocks: Reusing Execution History to Accelerate Web Scripts”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133875. URL: <https://doi.org/10.1145/3133875>.
- [19] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. “Rousillon: Scraping Distributed Hierarchical Web Data”. In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST ’18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 963–975. ISBN: 9781450359481. DOI: 10.1145/3242587.3242661. URL: <https://doi.org/10.1145/3242587.3242661>.
- [20] Souti Chattopadhyay et al. “What’s Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12. ISBN: 9781450367080. URL: <https://doi.org/10.1145/3313831.3376729>.
- [21] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv e-prints*, arXiv:2107.03374 (July 2021), arXiv:2107.03374. arXiv: 2107.03374 [cs.LG].
- [22] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. “Metamorphic testing: a new approach for generating next test cases”. In: *Technical Report HKUST-CS98-01*, (1998).

- [23] Yanju Chen, Ruben Martins, and Yu Feng. “Maximal Multi-Layer Specification Synthesis”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 602–612. ISBN: 9781450355728. DOI: 10.1145/3338906.3338951. URL: <https://doi.org/10.1145/3338906.3338951>.
- [24] Kyunghyun Cho et al. “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)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: <http://www.aclweb.org/anthology/D14-1179>.
- [25] Aakanksha Chowdhery et al. “PaLM: Scaling Language Modeling with Pathways”. In: *arXiv e-prints*, arXiv:2204.02311 (Apr. 2022), arXiv:2204.02311. arXiv: 2204.02311 [cs.CL].
- [26] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. URL: <https://doi.org/10.1145/1988042.1988046>.
- [27] Allen Cypher et al., eds. *Watch What I Do: Programming by Demonstration*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262032139.
- [28] Aditya Desai et al. “Program Synthesis Using Natural Language”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 345–356. ISBN: 9781450339001. DOI: 10.1145/2884781.2884786. URL: <https://doi.org/10.1145/2884781.2884786>.
- [29] Jacob Devlin et al. “RobustFill: Neural Program Learning under Noisy I/O”. In: *ICML 2017*. 2017. URL: <https://www.microsoft.com/en-us/research/publication/robustfill-neural-program-learning-noisy-io/>.
- [30] Dawn Drain et al. “Generating Code with the Help of Retrieved Template Functions and Stack Overflow Answers”. In: *arXiv e-prints*, arXiv:2104.05310 (Apr. 2021), arXiv:2104.05310. arXiv: 2104.05310 [cs.IR].
- [31] Ian Drosos et al. “Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: ACM, 2020, pp. 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376442. URL: <https://doi.org/10.1145/3313831.3376442>.
- [32] Fbdesignpro. *sweetviz*. URL: <https://github.com/fbdesignpro/sweetviz>.

- [33] Yu Feng et al. “Component-Based Synthesis for Complex APIs”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 599–612. ISBN: 9781450346603. DOI: 10.1145/3009837.3009851. URL: <https://doi.org/10.1145/3009837.3009851>.
- [34] Yu Feng et al. “Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 422–436. ISBN: 9781450349888. DOI: 10.1145/3062341.3062351. URL: <https://doi.org/10.1145/3062341.3062351>.
- [35] Yu Feng et al. “Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 422–436. ISSN: 0362-1340. DOI: 10.1145/3140587.3062351. URL: <http://doi.acm.org/10.1145/3140587.3062351>.
- [36] Yu Feng et al. “Program Synthesis Using Conflict-Driven Learning”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 420–435. ISBN: 9781450356985. DOI: 10.1145/3192366.3192382. URL: <https://doi.org/10.1145/3192366.3192382>.
- [37] Yu Feng et al. “Program Synthesis Using Conflict-driven Learning”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 420–435. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192382. URL: <http://doi.acm.org/10.1145/3192366.3192382>.
- [38] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Sept. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: <https://aclanthology.org/2020.findings-emnlp.139>.
- [39] A. Ferrari and M. Russo. *Introducing Microsoft Power BI*. Introducing. Pearson Education, 2016. ISBN: 9781509302758. URL: <https://books.google.com/books?id=U1qsDAAAQBAJ>.
- [40] Nat Friedman. 2021. URL: <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/>.
- [41] T.a Gao et al. “Datatone: Managing ambiguity in natural language interfaces for data visualization”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*. UIST '15 (2015), pp. 489–500. DOI: 10.1145/2807442.2807478. URL: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84958249800%7B%5C%26%7DpartnerID=40%7B%5C%26%7Ddmd5=f0eb3ceb834a66e6d0eb6b59ffc57163>.

- [42] Sumit Gulwani. “Automating String Processing in Spreadsheets Using Input-output Examples”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. URL: <http://doi.acm.org/10.1145/1926385.1926423>.
- [43] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010. URL: <http://dx.doi.org/10.1561/2500000010>.
- [44] Philip J. Guo et al. “Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST ’11. Santa Barbara, California, USA: Association for Computing Machinery, 2011, pp. 65–74. ISBN: 9781450307161. DOI: 10.1145/2047196.2047205. URL: <https://doi.org/10.1145/2047196.2047205>.
- [45] Pat Hanrahan. “VizQL: A Language for Query, Analysis and Visualization”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: Association for Computing Machinery, 2006, p. 721. ISBN: 1595934340. DOI: 10.1145/1142473.1142560. URL: <https://doi.org/10.1145/1142473.1142560>.
- [46] Yeye He et al. “Transform-data-by-example (TDE): An Extensible Search Engine for Data Transformations”. In: *Proc. VLDB Endow.* 11.10 (June 2018), pp. 1165–1177. ISSN: 2150-8097. DOI: 10.14778/3231751.3231766. URL: <https://doi.org/10.14778/3231751.3231766>.
- [47] Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. “Predictive Interaction for Data Transformation”. In: *CIDR*. 2015.
- [48] Joseph M Hellerstein, Jeffrey Heer, and Sean Kandel. “Self-Service Data Preparation: Research to Practice.” In: *IEEE Data Eng. Bull.* 41.2 (2018), pp. 23–34.
- [49] *How can I normalize data and create a stacked bar chart?* URL: <https://stackoverflow.com/questions/57337796/how-can-i-normalize-data-and-create-a-stacked-bar-chart> (visited on 04/22/2021).
- [50] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.
- [51] Hamel Husain et al. “CodeSearchNet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).

- [52] Naman Jain et al. “Jigsaw: Large Language Models Meet Program Synthesis”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1219–1231. ISBN: 9781450392211. DOI: 10.1145/3510003.3510203. URL: <https://doi.org/10.1145/3510003.3510203>.
- [53] Susmit Jha et al. “Oracle-guided Component-based Program Synthesis”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 215–224. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806833. URL: <http://doi.acm.org/10.1145/1806799.1806833>.
- [54] A. Kalyan et al. “Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples”. In: *ArXiv e-prints* (Apr. 2018). arXiv: 1804.01186 [cs.AI].
- [55] Sean Kandel et al. “Enterprise Data Analysis and Visualization: An Interview Study”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926. DOI: 10.1109/TVCG.2012.219.
- [56] Sean Kandel et al. “Wrangler: Interactive Visual Specification of Data Transformation Scripts”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, pp. 3363–3372. ISBN: 9781450302289. DOI: 10.1145/1978942.1979444. URL: <https://doi.org/10.1145/1978942.1979444>.
- [57] Jared Kaplan et al. “Scaling Laws for Neural Language Models”. In: *arXiv e-prints*, arXiv:2001.08361 (Jan. 2020), arXiv:2001.08361. arXiv: 2001.08361 [cs.LG].
- [58] Mary Beth Kery et al. “The Future of Notebook Programming Is Fluid”. In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383085. URL: <https://doi.org/10.1145/3334480.3383085>.
- [59] Meraj Khan et al. “Data Tweening: Incremental Visualization of Data Transforms”. In: *Proc. VLDB Endow.* 10.6 (Feb. 2017), pp. 661–672. ISSN: 2150-8097. DOI: 10.14778/3055330.3055333. URL: <https://doi.org/10.14778/3055330.3055333>.
- [60] D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. In: *ArXiv e-prints* (Dec. 2014). arXiv: 1412.6980.
- [61] T. Kluyver et al. “Jupyter Notebooks - a publishing format for reproducible computational workflows”. In: *ELPUB*. 2016.
- [62] Virtus Lab. *Pandas Stubs*. <https://github.com/VirtusLab/pandas-stubs>. 2021.

- [63] Vu Le and Sumit Gulwani. “FlashExtract: A Framework for Data Extraction by Examples”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 542–553. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594333. URL: <http://doi.acm.org/10.1145/2594291.2594333>.
- [64] Vu Le and Sumit Gulwani. “FlashExtract: A Framework for Data Extraction by Examples”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 542–553. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594333. URL: <http://doi.acm.org/10.1145/2594291.2594333>.
- [65] Doris Jung-Lin Lee et al. “Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows”. In: *Proc. VLDB Endow.* 15.3 (Nov. 2021), pp. 727–738. ISSN: 2150-8097. DOI: 10.14778/3494124.3494151. URL: <https://doi.org/10.14778/3494124.3494151>.
- [66] Woosuk Lee et al. “Accelerating Search-based Program Synthesis Using Learned Probabilistic Models”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 436–449. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192410. URL: <http://doi.acm.org/10.1145/3192366.3192410>.
- [67] Fei Li and H. V. Jagadish. “Constructing an Interactive Natural Language Interface for Relational Databases”. In: *Proc. VLDB Endow.* 8.1 (Sept. 2014), pp. 73–84. ISSN: 2150-8097. DOI: 10.14778/2735461.2735468. URL: <https://doi.org/10.14778/2735461.2735468>.
- [68] Fei Li and H. V. Jagadish. “Constructing an interactive natural language interface for relational databases”. In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 73–84. ISSN: 21508097. DOI: 10.14778/2735461.2735468. URL: <http://dl.acm.org/citation.cfm?doid=2735461.2735468>.
- [69] Yujia Li et al. “Gated Graph Sequence Neural Networks”. In: *CoRR* abs/1511.05493 (2015). arXiv: 1511.05493. URL: <http://arxiv.org/abs/1511.05493>.
- [70] Steve Lohr. “For big-data scientists, ‘janitor work’ is key hurdle to insights”. In: *New York Times* 17 (2014), B4.
- [71] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. “Show Me: Automatic Presentation for Visual Analysis”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1137–1144. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.70594. URL: <https://doi.org/10.1109/TVCG.2007.70594>.
- [72] Z. Manna and R. Waldinger. “Synthesis: Dreams \rightarrow Programs”. In: *IEEE Transactions on Software Engineering* SE-5.4 (1979), pp. 294–328. DOI: 10.1109/TSE.1979.234198.

- [73] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <http://tensorflow.org/>.
- [74] Mikael Mayer et al. “User interaction models for disambiguation in programming by example”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 2015, pp. 291–301.
- [75] Wes McKinney. “pandas: a Foundational Python Library for Data Analysis and Statistics”. In: *Python High Performance Science Computer* (Jan. 2011).
- [76] Microsoft. *Gated Graph Neural Network Samples*. <https://github.com/Microsoft/gated-graph-neural-network-samples>. Accessed October 17th, 2018. 2017.
- [77] Microsoft Corporation. *Microsoft Excel*. URL: <https://www.microsoft.com/en-us/microsoft-365/excel>.
- [78] Maxwell Nye et al. “Show Your Work: Scratchpads for Intermediate Computation with Language Models”. In: *arXiv e-prints*, arXiv:2112.00114 (Nov. 2021), arXiv:2112.00114. arXiv: 2112.00114 [cs.LG].
- [79] C. Olsten et al. “VIQING: visual interactive querying”. In: *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254)*. 1998, pp. 162–169. DOI: 10.1109/VL.1998.706159.
- [80] Michael Palmer. *Data is the New Oil*. https://ana.blogs.com/maestros/2006/11/data_is_the_new.html.
- [81] pandasprofiling. URL: <https://github.com/pandas-profiling/pandas-profiling>.
- [82] Yannis Papanikolaou. *Teach me how to Label: Labeling Functions from Natural Language with Text-to-text Transformers*. 2021. arXiv: 2101.07138 [cs.CL].
- [83] Emilio Parisotto et al. “Neuro-Symbolic Program Synthesis”. In: *ICLR 2017*. 2017. URL: <https://www.microsoft.com/en-us/research/publication/neuro-symbolic-program-synthesis-2/>.
- [84] Md Rizwan Parvez et al. “Retrieval Augmented Code Generation and Summarization”. In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 2719–2734. DOI: 10.18653/v1/2021.findings-emnlp.232. URL: <https://aclanthology.org/2021.findings-emnlp.232>.
- [85] DJ Patil. *Data Jujitsu*. ” O’Reilly Media, Inc.”, 2012.
- [86] Fabian Pedregosa et al. “Scikit-Learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.
- [87] F Pérez and BE Granger. *Computational Narratives as the Engine of Collaborative Data Science*. 2015.[Internet][cited 4 Oct 2018].

- [88] Daniel Perry et al. “VizDeck: Streamlining exploratory visual analytics of scientific data”. In: 2013.
- [89] Spreadsheet Planet. *How to Fill Blank Cells with Value above in Excel (3 Easy Ways)*. URL: <https://spreadsheetplanet.com/fill-blank-cells-with-value-above-in-excel/>.
- [90] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. “Program Synthesis from Polymorphic Refinement Types”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA: ACM, 2016, pp. 522–538. ISBN: 9781450342612. DOI: 10.1145/2908080.2908093. URL: <https://doi.org/10.1145/2908080.2908093>.
- [91] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. “Program Synthesis from Polymorphic Refinement Types”. In: *SIGPLAN Not.* 51.6 (June 2016), pp. 522–538. ISSN: 0362-1340. DOI: 10.1145/2980983.2908093. URL: <http://doi.acm.org/10.1145/2980983.2908093>.
- [92] Oleksandr Polozov and Sumit Gulwani. “FlashMeta: A Framework for Inductive Program Synthesis”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 107–126. ISBN: 9781450336895. DOI: 10.1145/2814270.2814310. URL: <https://doi.org/10.1145/2814270.2814310>.
- [93] V. Raman and J.M. Hellerstein. “Potter’s wheel: An interactive data cleaning system”. In: *Proceedings of the international conference on Very Large Data Bases*. 2001, pp. 381–390.
- [94] Bernadette M Randles et al. “Using the Jupyter notebook as a tool for open science: An empirical study”. In: *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE. 2017, pp. 1–2.
- [95] Tye Rattenbury et al. *Principles of Data Wrangling: Practical Techniques for Data Preparation*. 1st. O’Reilly Media, Inc., 2017. ISBN: 1491938927.
- [96] 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*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 419–428. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594321. URL: <http://doi.acm.org/10.1145/2594291.2594321>.
- [97] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. 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)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3982–3992. DOI: 10.18653/v1/D19-1410. URL: <https://aclanthology.org/D19-1410>.

- [98] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: <http://arxiv.org/abs/1908.10084>.
- [99] Acumen Research and Consulting. *Data Preparation Tool Market*. 2021. URL: <https://www.acumenresearchandconsulting.com/data-preparation-tools-market>.
- [100] Reudismam Rolim et al. “Learning Syntactic Program Transformations from Examples”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 404–415. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.44. URL: <https://doi.org/10.1109/ICSE.2017.44>.
- [101] Arvind Satyanarayan et al. “Critical Reflections on Visualization Authoring Systems”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 461–471. DOI: 10.1109/TVCG.2019.2934281.
- [102] Brian Scally. *How to make a correlation matrix in Tableau*. URL: <https://www.thedataschool.co.uk/brian-scally/how-to-make-a-correlation-matrix-in-tableau>.
- [103] Vidya Setlur et al. “Eviza: A Natural Language Interface for Visual Analysis”. In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST ’16* (2016), pp. 365–377. DOI: 10.1145/2984511.2984588. URL: <http://doi.acm.org/10.1145/2984511.2984588>.
- [104] Vidya Setlur et al. “Sneak Pique: Exploring autocompletion as a data discovery scaffold for supporting visual analysis”. In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 966–978.
- [105] Helen Shen. “Interactive notebooks: Sharing the code”. In: *Nature* 515.7525 (2014), pp. 151–152.
- [106] Kensen Shi, David Bieber, and Rishabh Singh. “TF-Coder: Program Synthesis for Tensor Manipulations”. In: *ACM Trans. Program. Lang. Syst.* 44.2 (May 2022). ISSN: 0164-0925. DOI: 10.1145/3517034. URL: <https://doi.org/10.1145/3517034>.
- [107] Rishabh Singh and Armando Solar-Lezama. “Synthesizing Data Structure Manipulations from Storyboards”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 289–299. ISBN: 9781450304436. DOI: 10.1145/2025113.2025153. URL: <https://doi.org/10.1145/2025113.2025153>.
- [108] Calvin Smith and Aws Albarghouthi. “MapReduce Program Synthesis”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA: ACM, 2016, pp. 326–340. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908102. URL: <http://doi.acm.org/10.1145/2908080.2908102>.

- [109] Armando Solar-Lezama et al. “Combinatorial Sketching for Finite Programs”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 404–415. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168907. URL: <http://doi.acm.org/10.1145/1168857.1168907>.
- [110] *Stacked bar chart*. URL: https://matplotlib.org/stable/gallery/lines_bars_and_markers/bar_stacked.html (visited on 04/22/2021).
- [111] Chris Stolte, Diane Tang, and Pat Hanrahan. “Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases”. In: *IEEE Transactions on Visualization and Computer Graphics* 8.1 (Jan. 2002), pp. 52–65. ISSN: 1077-2626. DOI: 10.1109/2945.981851. URL: <https://doi.org/10.1109/2945.981851>.
- [112] Chris Stolte, Diane Tang, and Pat Hanrahan. “Query, Analysis, and Visualization of Hierarchically Structured Data Using Polaris”. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’02. Edmonton, Alberta, Canada: Association for Computing Machinery, 2002, pp. 112–122. ISBN: 158113567X. DOI: 10.1145/775047.775064. URL: <https://doi.org/10.1145/775047.775064>.
- [113] Chengnian Sun et al. “Perses: Syntax-Guided Program Reduction”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 361–371. ISBN: 9781450356381. DOI: 10.1145/3180155.3180236. URL: <https://doi.org/10.1145/3180155.3180236>.
- [114] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’14. Montreal, Canada: MIT Press, 2014, pp. 3104–3112.
- [115] Tableau. *Tableau Software*. URL: <https://tableau.com>.
- [116] *The Kaggle Data-Science Platform*. URL: <https://www.kaggle.com/>.
- [117] *The Mypy Optional Static Type Checker*. URL: <http://mypy-lang.org/>.
- [118] *The pandas project*. <https://pandas.pydata.org>. Accessed October 11th, 2018. 2014.
- [119] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models”. In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA ’22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: 10.1145/3491101.3519665. URL: <https://doi.org/10.1145/3491101.3519665>.

- [120] Ashish Vaswani et al. “Attention is All You Need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.
- [121] *Voice Call Quality Customer Experience*. URL: <https://data.gov.in/catalog/voice-call-quality-customer-experience>.
- [122] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. “Synthesizing Highly Expressive SQL Queries from Input-Output Examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 452–466. ISBN: 9781450349888. DOI: 10.1145/3062341.3062365. URL: <https://doi.org/10.1145/3062341.3062365>.
- [123] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. “Synthesizing Highly Expressive SQL Queries from Input-output Examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 452–466. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062365. URL: <http://doi.acm.org/10.1145/3062341.3062365>.
- [124] Chenglong Wang et al. “Falx: Synthesis-powered visualization authoring”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–15.
- [125] Chenglong Wang et al. “Visualization by Example”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371117. URL: <https://doi.org/10.1145/3371117>.
- [126] Xinyu Wang, Isil Dillig, and Rishabh Singh. “Program Synthesis Using Abstraction Refinement”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158151. URL: <https://doi.org/10.1145/3158151>.
- [127] Xinyu Wang, Isil Dillig, and Rishabh Singh. “Program Synthesis Using Abstraction Refinement”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 63:1–63:30. ISSN: 2475-1421. DOI: 10.1145/3158151. URL: <http://doi.acm.org/10.1145/3158151>.
- [128] Xinyu Wang et al. “Learning Abstractions for Program Synthesis”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 407–426. ISBN: 978-3-319-96145-3.
- [129] Elaine Weyuker. “On Testing Non-Testable Programs”. In: *Computer Journal* 25 (Nov. 1982). DOI: 10.1093/comjnl/25.4.465.
- [130] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. “Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study”. In: *ArXiv abs/1911.00568* (2019).
- [131] Kanit Wongsuphasawat et al. “Voyager 2: Augmenting Visual Analysis with Partial View Specifications”. In: *ACM Human Factors in Computing Systems (CHI)*. 2017. URL: <http://idl.cs.washington.edu/papers/voyager2>.

- [132] Kanit Wongsuphasawat et al. “Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations”. In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2016). URL: <http://idl.cs.washington.edu/papers/voyager>.
- [133] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. “B2: Bridging Code and Interactive Visualization in Computational Notebooks”. In: *ACM User Interface Software & Technology (UIST)*. 2020. DOI: 10.1145/3379337.3415851. URL: <http://vis.csail.mit.edu/pubs/b2>.
- [134] Yuhao Wu et al. “How Do Developers Utilize Source Code from Stack Overflow?”. In: *Empirical Softw. Engg.* 24.2 (Apr. 2019), pp. 637–673. ISSN: 1382-3256. DOI: 10.1007/s10664-018-9634-5. URL: <https://doi.org/10.1007/s10664-018-9634-5>.
- [135] Frank F Xu et al. “Incorporating External Knowledge through Pre-training for Natural Language to Code Generation”. In: *Annual Meeting of the Association for Computational Linguistics*. ACL. ACL, 2020, pp. 6045–6052. DOI: <http://dx.doi.org/10.18653/v1/2020.acl-main.538>.
- [136] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. “In-IDE Code Generation from Natural Language: Promise and Challenges”. In: *ACM Trans. Softw. Eng. Methodol.* TOSEM 31.2 (2022). ISSN: 1049-331X. DOI: <http://dx.doi.org/10.1145/3487569>. URL: <https://doi.org/10.1145/3487569>.
- [137] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. “Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example”. In: *Proc. VLDB Endow.* 11.5 (Jan. 2018), pp. 580–593. ISSN: 2150-8097. DOI: 10.1145/3187009.3177735. URL: <https://doi.org/10.1145/3187009.3177735>.
- [138] Navid Yaghmazadeh et al. “SQLizer: Query Synthesis from Natural Language”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133887. URL: <https://doi.org/10.1145/3133887>.
- [139] Kuat Yessenov et al. “A Colorful Approach to Text Processing by Example”. In: *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. UIST ’13. St. Andrews, Scotland, United Kingdom: Association for Computing Machinery, 2013, pp. 495–504. ISBN: 9781450322683. DOI: 10.1145/2501988.2502040. URL: <https://doi.org/10.1145/2501988.2502040>.
- [140] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 183–200. ISSN: 0098-5589. DOI: 10.1109/32.988498. URL: <https://doi.org/10.1109/32.988498>.
- [141] Victor Zhong, Caiming Xiong, and Richard Socher. “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning”. In: *arXiv e-prints*, arXiv:1709.00103 (Aug. 2017), arXiv:1709.00103. arXiv: 1709.00103 [cs.CL].

- [142] Moshé M. Zloof. “Query-by-Example: The Invocation and Definition of Tables and Forms”. In: *Proceedings of the 1st International Conference on Very Large Data Bases*. VLDB '75. Framingham, Massachusetts: Association for Computing Machinery, 1975, pp. 1–24. ISBN: 9781450339209. DOI: 10.1145/1282480.1282482. URL: <https://doi.org/10.1145/1282480.1282482>.