Interactive Data Profiling

Will Epperson

June 2025

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Adam Perer, Co-Chair, CMU HCII
Dominik Moritz, Co-Chair, CMU HCII & Apple
Sherry Tongshuang Wu, CMU HCII
Aniket Kittur, CMU HCII
Gagan Bansal, Microsoft Research

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

Copyright © Will Epperson 2025

Keywords

Data Visualization, Data Science, Automatic Data Visualization, Exploratory Data Analysis, Data Profiling, Data Quality, Tabular Datasets, Text Datasets

Abstract

Data has been a key driver behind recent advances in science, engineering, and artificial intelligence. As datasets have grow larger and more complex, the primary bottleneck has shifted from access to data towards the human effort required to interpret it. Human expertise is essential to understand datasets, however generating this understanding during analysis remains a time-consuming and manual process. Many AI modeling failures are, at their core, data problems—issues that might have been addressed earlier with better tools for understanding the data. Data visualization facilitates understanding through visual representations, however existing approaches to visual data exploration introduce friction that slows users down, requiring manually defining charts and interactions through code or context switching to a new analysis tool. *How can we build flexible and lightweight systems to help people more quickly understand their data?*

This thesis develops systems for **Interactive Data Profiling** that accelerate data exploration through lightweight and interactive interfaces that fit into current analysis workflows. We first motivate this problem through a large scale interview study and survey of data scientists that reveals the potential for tools to help users manage the repetitive code used for data profiling. We then discuss the design, implementation, and evaluation of three systems that develop the approach of interactive data profiling. First, we describe AUTOPROFILER, a system that augments programming environments with automatic data profiles that show summaries of the data in memory and update as a user programs. We then extend this approach with SOLAS which tracks the history of a user's analysis code to create data profiles adapted to the current task and user interest. User evaluations demonstrate how the lightweight visualizations and fast feedback loops enabled by these systems help users quickly identify important patterns and data quality issues. Finally, we present TEXTURE, a general-purpose text exploration tool that enables users to iterate on attributes for describing their text and then explore results in the interactive UI. Expert user studies show how TEXTURE enables more efficient exploration and helps users uncover new insights from their text datasets.

Together, these tools describe how to situate interactive data profiling within data science workflows to enable a fast feedback loop between manipulating data and inspecting the results. As data remains an increasingly important component of modern work, interactive data profiling systems can play a critical role in enabling faster, more reliable understanding of the data behind models and decisions.

Acknowledgements

Forthcoming...

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's data-rich world, more and more applications are powered by insights derived from data or AI models trained on vast amounts of data. Data drives treatment decisions in healthcare, informs news stories in data journalism, helps businesses understand current and future trends, and has fueled recent advancements in Artificial Intelligence (AI). However, this proliferation of data means that it is increasingly difficult to actually make sense of it. As Hebert Simon pithily observed: "A wealth of information creates a poverty of attention" [149].

This observation still rings true today when analyzing and modeling data. Data is overabundant, yet high-quality data can be scarce. This combination makes it hard to appropriately direct attention when working with large datasets. For example, prior research has shown that many popular AI model benchmark and training datasets contain label errors, duplicates, and ambiguous data points that make evaluation difficult [35, 72, 155]. Other estimates describe identifying and cleaning dirty data as 80% of the cost of data warehousing projects [26].

The sheer quantity of data available for analysis and potentially poor quality mean that it is critical that users have tools that can help them properly explore and understand their data. Despite advancements in automated techniques for detecting quality issues and mining insights from data, human judgment remains essential for interpreting data within specific analytical contexts. Users must apply their domain and contextual knowledge to make sense of distributions and handle issues like outliers, poorly labeled instances, or missing data [76].

Dealing with quality issues and exploring results are all part of the process of Exploratory Data Analysis (EDA). EDA involves examining sample values, statistics, and summary visualizations to profile a dataset and discover data insights that can guide next steps [156, 165]. Importantly, EDA not only happens once at the beginning of an analysis but occurs iteratively as users manipulate, clean, and add new results to their data [5, 75, 165]. After each iteration, users must re-*profile* their data to understand the latest version.

While modern data programming and visualization tools have become increasingly performant and expressive, users must still manually create visualizations and summaries to profile their data. Open-source libraries make it possible to create custom data transformations and visualizations for exploring datasets [119, 142]. However, leveraging these tools effectively requires significant time and expertise. Surveys of data scientists indicate that nearly half of their time is spent on data cleaning and visualization alone [8]. Since understanding datasets is difficult and time-consuming, this critical step is often bypassed in favor of downstream tasks such as model training or statistical testing. This neglect can lead to harmful data cascades, where initial issues compound, ultimately compromising model performance in high-stakes domains like healthcare and loan allocation [141].

1.1 Interactive Data Profiling

In this thesis, we investigate how to design tools that help users quickly explore, profile, and understand their data. Data science workflows are inherently iterative: users repeatedly acquire new data, clean it, train models, examine results, and revisit earlier steps as new insights emerge [5, 75, 165]. Yet existing data exploration tools are not optimized for these iterative workflows. Many systems operate as standalone applications, requiring users to repeatedly export data from their programming environments, analyze results externally, and then return to programming. Even within programming environments, manually creating visualizations after each step introduces friction and delays feedback. This thesis explores how to design integrated data exploration tools that streamline iteration, enabling users to more easily profile each version of their dataset throughout analysis (see Figure 1.1). This integrated approach reduces the feedback loop between data manipulation and understanding, ultimately facilitating more effective data analysis.

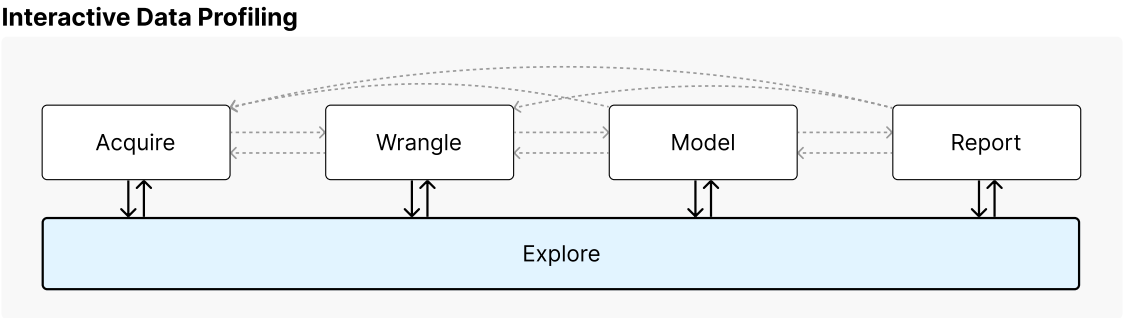

Figure 1.1: Traditional data workflows emphasize the iterative nature of working with data; however, tools are still designed for exploration as a stand-alone step. **Interactive data profiling** considers how to to design data exploration tools that are easily accessible from each step in analysis to speed up the feedback loop of data understanding.

We discuss the design, implementation, and evaluation of three different systems that develop the approach of interactive data profiling. These systems focus on increasingly **dynamic** forms of data profiles—first offering the same profile for every dataset in memory, then presenting data profiles that adapt based on historical data interactions, and finally offering an approach for flexibly profiling data attributes for unstructured data as they are defined by a user.

We first describe how tabular data profiling tools can be designed to offer faster feedback as users program with their data. Interactive programming environments like computational notebooks have become the tool of choice for data science because they enable the type of iterative and incremental programming that is characteristic of data science [140]. Accordingly, visualization tools have been developed to help users visualize their data in computational notebooks [96, 120, 138]. However, we find that these tools fail to offer the fast feedback necessary to truly enable rapid EDA while programming with data. We therefore present the design of two systems designed to speed up the feedback cycle while notebook programming with tabular datasets: AUTOPROFILER and SOLAS. AUTOPROFILER integrates automatic data profiles into computational notebooks that automatically

update after each interaction with the data. Through these data profiles, users can quickly inspect a summary of each dataset in memory then drill down into individual columns and insights. SOLAS then explores how to make these data profiles more dynamic by considering the *provenance* of the dataset and tracking the history of a user's interactions to adapt data profiles to the current task and user's interest. We discuss the designs of these systems, their implementation, and user studies that show how they make it easier for data scientists to understand their tabular datasets while programming by automating many of the repetitive steps of manual EDA.

The final part of this thesis extends these ideas beyond tabular data to support the exploration of unstructured text corpora. Understanding collections of text documents is essential in fields from computational social science to NLP, and requires first defining meaningful representations—such as words, phrases, topics, or classes—and then building visualizations to profile these attributes. In current workflows, users must manually construct each profiling visualization, making it difficult to link insights across views through interaction and slowing the feedback cycle in analysis

To address these challenges, we introduce TEXTURE, which enables fast profiling and interactive exploration of text datasets. TEXTURE provides a configurable data schema for categorizing descriptive text attributes and an interactive interface for exploring them. Building on the workflows developed in the previous systems, TEXTURE adds an additional layer by allowing users to explore different structured representations of their text data within the same interactive environment. We present the design and implementation of TEXTURE and report results from a user study with expert users. Our evaluation shows that TEXTURE is both expressive across domains and effective in helping users uncover new insights in their text data.

1.2 Thesis Statement

This thesis hypothesizes that interactive systems for data profiling can accelerate users' understanding of their data and help them effectively explore datasets and discover issues. Interactive data profiling tools create a fast feedback loop between interacting with data and visualizing results by automatically generating visual profiles within programming environments that update as users code, adapting profiles based on users' analysis history, and enabling users to define custom attributes from unstructured text and interactively explore the resulting visual data profiles.

1.3 Thesis Overview

Chapter 2 discusses background and prior research on interactive data visualization, exploratory analysis, and data profiling tools.

Chapter 3 presents a motivational interview and survey study on how developers manage and reuse the code they write for data analysis. The results of this study reveal different strategies developers use to re-purpose previously developed code for new tasks, including difficulties with reusing code for common repetitive tasks like exploratory data analysis.

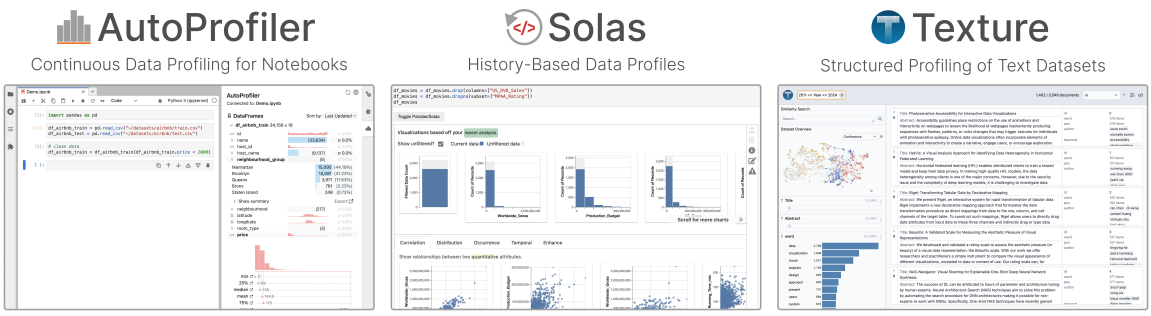

Figure 1.2: An overview of the systems presented in this thesis.

We discuss how the findings from this study inform the design of tools to help people save time during analysis by speeding up exploratory analysis.

Chapter 4 presents AUTOPROFILER, a system that augments computational notebooks with a live data profiling sidebar that automatically profiles data in memory and updates when the data changes. This design enables users to rapidly see an overview of their dataset as they iterate during analysis, speeding up the feedback loop between making changes and verifying the results. We present the results from a lab study and longitudinal case study evaluating how scientists use AUTOPROFILER during their work, finding that AUTOPROFILER helps users learn key information about their datasets without having to write any extra code and enables users to notice unexpected patterns in their data.

Chapter 5 describes how SOLAS extends the ideas of tabular notebook profiling presented in AUTOPROFILER by incorporating a user's analysis history through the code they write. Tracking analysis history enables task-specific data overviews and ranking overview visualizations according to a user interest model in each column. We discuss how SOLAS models user interest by tracking the history of analysis code and the results from a user evaluation that shows that users find the task-specific visualizations enabled by SOLAS to be more helpful than a baseline.

Chapter 6 then discusses how to develop interactive data profiling tools for text data. While AUTOPROFILER and SOLAS support exploration of structured tabular data, many critical workflows involve unstructured data such as text. Text exploration poses distinct challenges since the data cannot be easily profiled into a single overview visualization. To address this, we frame text exploration as a structured data problem, where arbitrary descriptive attributes derived from text—such as words, phrases, topics, or embeddings—can guide interactive exploration. We present TEXTURE, an interactive system that enables users to define these structured attributes through a configurable schema. TEXTURE then automatically visualizes structured attributes in interactive charts linked back to the text, allowing users to rapidly filter, explore, and form new hypotheses. Through an expert user study with real-world datasets, we demonstrate how TEXTURE shortens the feedback loop between hypothesis formulation and validation, enabling users to uncover critical quality issues in their datasets and learn new things even in datasets they had previously analyzed.

Chapter 7 concludes this thesis with a discussion of impact and future research opportunities.

1.4 Prior Publications and Authorship

While I am the principal author of this research, this thesis is the result of years of collaboration with my advisors Dominik Moritz and Adam Perer along with co-authors on each individual paper. At the beginning of each chapter, I include references to prior publications.

Chapter 2
Background & Related Work

This chapter discusses background material and related research. First, we discuss the users who work with data and their tasks. We then motivate the importance of data understanding for data analytics and AI workflows. We conclude with an overview of prior systems to help users explore data while programming.

2.1 Who Programs with Data and What are They Doing?

Many different kinds of users program with data. Several previous researchers have described these users, their goals, and needs. Early descriptions of data scientists describe how the most universal skill among data scientists is the ability to write code to extracting meaning and insight from data [29]. End user programming has long studied users who do "programming to achieve the result of a program primarily for personal rather than public use" [89]. This is distinct from professional developers like software engineers since the code is a means to achieve an end, rather than the end in itself. Guo similarly describes *research programmers* as users who are "writing programs to get insights from data" [56]. Another definition comes from Kery who describes *exploratory programming* as a "programming task in which a specific goal or a means to that goal must be discovered by iteratively writing code for multiple ideas" [79]. This builds on earlier definitions that describe the iterative software support needed for AI development [145].

These prior characterizations have several elements in common. First, for data scientists, programming is a critical tool but is a means to an end rather than the primary objective. This leads to the second observation that iteration is key when working with data. Data scientists iterate on their data, on visualizations, on models, and on results to explore different ways of making sense of and using their data. The importance of iteration influences the choice of tools that data scientists use to work with their data (like computation notebooks) and also the design of Interactive Data Profiling tools that must work within highly iterative workflows.

Iteration is a central element of prior task frameworks that describe the workflows and tasks data scientists go through when developing machine learning models or using visualizations to understand their data. Piroulli and Card describe the process of data science as a sensemaking activity [127]. Data analysis starts from raw data which is then filtered to find evidence. This evidence is organized into schemas and hypotheses and finally evaluated for presentation. They describe the challenges in finding relevant information, along with ensuring that evidence is wholistically considered.

Kandel *et al.* [75] describe a five step process for enterprise data analysis and visualization consisting of data discovery, wrangling, profiling, modeling, and reporting. In their definition, data profiling refers to understanding a dataset, potential quality issues, and as-

sumptions they can make about their dataset such as each attribute's distribution. Later task models describe the Exploratory Data Analysis (EDA) process with very similar steps, and emphasize the iterative nature of analysis [5, 165]. While working with data, analysts continually iterate between these different steps as they learn about their data and explore different paths. Data exploration and understanding is not an atomic step of this process but something that occurs continually throughout analysis [5]. Similar breakdowns of the process of developing Machine Learning or AI models emphasize the iterative nature of the process as developers define a model goal, prepare their data, train the model, evaluate and then iterate [6]. The Interactive Data Profiling tools presented in this thesis are designed for iterative analysis. They focus on providing a fast feedback loop between writing code to manipulate data and seeing visual representations of the results so that users can spend less time transitioning between activities and more time making progress in analysis.

Prior research has also studied how developers debug their programs, a process involving both program comprehension [137] and sensemaking to fix issues [52]. Similarly, when working with data, practitioners must first understand what is in a dataset and then investigate any resulting issues. Many of the difficulties in making sense of software also apply to data, as developers often struggle to connect abstract observations to concrete fixes. Techniques aiding program comprehension and debugging, such as explanatory debugging [88], have informed approaches to help users interpret data-intensive systems like machine learning models [91]. Data programming includes additional challenges beyond traditional programming because it demands both software engineering and data comprehension skills [6, 66]. While program correctness can often be objectively measured (e.g., through tests), data quality may depend on context and can only be evaluated by the end user performing the data work in their context [54].

2.2 Understanding Datasets Through Visualization

Ensuring data quality and understanding what is present or missing in a dataset are fundamental to analytics and modeling. Failing to fully understand the data introduces risks. Recent research has highlighted how with the increasing emphasis on developing AI models, people often undervalue data quality which compounds into negative downstream effects like poorly performing models in deployment [141]. As the popular "garbage in, garbage out" adage for machine learning warns: poor data yields poor outcomes.

However, dataset issues are often not discovered until after an analysis is completed or a model deployed, affecting both industrial settings and widely used benchmark datasets. For example, Swayamdipta *et al.* [155] found that many difficult instances in Natural Language Processing (NLP) benchmarks stem from mislabeled data, and similar issues appear in Computer Vision benchmarks, where the only remaining errors in some benchmark datasets are largely because of label errors [159].

Data visualization can help people understand their data and spot potential issues by engaging the human perceptual system [21]. As Shneiderman says: "the purpose of visualization is insight, not pictures" [70], reflecting the use of visualization to offer deeper insight into the trends or observations that lie within a dataset. These insights can take many forms, from "Aha!" moments when a user finds something surprising in their data,

to more rote observations like understanding the distribution of an attribute or the presence of outliers [12, 19, 20]. Data insight often involves connecting data facts with the domain knowledge that an analyst already has [20, 100]. This domain knowledge and the context in which data is used is critical for a user to make a decision on if an observation is actually a meaningful insight or error.

Using visualizations to understand a dataset has been a longstanding best practice. Tukey was an early advocate for plotting distributions and summary statistics through Exploratory Data Analysis (EDA) to get to know your data before confirmatory analysis (hypothesis testing) begins [156, 157]. Traditional approaches for Exploratory Data Analysis (EDA) and Visual Analytics emphasize the importance of exploring datasets throughout analysis and using visualizations combined with data mining algorithms to understand data [78, 156]. Current best practices taught in university statistics courses still emphasize the importance of starting analysis with summaries of individual columns, such as distributions and descriptive statistics, before moving on to plot combinations of columns or investigating correlations [143]. However, little research has explored how to best design tools that integrate data exploration directly into the programming workflows where the majority of data work happens.

2.3 Automatic Data Visualization and Profiling

Despite the importance of data understanding and visualization, it still proves challenging and time consuming for users. Multiple surveys of production data scientists routinely describe the difficulty and time spent on data understanding, profiling, and wrangling [8, 75, 85]. For example, a recent Anaconda foundation survey described that data scientists self-reported spending almost 50% of their time on data cleaning and visualization [8].

Visualization recommendation systems aim to automate the visual presentation of data to speed up the data understanding process. Visualization recommendation typically has two goals: (1) helping analysts follow best practices by creating visualizations that are both expressive and effective, and (2) removing the tedium of crafting visualizations to make the exploration process faster and more robust [59]. Some systems automate visual presentation and then rank charts according to metrics of interest such as high correlation [32], charts that satisfy a particular pattern in the data [147], or emphasize "data variation over design variation" to explore a wide range of attribute combinations [167, 168]. Other tools explicitly consider data quality in visualization like the Profiler system which checks data for common quality issues such as missing data or outliers, and presents charts to the user in *data profiles* that highlight potential issues [76]. We adopt this term throughout this thesis to refer to any data overview visualization as a data profile.

Many of the visual analytics systems for exploration follow Schneiderman's Visual Information Seeking Mantra: "Overview first, zoom and filter, then details-on-demand" [146]. This presents a common interaction paradigm where systems can first provide an overview, then support follow up queries through *interaction* (zoom, filter, details). The Interactive Data Profiling tools in this thesis incorporate these ideas for data programming workflows. They show data overviews quickly and easily, then allow users to ask subsequent questions through interaction (either through code or through the GUI).

2.4 Meeting Users Where They Work: Supporting Data Programming

While many visualization recommendations tools provide algorithms and interactions to facilitate data exploration, they were not designed for a programming centric workflow, making it hard for data analysts to incorporate them into their existing tool stack [5]. For data work, this means integrating into the Python and Jupyter ecosystems.

Python has become the dominant programming language for data work, overtaking JavaScript as the most popular language on Github (a code sharing website) in 2024 [150]. This surge in popularity can be attributed, in part, to widely used open-source Python libraries for data science, such as Pandas for data manipulation [119], PyTorch for AI model training [24], and Jupyter for computational notebook programming [129]. Computational notebooks provide users with an interactive and iterative programming environment that is highly conducive to the iterative work of data analysis and visualization [123, 140]. Prior research has explored how to make data programming easier in computational notebooks from helping users track versions of their analysis [79] to cleaning up the messy code in notebooks [58].

Researchers have begun to explore how to integrate visualization recommendations into notebook-based programming. Lux [96] and other open-source tools [14, 55, 120, 138] provide on-demand exploratory data analysis (EDA) information for individual Pandas dataframes. Meanwhile, other tools examine how to balance interacting with data either through code or a graphical user interface (GUI). Although programming languages are flexible and expressive, GUIs are often more responsive and easier to use [5]. Some prior notebook systems have attempted to bridge this gap by automatically writing interactions made through charts [170] or widgets [82] back into the notebook environment.

The Interactive Data Profiling tools introduced in this thesis build on these earlier efforts, offering automatic data visualization capabilities both within the notebook environment and Python programming workflows. AUTOPROFILER and SOLAS focus on delivering rapid feedback during notebook programming, enabling a tight feedback loop between writing code and inspecting results. TEXTURE helps users understand unstructured text datasets in Python workflows, where the notion of an "overview visualization" requires iterating on different summary attributes to profile.

2.5 Interactively Exploring Data

Prior research has explored enabling dataset interactions to support follow-up analysis. Data visualization frameworks like Vega-Lite allow users to make chart selections that filter other visualizations [142]. Similarly, Mosaic expresses interactions as SQL predicates to support large-scale dataset interactions [61]. Other work links data interactions back to programming environments, such as B2 [170], which persists chart interactions as notebook code, and Mage [82], which provides an API for interactions across notebook GUIs and code. This thesis builds on these patterns to support interactions with data profiles beyond the initial overview. Both AUTOPROFILER and SOLAS focus on user interactions through code to manipulate data before profiling the results. AUTOPROFILER includes interactions

similar to Mage, writing code to the notebook on the users behalf based on the data profile. TEXTURE supports interactions in the interface for cross-filtering data profile charts.

Chapter 3

Motivating Study on Coding Practices in Data Science

This chapter is adapted from the following published paper:

> [38] Will Epperson, April Yi Wang, Robert DeLine, and Steven M. Drucker. "Strategies for reuse and sharing among data scientists in software teams". *ACM ICSE-SEIP*. 2022.

3.1 Summary

This chapter describes an interview study and survey that explore how data scientists reuse and share the code they write for analysis. Since data science relies on code closely coupled with data, we investigated how data scientists work with this data-centric code, which tasks are frequently reused in future analyses, and how tools might support repetitive programming tasks involving data. Our findings identify five strategies that data scientists employ to manage their data-centric code, including the practice of using *template notebooks*—pre-existing scripts or notebooks that serve as customizable starting points for new tasks. These notebooks are useful for repetitive workflows like exploratory analysis, where users want to ask similar questions or generate similar visualizations, even when the underlying data schema changes for a new dataset. This pattern highlights the need for lightweight overview tools to address common analysis questions and help users quickly understand a dataset's contents. In this chapter, we detail the methods and findings from our interviews (N=17) and survey (N=132) conducted with data scientists at Microsoft.

3.2 Introduction

As software engineering developed into a mature discipline, the ability to effectively share and reuse code has become a critical factor for success [46, 53, 87, 90]. Particularly as organizations grow, information management becomes both more difficult and more important. This information takes the form of actual source code but also the documentation for this code, specifications around the problem the code was initially created to solve, and who to talk to in an organization to learn more about the code. Well executed software reuse leads to fewer problems in code, less effort spent correcting problems, and higher developer productivity [110]. Technologies such as version control [48] have become commonplace to help keep track of versions between files and principles like "DRY" (Don't Repeat Yourself) are baked into software developers' minds.

However, the field of data science presents new and unique challenges in terms of sharing and reuse. The *people* writing the code come from different backgrounds, the *code* itself lives in a variety of formats including raw text files, computational notebooks, and ad hoc queries, and *data* permeates the entire analysis process.

Data scientists are a relatively recent role on software development teams, and they work alongside established roles like software developers, operations, and program managers [13, 84, 86, 107, 108]. In the industrial setting, data scientists are often not directly responsible for the data they analyze. Their partners, the data engineers, collect, store, and maintain datasets that data scientists access for their analysis [84, 86]. For teams whose services use machine learning (ML), data engineers also deploy, scale out, and maintain ML models that data scientists create [6]. Further, some data scientists work on their own team's data, some act as a centralized service working with several product teams, and some act as consultants working with third-party companies.

A data scientist's work is often exploratory or ad hoc in nature and involves using data to craft analyses, models, and visualizations that are reported outside of the coding environment [86]. At Microsoft, each data science team is free to go about this process however they see fit which leads to a wide variety of approaches. To scale model and inference pipelines into production, this process is handed off to adjacent data engineers.

This unique role for data scientists has led to new approaches and problems for code reuse and sharing. The exploratory, open-ended nature of data science coding impacts the incentives for investing time into reuse. For instance, if code is only used to answer a one-off analysis question, there is less incentive to invest the time into making this a reusable function. Furthermore, reusable code for data analysis must support customization and adaptation since each data analysis is slightly unique.

In this work, we specifically focus on the reuse of analysis *code* as it relates to data science work. This code is almost always coupled with data assets for the analysis, however we consider the versioning and reuse of data itself outside the scope of our study. Decisions about the team's data management are often made at the team level and are subject to corporate policies, customer agreements, laws and regulations. Data scientists typically enjoy more agency over the data analysis code than over the data itself. Hence, we focus on their work practices and pain points around the data analysis code as a topic where the research community can usefully intervene. Additionally, we focus on the reuse of code developed within teams or organizations rather than external library use.

By "reuse" we refer specifically to the consumption of code or other artifacts from previous work. This might be the author of that work reusing their own past analysis or that of someone else. On the other hand, by "sharing" we refer to the production of code or other artifacts for oneself and then providing it to someone else for another task. The acts of sharing and reuse might be viewed as two sides of the same coin; unique practices exist for both facets.

To provide a better understanding of how data scientists go about both reusing and sharing past work, we conducted interviews with professional data scientists to understand their current practices related to sharing and reuse of analysis. From these interviews we synthesized five different strategies for sharing and reuse that we developed into a survey to understand how these approaches generalize across a larger population. In the remainder of this chapter, we first present related work regarding sharing and reuse in data science,

followed by a discussion of our study methodology. We then discuss the strategies for reuse and sharing generated from our interviews and survey, along with determinants of reuse that encourage or discourage sharing and reuse among data scientists. Lastly, we discuss opportunities for future work to address unmet needs for sharing and reuse in the practice of data science. In summary, this chapter contributes:

1. Based on an interview study and survey with 149 professional data scientists, we characterize five primary strategies for sharing and reuse of analysis code.
2. We report our participants' determinants and obstacles for sharing and reuse practices and discuss implications for future tools.

3.3 Background and Related Work

3.3.1 Reusing and Sharing in Software Engineering

Reusing and sharing code benefits software developers by saving their time and resources to build and maintain applications, while maintaining code simplicity [46, 53, 87, 90]. Effective reuse relies on concise and expressive abstractions [90]. Using various form of abstractions, common strategies for software reuse include high-level languages, ad hoc code scavenging, and source code components such as libraries [90]. In particular, with the growth of open source software, library reuse has become a prevalent practice in software engineering [1, 63, 139]. Library repositories like NPM and PyPI have facilitated the sharing, discovering, and management of third-party libraries, which lead to the increasing usage of third-party libraries among software developers [1, 139]. This huge demand further incentivizes the creation and implementation of third-party libraries. However, library reuse has its limitations. Xu et al. found that developers would replace an external library with their own implementation if the library is over complicated or not flexible to satisfy their needs [172]. In data science, code is less formal compared to traditional software engineering [107]. Although data scientists generally benefit from libraries for performing common data operations and computations, these libraries tend to be low level. The practice of sharing and reusing entire analyses or workflows in data science remains unexplored and worth investigating. Thus, our work aims to reveal the reuse and sharing practice in data science programming, understand the different reuse decisions between traditional software engineering and data science, and identify design opportunities for facilitating reuse to improve work efficiency.

3.3.2 The Process of Data Science

Several researchers have investigated what steps data scientists go through in their work and how they seek to coordinate their efforts. Machine learning product development involves iterations of a process that begins with gathering model requirements and ends with model deployment and monitoring [6]. Throughout this process, data scientists, domain experts, team leaders and software engineers collaborate in unique roles as indicated by tool use (technical vs non-technical users) [175]. In this process, Jupyter notebook users tend to think less about future use of their code even though notebooks are touted as a more readable platform [175].

Several large scale reviews of computational notebooks have documented that despite the benefits of computational notebooks, they can encourage bad coding practices because of unexpected execution order and a lack of modular code [126, 140]. For example, Rule et al discovered that nearly half of the 1 million Jupyter notebooks they scraped from Github were uploaded with non-linear execution orders [140]. These notebooks serve a variety of purposes like data exploration, places to store code, or for ad hoc versioning of analyses. However, analysts must invest time and effort to clean up their notebook to make it reproducible or reusable by others. Our investigation is distinct in that we take a broader look at how data scientists specifically reuse and share their work across all platforms.

3.3.3 Tools for Data Science Workflow Management

Several platforms have been developed to help data scientists share code while programming. Git is a widely used version control system that lets users manage version of raw files for software projects [48]. However git does not work well for comparing versions of rich text files like computational notebooks. Systems like Verdant augment notebooks to better support versioning by tracking a user's analysis history [80]. Our work discusses unmet needs for sharing and reuse in data science that future tools might address.

3.4 Methods

To better understand the state of the art in sharing and reuse practices in data science today, we conduced semi-structured interviews with 17 data scientists at Microsoft. To ensure the generality of our findings, we then developed a survey that was completed by an additional 132 data scientists. Both studies were approved by an Internal Review Board, and all participants signed consent forms. Interview participants were each compensated with $25 USD gift card. Survey participants were entered into a raffle for three $100 USD gift cards.

3.4.1 Interview Study

For our interviews we recruited 17 participants from a pool of data scientists chosen at random from the employee database, based on their job titles, levels, and business units. In particular, we recruited data scientists from software product or service teams and excluded those from non-product units like Research and Legal. We conducted the interviews in one-hour sessions where we asked about the participants' sharing and reuse practices as an individual and as a member of their team. These interviews were transcribed and then analyzed for themes. Specifically, two of the authors re-read the transcripts, coded them, and card sorted the codes to come up with themes. The authors then discussed the themes until a consensus was reached. The interviews revealed both common strategies for reuse as well as factors that encourage and discourage reuse. Throughout this chapter, interview participants are called informants, and their quotes are designated with "IP".

3.4.2 Survey

To assess the generalizability of our identified themes, we ran a survey with data scientists drawn at random from the same pool as the interviews. In total, 132 participants filled out our survey, out of 563 invited (23% response rate). The survey asked for the following information:

- Background about the respondent's years of experience, tool usage, and team size;
- For each of the five strategies:
  - Frequency of reuse;
  - How the reuse happens (e.g. copy-paste, function call, etc.);
  - What functionality is reused;
  - How reused work is found;
  - Frequency of sharing;
  - Additional work required for sharing
- Influences on their willingness to do reuse and sharing
- Best practices and pain points for reuse and sharing

The survey took 10–15 minutes to complete. Throughout this chapter, survey participants are called respondents, and their quotes designated with "SP".

3.5 Participant Backgrounds

Our informants (9 male, 8 female) had experience ranging from 2–14 years of professional data analysis work. All informants were from different teams. They had a range of educational backgrounds, including fields like statistics, math, or computer science. Our respondents (85 male, 46 female, 1 did not say) reported a range of professional experience in data science from 3 months to 35 years, with a mean of 8.6 years.

3.5.1 Study Context

Our study is conducted at Microsoft, a large software corporation. At Microsoft, data scientists work both on dedicated teams and also within software engineering teams. There exists little standardization across teams mandating how data scientists must go about their work. This leads to a wide variety of tool usage and reuse strategies.

3.5.2 Team Composition

The informants are either members of dedicated data science teams, situated within larger product teams or work on product teams alongside software engineers. The survey respondents report team sizes ranging from 1–22 people, with a mean of 8 people (after removing outlier responses). However even on teams with many data scientists, most projects involve only one or two of them: 74% of respondents work either alone or in pairs on projects. Analysts communicate with team members throughout their project to communicate results, seek help, and to find code for reuse (see Section 3.8 for details), however most projects only involve one or two data scientists actually touching the code or data. This is starkly

different from large software projects within Microsoft that may have dozens of developers iterating on a single, interdependent, code base.

As noted in prior literature, much of the work done by data scientists is exploratory in nature and their coding practices reflect this. Data scientists will often begin an analysis, but if it turns into something recurring or the model they built must be scaled up for production deployment, the code will be handed over to an adjacent software engineering team. Several of our informants expressed admiration for the high quality code produced on production teams and claimed it was far superior to their own in terms of organization, commenting, and style. One of our interview participants (IP6) noted that the software engineers they work with "write pretty fabulous code in terms of readability and actual usability".

3.6 Tools shape ability to share and reuse

The interview participants use a variety of tools to do their work. The most popular languages used were Python, R, a query language for large-scale relational queries (Cosmos/SCOPE), and a query language for large-scale telemetry data (Kusto Query Language/KQL [109]). Analysis was done in a combination of computational notebooks (Jupyter, Visual Studio Code Notebooks, and Databricks) and integrated development environments. However, even data scientists on the same team rarely used the exact same tool stack. The tool chosen to write code is often shaped by how and where the data is stored, which in turn shapes the sharing practices afforded by the tool.

For example, several teams used Databricks for writing their analyses [27]. Databricks allows code to be written in a variety of languages including Python and R in a computational notebook interface. Teams using these tools developed reuse strategies around notebooks such as template notebooks and notebook libraries (Section 3.8.3). Yet other teams that do most of their analysis using KQL developed strategies to share their work through the tool where they write their queries.

Lastly, many participants noted that the final output of their work is often not the code itself. This in turn shapes both the practice of reuse, as well as the incentives to invest in creating reusable code. Results are presented as PowerPoint presentations, text documents, or dashboards. This finding is consistent with prior literature about how artifacts of data analysis are shared outside of the computing environment [175]. The consumers of these results seldom ask for the code that generated the results they are seeing. This is distinct from traditional software engineering where the outcome *is* the code. For data science, the outcome is the *result* of the analysis.

3.7 What functionality is reused?

Informants mentioned a variety of tasks for which they share and reuse code. In this section, we discuss these tasks in aggregate across all reuse strategies, from most to least common task.

Data preprocessing, transformation, cleaning. The most commonly reused data science code is for data preprocessing. This includes cleaning data, transforming data between

formats, and generally processing data into a usable state before analysis can begin. Since data analysis pipelines often start with raw, uncleaned sources, reusing code that cleans and formats data for analysis helps speed up this process and ensures that the cleaning is done consistently. Reusable cleaning code can also encapsulate idiosyncratic details of how data is represented in different data sources.

Reading and writing data. The second most common reuse task was reading and writing data from sources. Since many data sets are too large to fit locally on an analyst's machine, data scientists frequently read and write data (samples) from the same data stores or submit jobs to run on cloud servers. Working in the cloud often involves configuration details like access keys and resource IDs, which are hard to remember and therefore often copy-pasted from previous work. Informants also reuse queries that are shared within a team for commonly accessed data or to make sure different analyses are looking at the same slice of the data. For example, IP5 mentioned that engineering teams often share a query to pull anomalous data that needs further analysis: "we interact with the engineers on the team who can help us understand the telemetry better and identify what is the right query to use for certain things."

Modelling and evaluation. Code for creating and evaluating models is also frequently reused. Data scientists are often creating very similar models since their data is semantically similar over time. For example, some teams only do time series forecasting and thus use models appropriate for time series data. Other teams deal with natural language text data and so use state of the art deep neural nets for language processing tasks. This reusable modelling code can be individual snippets or entire modelling pipelines.

Data visualization and reporting. Our informants often create similar visualizations during the course of a project and reuse these across projects. They reuse visualization code when the data is similar across projects, when the task is similar (for example, showing model performance), or to reuse the work of finding the desired visualization parameters. Furthermore, several informants mentioned that they like to maintain a similar style across their charts so reuse common code to do this styling, depending on which visualization library they are using. Informants also create dashboard templates for tools like Power BI to save time and maintain consistency.

Miscellaneous tasks. Various other tasks were mentioned with less frequency, but still offer some breadth as to what data scientists view as worth sharing and reusing. For instance, some teams focusing on ML model development maintain library functions for post-processing, model selection, and model ensembling. Others reuse code that runs pipelines for analysis, composing many of the aforementioned steps together. These pipelines allow data scientists to iterate faster on models by automating tedious steps such as hyperparameter selection or data formatting.

3.8 Approaches to Reuse and Sharing

Since each informant works on a different team, each reported a slightly different approach to sharing and reuse in their work, shaped by the kinds of data they used and their team's sharing structure. We thematically grouped their sharing and reuse strategies into five distinct strategies. Some strategies are personal, namely copying previous work and keeping a

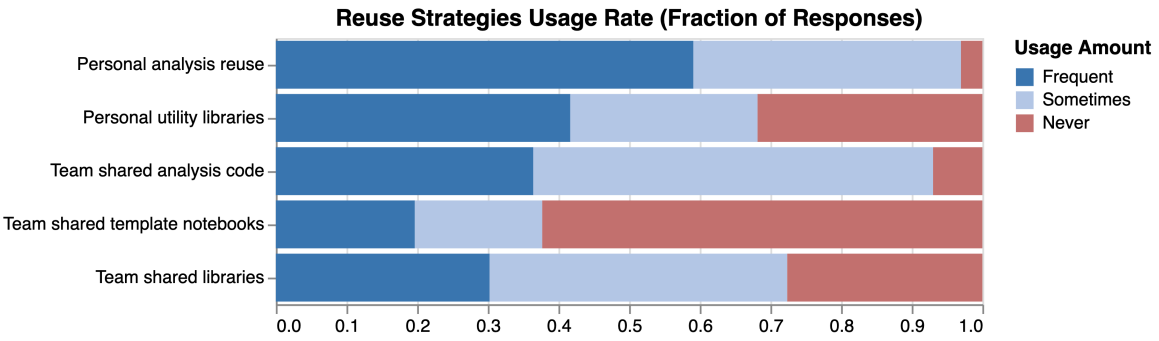

Figure 3.1: We bin usage rate into three categories: daily or weekly usage as *Frequent*, monthly, seasonal, or yearly usage as *Sometimes*, and *Never* for respondents who never use a strategy. Using personal past work is the most frequently used strategy, and notebook templates the least frequent.

personal utility library. Other strategies are team-wide efforts: sharing notebooks, creating template notebooks, and developing shared libraries. Every survey respondent used *at least one* of these strategies to share and reuse their code. We discuss three strategies in detail since they relate to broader elements of data debugging.

3.8.1 Personal analysis reuse

The most basic form of reuse is to look at one's own past analyses as a reference for current work. This code is only maintained by a single individual and is not shared. Nearly every respondent participated in this form of reuse at least sometimes (97%). Local analysis code is reused for all of the tasks mentioned in Section 3.7, from data processing to model evaluation. As one respondent (SP63) described, "Anything I know I did before, I reuse it."

Data scientists find prior work almost exclusively by memory since they are the primary author of these files. Furthermore, there is a range in which local analysis files are tracked through version control. Some informants put all of their analysis in a (personal) Git repository. Though, it is of note that this is primarily so that the code files can be accessed on another machine rather than to track versions of files. Others leave their analyses on the local file system.

The majority of analysis code stored locally is reused via copy and paste. A common case is cloning code from one computational notebook to another, then adjusting the code to suit the current context. When the reused code is a series of top-level statements rather than class or function definitions, it cannot be imported and called as an API. (Turning the reused code into a library is discussed in the next section.) In our survey, the next most common way of reusing this code was simply looking at it for reference and the least common was importing the code.

As data scientists work on a variety of projects over time, they store their analysis code. Often, this takes the form of messy analysis notebooks or scratch files to produce a certain result for an analysis. These notebooks and files are not cleaned before storage by adding extra comments or documentation as participants do not necessarily *anticipate* reusing them in the future. In this sense, reusing local analysis code is opportunistic reuse – when the code is initially written the data scientist is not sure if they will use it again in

the future and does so when the time arises. This also means reusing in this way requires little effort since minimal prior planning is required.

The benefit, and drawback, of this strategy is its ease – all it takes is putting files in some form of file structure and then finding relevant files later. Projects that begin in local file storage might later make it into a different form of reuse if they need to be shared with others. This low effort setup can lead to issues like multiple saved versions of the same file or the same analysis with incremented suffixes ("analysis-v1", "analysis-v2", "analysis-v3", ...). However, since data scientists are searching over code that they wrote they generally find it easy to find past analysis stored in this form. Several interview informants intentionally organized previous analyses by project or data store to aid in future search.

3.8.2 Personal utility libraries

Similar to local analysis code, some participants have developed a personal "utility library" of common functions or code snippets they find themselves repeating. This utility library is only used and maintained by a single individual. However, this strategy is distinct from the local folders of analysis code in that code is intentionally cleaned up before being stored in the library. The code stored in personal utility libraries is often short (no more than a dozen lines), restructured into a callable API (class and function definitions), and parameterized so it can be used in new contexts. This code might be used for all of the different tasks mentioned in Section 3.7 from data cleaning to visualization.

Since this code has been restructured into a callable API, the most common way respondents use this code is by importing or calling it directly. However, copy and pasting from the personal utility library is still very common. For such small snippets of code, it can be just as easy to copy and paste into the development environment such as the current analysis notebook. In a similar fashion to personal analysis code, most functions in a personal utility library are found by memory since the author has an intimate knowledge of the code contained therein.

Data scientists add new things to their personal utility library when they notice they repeat a task often enough it is worth cleaning up into a reusable function. Over time, this evolves into a utility library of assorted functions and templates for various tasks:

> "*These files are just calls which I have been using for three or five years now. I just constantly go to them again and again and again. So, I extracted these very, very generic common things in single repo called utils that each file just does a single thing.*" - IP12

Personal utility libraries benefit data scientists by increasing productivity for common tasks. In particular, the search cost for finding code is very low since data scientists know what is in the library by memory. The only real cost of maintaining a personal utility library is the time investment to create reusable components for common tasks. However, since these libraries are only made for personal use there is little risk that this work will go to waste.

3.8.3 Team shared template notebooks

Typically, computational notebooks have issues of non-reproducible code that limit their ability to be shared and reused [140]. However, as they have grown into the de facto tool for data science, users have developed strategies for making notebooks more reusable, namely through *template notebooks.*

Template notebooks are normal computational notebooks that have been cleaned up and generalized for a certain task. This cleaning can take many forms. Some teams put parameters at the top of the notebook and leave "TODOs" in the comments to fill in these parameters before running the notebook like a function. Over time, each of these task-specific notebooks come together as a sort of "library" for common code on the team. These notebooks are intentionally cleaned so that they will run top to bottom without issue. In this way, template notebooks are run more like a traditional python script rather than an interactive notebook. Several informants also discussed using the `%run` command in Databricks, which allows one notebook to invoke another like a function, including parameter passing.

Alternatively, a single notebook may contain numerous common functions in a *notebook as a library* (NAL). For example, IP6's team has a centrally shared Databricks notebook that has numerous commonly used functions. These functions are cleaned and abstracted before addition to the NAL. Yet when these functions are reused, they are copy and pasted out of the shared notebook rather than imported. This notebook is essentially a traditional software engineering library – similar functions grouped together under one umbrella. However it lives in *same place* analysts do their work.

Some tools have extra functionality that supports the use of template notebooks and NALs. For example, Databricks allows notebooks to be parameterized so that they can be run as a function (top to bottom) with new parameters or a default value (see [28] for more details on this syntax). Some of our informants have developed whole analysis pipelines using this functionality where one notebook does data cleaning and then calls another notebook to do modelling, and so on.

Template notebooks were used least frequently among the surveyed responses. This can be attributed to two main factors. The first is that not every notebook shared with teammates is a *template* notebook. Past analyses that are tightly coupled with a particular data source are shared on team-wide stores but unless they are cleaned to the extent they can be run independently, they are not a template. Secondly, there is limited tool support for template notebooks outside of the Databricks environment, which was used on a limited number of teams. Adding *TODO* comments to Jupyter notebooks was one alternative strategy, however executing Jupyter notebooks as a function is uncommon.

Template notebooks are most commonly accessed from Git repositories for Jupyter based templates or within an analysis ecosystem such as Databricks, where tool features best support reuse. Even for notebooks stored in a version-controlled Git repository, participants seldom use common Git operations like branching and merging. Rather, they use Git as a remote store for their work that others have easy access to, more similar to any cloud storage location. Several participants even mentioned they are not very comfortable with Git workflows and it is something they would like to work on.

Code is most commonly reused by copy and paste or (if the tool supports it) by calling

the code directly. As is the case with all forms of shared code artifacts, the most common way among survey respondents to find relevant notebook templates is by communicating with teammates. However, the next most popular response was based on memory indicating that most shared notebooks are for small, modular tasks that data scientists easily remember.

Our interview informants mentioned that most notebook template additions are done by a select few members of their team. However, everyone has the ability to share new templates. The division between reuse versus sharing frequency also holds for notebook templates: the modal response for adding new template notebooks is seasonally, whereas they are most commonly used weekly.

Why do data scientists use shared template notebooks rather than a library? One reason is that template notebooks are more compatible with where they do most of their work. If a reused notebook contains lots of visualization or table output, this most easily re-run *as a notebook* rather than a separate library. Another reason for template notebooks is that they support tweaking the code to the analysis at hand. As SP131 says, they use template notebooks "when the code requires customization when applied to different scenarios." Another motivation, from SP46, is that template notebooks are useful for "Just about any situation involving analytics. Sharing notebooks is far easier and portable than making a code library. For Data Scientists at least."

Creating template notebooks is often faster than creating a traditional library. For example, survey respondents mentioned they chose to create template notebooks because of "Extreme time constraint which limits the time I can spend on doing things the right way" (SP43) or "Most of the time because the flow is easier" (SP24).

3.9 Discussion

In preceding sections, we have presented interview and survey results on *how* data scientists go about sharing and reusing past analysis; here we delve into a discussion of *why*. We also distinguish how reuse in data science is distinct from reuse in traditional software engineering and present opportunities for future tools to improve the experience of sharing and reuse in data science.

3.9.1 Code Is Not the Deliverable in Data Science

Throughout our interviews and survey, data scientists repeatedly mentioned how the code that they write is separate from their project deliverables. In data science, the outcome of an analysis is not the analysis code itself, but whatever insights, models, or datasets are produced from that analysis. These insights are delivered through slide decks, word documents, and interactive dashboards. However, the consumers of these artifacts rarely care about the analysis code itself or might not have the technical skill set to understand all the code that went into producing the analysis output. Future tools might investigate how to tie presentation artifacts such as charts or tables back to the code and data used to create the artifact to help speed up this iteration cycle and support reuse at the presentation level as well.

Furthermore, many analyses are one-off, unrelated requests. Non-code deliverables and one-off analyses ostensibly combine to create disincentives for sharing and reuse. Despite this, reuse and sharing are still very common. Data scientists realize how much reusing past analyses improves their productivity by avoiding re-work. This different incentive structure leads data scientists to adapt common software engineering reuse strategies to their needs. The reuse of personal code and team wide libraries have strong parallels with typical software reuse whereas personal utility libraries, shared analysis stores, and notebook templates developed out of the unique needs for code reuse in data science. Future work might attempt to quantify the benefits of reusing past code, for instance in terms of the time required to complete an analysis.

3.9.2 Need for Modular and Reusable Data Science Tools

Given the need for customization, it is difficult to create modular data science analysis components. Libraries are typically used for components that change little over time, like data access APIs. However full analyses need to be customized almost every time and so are more likely to be shared using a customizable interface like a computational notebook.

There are few tools that support this kind of interaction. The best example in the tools surveyed was the `%run` syntax in Databricks for running notebooks as functions discussed in Section 3.8.3. This feature allows the parameterization of notebooks. However, if notebooks need to be customized beyond the available parameters, data scientists will often just clone and edit the notebook.

Observable offers a notebook style interface for JavaScript programming that lets users import cells from any other Observable notebook [17]. Observable is most often used for visualization creation; future work might explore how cell-level imports can aide sharing and reuse in other data science programming environments. Additionally, future tools might investigate how to combine modularity at the functional level with *modular analyses* that can be customized to the current data by allowing the addition or deletion of entire analysis steps.

3.10 Limitations

Our interviews and survey are subject to several limitations. We only interviewed data scientists from a single company with a relatively mature data science practice. However, even within one company not every data science team had uniformly mature reuse practices. We expect these results will generalize to other data scientist populations; some of the issues described may even be felt more acutely by smaller organizations with less organized data science practices. However, future work might explore this explicitly. For our survey, incorrect branching logic caused not all participants to see the Likert rating questions at the end. We report all survey results as percentages of those that responded; skipped questions or no responses are excluded from reported counts.

3.11 Conclusion

This chapter presents the results of 17 interviews and a 132 person survey and how and why data scientists reuse the code they write for analysis. Our investigation revealed data scientists reuse work through five strategies ranging from ad hoc reuse of their personal code to template notebooks for a task. Furthermore, the difficulty in creating modular, reusable code components that work within current tools underlies why code reuse is more challenging in data science than traditional software engineering. This study highlights **the need for general purpose tools that fit into data scientists' existing workflows and handle common, repetitive tasks**.

The strategy of template notebooks highlights how code reuse is most powerful when coupled with the ability to interact and customize. Participants would craft template notebooks for a task like exploring their data, but often times data exploration requires further interactions with the data that are not in the original template. We build on this insight in the following chapters to create tools that offer initial data views to speed up exploration, but also are flexible enough to interact with the data in order to ask task-specific questions.

Chapter 4

AUTOPROFILER: Continuous Data Profiling for Tabular Data

This chapter is adapted from the following published paper:

> [36] Will Epperson, Vaishnavi Gorantla, Dominik Moritz, and Adam Perer. "Dead or Alive: Continuous Data Profiling for Interactive Data Science". *IEEE VIS*. 2023.

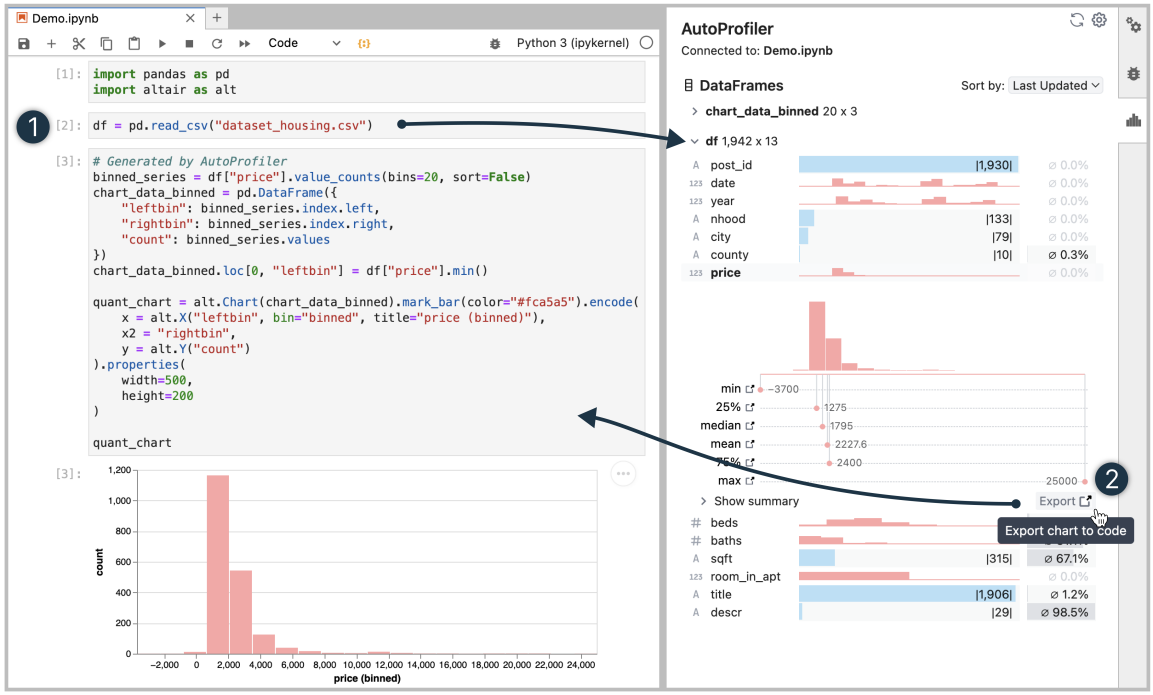

Figure 4.1: In AUTOPROFILER, data profiles update whenever the data in memory updates. This enables a fast feedback loop between (1) authoring new code to transform a dataset and (2) immediately profiling the results. AUTOPROFILER also includes interactions for writing insights from the interface back to code, such as exporting charts.

4.1 Summary

This chapter introduces two of the core technical contributions of interactive data profiling: providing automatic data profiles and facilitating fast feedback after interactions. We introduce the idea of **continuous data profiling** as a process that allows analysts to immediately

see interactive visual summaries of their data throughout their data analysis to facilitate fast feedback while programming with data. AUTOPROFILER supports continuous data profiling by: (1) automatically displaying data distributions and summary statistics to facilitate data comprehension; (2) including *live* updates where overview visualizations are always accessible and react to data updates; (3) supporting follow up analysis and documentation by authoring code for the user back to the notebook. In a user study with 16 participants, we evaluate two versions of our system that integrate different levels of automation: both automatically show data profiles and facilitate code authoring, however, one version updates reactively ("live") and the other updates only on demand ("dead"). We find that both tools, dead or alive, facilitate insight discovery with **91% of user-generated insights originating from the tools rather than manual profiling code written by users.** Participants found live updates intuitive and felt it helped them verify their transformations while those with on-demand profiles liked the ability to look at past visualizations. We also present a longitudinal case study on how AUTOPROFILER helped domain scientists find serendipitous insights about their data through automatic, live data profiles. This system lays the groundwork for how to design interactive data profiling UIs that augment programming workflows.

4.2 Introduction

In recent decades, data analysis is no longer bottlenecked by the technical feasibility of executing queries against large datasets, but by the difficulty in choosing where to look for interesting insights [11]. Interactive programming environments such as Jupyter notebooks help since they support fast, flexible, and iterative feedback when programming with data [5, 123]. However, while these coding tools were designed to track the state of program execution and variables for debugging, they were not inherently designed to track how data is manipulated and transformed. This forces users to manually make sense of and write additional code to explore their data.

Exploratory Data Analysis (EDA) is critical to understanding a dataset and its limitations and is a common task at the beginning of a data analysis [157, 165]. Yet the manual effort required to construct data profiles for EDA takes up a significant part of data analysts' time: recent surveys of data scientists show that they spend almost 50% of their time just cleaning and visualizing their data [8]. Since data profiling is so time intensive, it is easy for users to skip over important trends or errors in their data. This can lead to negative downstream consequences when this data is used for modeling and decision-making [141]. In particular, many data quality issues are potentially silent: models will still train or queries will execute, but the results will be incorrect [64]. For example, in the data profile of apartment prices in Figure 4.1 we can see that some apartment prices have negative values. If these values are not addressed, analyses or models that use this data may lead to wrong decisions.

We propose *continuous* data profiling as a process that allows analysts to immediately see interactive visual summaries of their data throughout their data analysis to facilitate fast and thorough analysis. To explore how automated tools can best support continuous data profiling, we have built a computational notebook extension AUTOPROFILER that tightly

integrates data profiling information into the analysis loop. AUTOPROFILER maintains the advantages of the interactive notebook programming paradigm, while giving users immediate feedback on how their code affects their data. This tightens the feedback loop between manipulating data and understanding it during data programming.

We explore three main features in AUTOPROFILER. First, it automatically displays profiling information about each dataframe and column to facilitate data understanding. By showing data distributions and summaries, AUTOPROFILER jump-starts a user's EDA. Second, when the data in memory updates, the profiling information updates accordingly. "Live" updates in user interfaces have been shown to reduce iteration time [104]; with AUTOPROFILER we apply this concept to data profiling to understand how it helps facilitate data understanding. Third, although AUTOPROFILER eliminates the repetitive work of authoring data profiling code, users still need to be able to conduct flexible follow-up analysis and persist interesting findings in their notebook [140]. AUTOPROFILER supports this by authoring code for the user through code exports to help users quickly select subsets, find outliers, or author charts.

We present two complimentary evaluations of AUTOPROFILER. In a user study with 16 participants, we evaluate two levels of automated assistance to see how different versions of the tool help users find errors and insights in their data. Half of the participants used AUTOPROFILER (a "live" profiler) and the other used a version that presents the same information but in a static, inline version (which we denote as "dead"). In this evaluation, we found that users experience similar benefits from both versions of the tool, "dead" or "live", and generate 91% of findings from the tools as opposed to their own code. Participants found live updates intuitive and felt it helped them verify their transformations while those with static profiles liked the ability to look at past visualizations. Furthermore, participants described how the systems sped up their analysis and exports facilitated a more fluid analysis. In our second evaluation, we conducted a long-term deployment of AUTOPROFILER with domain scientists to use the system during their analysis. These users described how the "live" system enabled them to find and follow up on interesting trends and how AUTOPROFILER facilitated serendipitous discoveries in their data by plotting things they might not have checked otherwise. We discuss how future automated assistants can build on AUTOPROFILER to augment data programming environments. AUTOPROFILER is open-sourced and available for use[1].

In summary, this chapter makes the following contributions:

1. We demonstrate the benefits of continuous data profiling with AUTOPROFILER, which supports data programming with automatic, live profiles and code exports.
2. We evaluate this tool in a controlled study and demonstrate how continuous profiling helps analysts discover insights in their data and supports their workflow.
3. We also present a longitudinal case study demonstrating how AUTOPROFILER leads to insights and discoveries during daily analysis workflows for scientists.

[1] https://github.com/cmudig/AutoProfiler

4.3 Related Work

Our work builds on prior literature on assisted data understanding, live interfaces, and linking GUI and code interfaces.

4.3.1 Data understanding is critical yet cumbersome

Understanding data and its limitations has long been an important, but often overlooked, part of analysis. Data understanding is difficult because of a variety of factors, including that data updates quickly in production environments, so automated methods and alerts have a high number of false positives [144], current popular tools require manual data exploration and become messy [123], and as datasets have grown, there are a large number of issues to check for. Prior systems in the visualization community have addressed parts of this space such as comparing data over time as models are trained on subsequent data versions [67] or methods for cleaning up notebooks during analysis [58]. However, more work is needed to understand how tools can facilitate discovering data and potential quality issues before they propagate to downstream models or analyses.

4.3.2 Prior assisted and integrated EDA tools

Prior visualization systems aim to automate the visual presentation of data to speed up data understanding. In general, this automation helps alleviate the burden of specifying charts so that users can focus more on insights rather than how to produce a specific chart [59]. Some systems automate visual presentation and then rank charts according to metrics of interest such as high correlation [32], charts that satisfy a particular pattern in the data [147], or contain attributes of interest [167]. Closely related to our work is the Profiler system, which checks data for common quality issues such as missing data or outliers, and presents potentially interesting charts to the user [76].

However, many of these systems exist in standalone tools, making them difficult to integrate into flexible data analysis workflows in programming environments like Jupyter notebooks [5]. Other systems have explored how to integrate visualization recommendations in the notebook programming context as well through visualization callbacks, libraries, embedded widgets, and similar notebook search [97, 125]. Lux [95] and other open source tools [14, 55, 120, 138] show EDA information on demand for individual Pandas dataframes. While Lux uses "always on" visualization recommendations to overwrite the default table view for pandas dataframes, users must still ask for visualizations by calling a dataframe explicitly. *Diff in the Loop* [161] presents a paradigm for automatically visualizing the differences between dataframes after each step in an analysis. Although these prior systems use automatic visualization, they still require the user to manually ask for this information after each data update and often present an abundance of information that can be difficult to compute in reactive times and for users to parse quickly. With AUTOPROFILER, we explore the benefits and design constraints around coupling automatic visualization with live updates and code authoring on the user's behalf.

4.3.3 Liveness in user interfaces

Fast iteration on data and models is a key element to effective data science [44, 144]. The fast, incremental feedback that users receive in Jupyter notebooks is part of the popularity of the platform [38, 123], yet the default presentation of data feedback in Jupyter is limited to a handful of rows. "Liveness" in user interfaces reduces iteration time through reactive updates [104], such as in spreadsheets [65]. Prior studies of liveness in data science tools have compared live interfaces to REPL (read-eval-print-loop) interfaces like Jupyter and found users like the responsiveness and clean coding that live interfaces afford [31]. Inspired by the affordances of live, reactive updates, AUTOPROFILER evaluates how automatically updating data profiles after a user changes their data can help reduce iteration time during analysis. When using AUTOPROFILER in Jupyter, users must still explicitly execute their code to manipulate the data, thus it is not a completely "live" environment. However, data profiles reactively update when data changes.

4.3.4 Linking code and GUI interactions

There is a tradeoff between tools that support using code to interact with data or direct manipulation. Programming languages are flexible and expressive, yet GUIs are responsive and easy to use [5]. Prior systems in the notebook setting have bridged this gap by writing interactions with a chart [170] or widget [82] back to the notebook automatically. This allows users to reuse analysis code and preserves the steps of their analysis. Selection exports in AUTOPROFILER serve a similar purpose of facilitating drill down into rows of interest in a dataset. Our code authoring approach differs from prior systems since we only write code to the notebook explicitly when the user asks, rather than implicitly after every interaction to avoid polluting the user's notebook.

Beyond their flexibility, programming languages remain popular for data science because they allow users to reuse old analysis code for new purposes [79], or use analysis "templates" to help users go through the same steps of analysis for similar tasks [38]. AUTOPROFILER's template exports serve a similar purpose to author code in the notebook and support follow-up analysis for tasks like customizing a plot, doing outlier analysis, or investigating duplicates.

4.4 Design Goals

We developed the following design goals to inform our system:

- **G1:** *Automatic & Predictable:* Basic data profiling information should be visualized automatically without any need for extra code in a consistent manner.
- **G2:** *Live:* When the data updates, so should all visualizations of it. This prevents "stale" data visualizations in a notebook and allows data profiles to be accessible throughout an analysis.
- **G3:** *Non-intrusive:* Since users are writing code to interact with their data, automatic visualization should not interfere with their flow.
- **G4:** *Initiate EDA:* Data profiles should present a starting point for understanding each column, which can inform follow-up analysis.

G5: *Persistence:* Tools should support writing findings to the notebook to enable reproducible and shareable analysis.

G1 and **G2** were motivated by the manual EDA which is the current status quo in notebook programming. We build on prior techniques in live interfaces [104] and automatic visualization [59, 95] to speed up the data profiling process and enable continuous data profiling. This eliminates the need to write repetitive profiling code to understand dataframes after each update. Importantly, we show the same profiling information for each type of column and visualize the data "as is" in order to facilitate finding issues (**G1**). With live updates, we situate our profiler alongside the programming environment rather than inline (**G3**) so that it does not take programmers out of their analysis *flow* [45]. This also helps declutter the programming environment since most preliminary visualization can be done in the sidebar. We make the design choice to show univariate profiling information to help users jump-start their EDA process (**G4**). Previous profiling systems often require scrolling to look through multiple pages of charts [95, 120], making it hard to find interesting problems or insights. Our goal is to facilitate rapid data understanding with data profiles, then allow users to do further custom analysis by handing off their analysis back to code through exports. Code exports also facilitate saving findings such as charts or code snippets to the notebook so that notebooks can be shared and reproduced (**G5**), a core goal in notebook data analysis [140].

4.5 Continuous Data Profiling with AUTOPROFILER

AUTOPROFILER provides data analysts rapid feedback on how their code affects their data to speed up insight generation. The system fits into a common existing workflow for analysis: using Pandas in Jupyter. Pandas is the most popular data manipulation library in Python, with millions of downloads every week [119]. Likewise, computational notebooks in Jupyter have become the tool of choice for data science in Python [123]. AUTOPROFILER focuses on Pandas users in Jupyter with the goal that features that support this workflow will generalize to other dataframe libraries such as Polars [128] or Arrow [9], as well as other notebook programming environments. The AUTOPROFILER system has three core features that enable continuous data profiling: automatic visualization (Subsection 4.5.1), live updates (Subsection 4.5.2), and code exports (Subsection 4.5.3).

4.5.1 AutoProfiler shows initial EDA automatically

AUTOPROFILER detects all Pandas dataframes in memory and presents them in the sidebar of the notebook. Each dataframe profile can be shown or hidden, along with more information about each column. This allows users to drill down into dataframes and columns of interest to see more information, providing details on demand. By situating AUTOPROFILER in the sidebar it also allows users to simultaneously look at both summary data profiles of their data in AUTOPROFILER and the default instance view inline from Jupyter.

We use the Pandas datatype of the column to show corresponding charts and summary information. We categorize the Pandas datatypes into semantic datatypes of numeric, categorical, or timestamp columns similar to previous Pandas visualization systems [39, 95].

Column profiles for each of these three data types are shown in Figure 4.2. Each column profile has three core components:

1. *Column Overview* which contains the name, data type, a small visualization, and the percentage of missing values.
2. *Column Distribution* which is shown by clicking on the overview to reveal a larger, interactive visualization of column values.
3. *Column Summary* that has extra facts about a column such as the number of outliers or duplicate values.

The overview, distribution, and summary shown depend on the data type of the column. Furthermore, the distribution and summary can be toggled on and off to show more details on demand [146]. This is important for large dataframes with many columns, or when there are many dataframes in memory to prevent unncessary scrolling. Many visual elements show hints on hover to further prevent visual clutter, providing further details on demand. Our core charting components were adapted from the open-source Rill Developer platform which shows data profiles for SQL queries [135]. We use the same visualizations in AUTOPROFILER with extra summary information and linked interactions to connect the profile to the notebook.

Quantitative Columns

For quantitative columns like integers and floats, we show a binned histogram so that users can get an overview of the distribution of the column. This histogram is shown in the column overview as a preview; a larger and interactive version is presented upon toggling the column open. On hover, users can see how many points are in each bin. We also show numerical summary information like the min, mean, median, and max of the column. This is similar to what is presented in the `describe()` function in Pandas to give a numeric summary of a column. In Figure 4.2 (left), we demonstrate this information for a price column where we can see that some of the prices in this distribution are negative, a potential error that should be inspected during analysis.

If users want to see more information, they can toggle the summary to see potential outliers, whether the column is sorted, and the number of positive, zero, and negative values. We use two common heuristics to detect outlier values. The first is if a value is greater than 3 standard deviations from the mean; the second is if a point falls outside of $1.5 * IQR$ away from the first or third quartile. Both forms of outlier detection code can be *exported to code* which allows users to investigate potential outliers more or change these thresholds for classifying the outliers with their code manually.

Categorical Columns

For categorical or boolean columns, we first show the cardinality of the column in the overview to let users understand the total number of unique values. Once toggled open, the distribution view shows the frequency of the top 10 most common values. This is similar to the commonly used `value_counts()` function in Pandas which shows the count of all unique values. In the categorical summary, we show extra information about the

character lengths of the strings in the column along with a more detailed description of the column's uniqueness. This uniqueness fact can be exported to code which lets users inspect duplicated data points. Once again, users can export a selection to code in the notebook to quickly filter their dataframe. For example, in Figure 4.2 (center) we show the information for the categorical column "county". This column has some default values of `"---"` that seem like an error, so a user can click "Export rows to code" to have the code `df[df.county == "---"]` written to their notebook and can investigate these rows further. Once this new code is written to the notebook, the user can look at this subselection in AUTOPROFILER or with their own Pandas code.

Temporal Columns

Our last semantic data type is for temporal columns, where we also show a distribution overview so users can see the count of their records over time. In the larger distribution view, users can hover over this chart to see the count of values at a particular point in time. We also show the range of the column and if the column is sorted or not. Users can drag over a selection of the column to zoom into the time range more in the visualization. We plan on adding selection exports to temporal columns in the future. In Figure 4.2 (right), we show the profiling information for a date column where a user can observe that the records in their dataset span 17 years, however are not evenly distributed with large spikes in certain years such as early 2012.

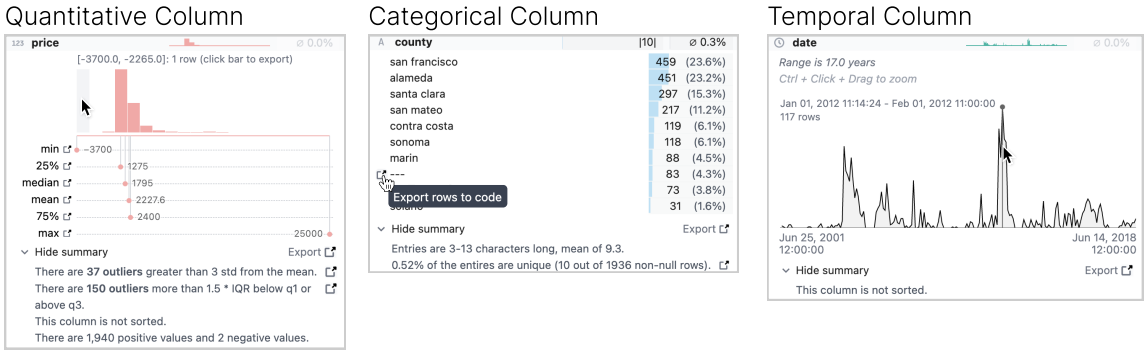

Figure 4.2: AUTOPROFILER shows distributions and summary information depending on the column type. For **quantitative** columns, we show a binned histogram along with summary statistics. On hover, the user can see the count in each bin or export the selection to code. We also show a summary with extra information like potential outliers that can be exported to code. For **categorical** columns like strings or boolean values, we show up to the top 10 most frequent values. On click, the selection can also be exported to code. For **temporal** columns, we show the count of records over time and the range of the column.

4.5.2 Live Data Profiles

Beyond showing useful data profiling information just once, AUTOPROFILER updates as the data in memory updates. Once a new cell is executed, AUTOPROFILER recomputes the data profiles for all Pandas dataframes in memory and updates the charts and statistics as

necessary in the interface. With live updates, AUTOPROFILER always shows the current state of all dataframes currently in memory in the notebook, allowing users to quickly verify if transformations have expected or unexpected effects on their data. Figure 4.3 shows this update when a string column is parsed to numeric. Here, Pandas initially parses this column as an object data type but when the user turns the column into an integer the distribution and summary information is updated. Live updates help users verify a wide range of transforms. For example, after updating the types of columns, applying filters, or dropping "bad" values.

AUTOPROFILER has several UI elements to help users track and assess changes after updates. The first is that when a user hovers over a column in any dataframe, if other dataframes have columns with the exact same name they are highlighted. For example, if a user takes the dataframe `df`, filters it to `df_filtered`, and then hovers on the Price column the linked highlights help the user make a visual connection between the two Price columns. With automatic dataframe detection and visualization, there can potentially be many dataframes in memory as users manipulate their data over an analysis. AUTOPROFILER supports sorting dataframe profiles to find those of interest. By default, the most recently updated profiles are shown at the top of the sidebar. A user can also sort alphabetically by the dataframe name. Furthermore, users can pin any profile so that it always appears at the top of the sort order.

Dataframe profiles are typically only shown for dataframes explicitly assigned to a variable with one exception: if the output from the most recently executed cell is a Pandas dataframe we will compute a profile for it with the name "Output from cell [5]". On the next cell execution, these temporary profiles are removed. This fits into a common notebook programming workflow where users display their dataframe after making a transformation to see how the data has changed.

4.5.3 Exports to code

In addition to interactive data profiles, AUTOPROFILER assists users in authoring code. AUTOPROFILER facilitates code creation in two ways: *selection* and *template* code exports. For both of these, a user clicks on a button or part of a chart and AUTOPROFILER writes code for them in the notebook below the user's currently selected cell. All code export snippets are pre-built into AUTOPROFILER and produce the same code snippet for each task with the dataframe and column names filled in so the code is ready to execute in the notebook.

Selection and template exports only differ in the kind of code they produce. *Selection exports* allow users to export selections from charts to help them filter their data, as mentioned in Subsection 4.5.1. For example, Figure 4.2 (left and center) demonstrates how a user can export selections from categorical and numeric charts to quickly filter their data. This helps users more quickly iterate on ideas during analysis to spend less time writing simple code and proved very popular in our user study.

AUTOPROFILER authors more complex code like charts or code to detect outliers with *template exports*. Code exports for these tasks are still relatively simple, only exporting up to 10 lines of code. However, this saves users from having to remember how to author a chart themselves or compute outliers. Users can then easily edit this code, for example

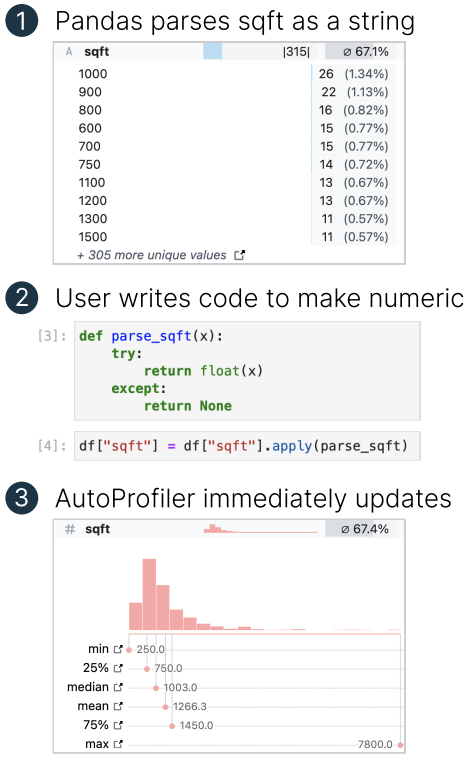

Figure 4.3: AUTOPROFILER updates the data profiles shown as soon as the data updates. In this example, Pandas parses the sqft column as a string type since some of the values initially have strings in them. Once the dataframe `df` updates in memory, AUTOPROFILER will update the profile shown. This way the user can see their transformation was successful, inspect the distribution of sqft, and even notice that the number of nulls increased by 0.3% after this parse.

to customize their visualization or change the threshold for an outlier. Prior work has discovered how data scientists often re-use snippets of code across analyses to help them speed up their workflows [38, 79]. AUTOPROFILER's exports serve as a form of these pre-baked "templates" for analysis steps. The other benefit of this type of export is that it helps preserve analysis in the notebook in the form of code, which supports more replicable analyses in notebooks, a common goal [126].

This linking between analysis in a visual analytics tool and notebook code has been introduced in previous systems such as Mage [82] and B2 [170]. Our goal here is similar: to support tight integration between GUI and code. However, our approach differs slightly in that we only write code to the notebook when the user *explicitly* clicks a button to prevent polluting the user's working environment.

4.5.4 Implementation and Architecture

AUTOPROFILER is built as a Jupyter Lab extension to augment a normal interactive programming environment with a data profiling sidebar. Figure 4.4 shows the components involved in a example live update loop. When a user executes new code, the kernel sends a signal that a cell was executed (step 1). AUTOPROFILER then interacts with the kernel to get all variables that are Pandas dataframes, and requests data profiles for each of these

variables (steps 2 - 4). When a user requests to export code, a new cell is created with the code (step 5). This is only a UI interaction, and when the user executes the generated cell, the update loop will trigger again. Whenever the kernel is restarted, the dataframes in memory are cleared so the profiles in AUTOPROFILER reset.

As a Jupyter extension, AUTOPROFILER can be easily installed as a Python package and included in a user's Jupyter Lab environment. This easy installation has proven very popular with users of our system. The frontend code for AUTOPROFILER uses Svelte [154] for all UI components.

All profiling functions are written in Python and execute code in Pandas. Pre-binning distributions in python makes serialization faster to avoid serializing entire dataframes. Since our profiling happens in Pandas, the performance of AUTOPROFILER generally scales with the capabilities of Pandas.

The scalability of our approach is primarily impacted by two main considerations: the number of columns in each dataframe and number of dataframes in memory. Pandas can still execute a single query relatively quickly for dataframes with up to millions of datapoints, and we consider a full benchmarking of pandas queries outside the scope of this work. Since requests to the Jupyter python kernel are currently executed serially, larger requests for dataframes with many columns or more dataframes in memory make updates slower. The AUTOPROFILER UI is not affected by the size of the underlying data since the queries return binned data counts or summary statistics so the UI remains responsive, it simply takes longer to fetch new data for larger or more dataframes. We have included several performance tweaks to make AUTOPROFILER usable for real workflows. For example, we do not calculate updates when the AUTOPROFILER tab is closed to avoid unnecessary computation.

The scalability of AUTOPROFILER can be improved with further engineering improvements. For example, the requests for profiling queries could be executed in parallel by augmenting the Jupyter kernel. Furthermore, faster query execution system like DuckDB [130] can speed up the response on individual queries over pandas. For particularly large datasets, the distributions and statistics could be estimated from samples.

4.6 Evaluation: User Study

We demonstrate the effectiveness of AUTOPROFILER in two ways. In this section, we discuss the results of a user study comparing two levels of automation support with AUTOPROFILER and in Section 4.7 we discuss the results of a longitudinal case study of users with AUTOPROFILER.

4.6.1 Participants

To evaluate how AUTOPROFILER helps data analysts in a sample data analysis task, we recruited Pandas and Jupyter users for a between-subjects user study. We recruited 16 participants from social media and our networks who were experienced data analysts. Our inclusion criteria required that participants be regular Pandas and Python users. Our participants had 2 to 12 years of experience doing data science (mean 4.8 years), and were all regular Python and Pandas users who frequently used Juptyer. The typical participant

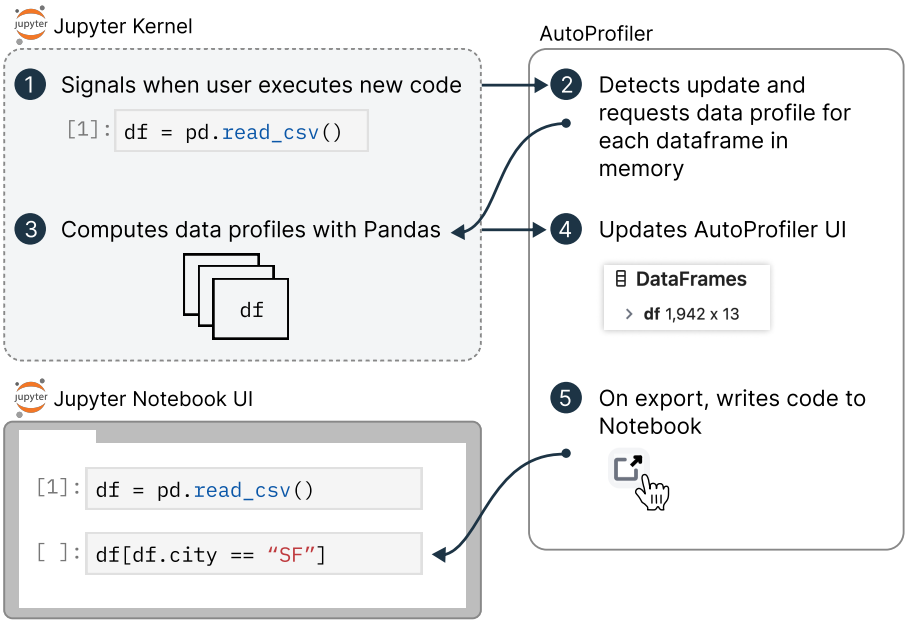

Figure 4.4: AUTOPROFILER profiling workflow. Data profiles are computed reactively when a user executes new code. Profiling is done in the kernel to speed up performance and avoid serializing the entire dataframe.

reported doing data analysis weekly and using Pandas daily, with all participants using Pandas at least monthly. Our participants worked in a variety of industries including autonomous vehicles, data journalism, and finance with job titles including data analyst, data engineer, post-doc, and researcher.

4.6.2 Research Questions

We had three primary research questions in our user study:

- **Q1.** *Live updates:* Does a profiler with live updates lead to more insights found than one with manual updates?
- **Q2.** *Starting point for EDA:* Does automatically providing visual data profiles lead users to write less code, and is this information helpful?
- **Q3.** *Linked code and GUI:* How does code exporting facilitate handoff for follow-up analysis?

These research questions correspond to the main features of our tool. We test how different levels of automation support continuous data profiling for **Q1** by comparing the number of insights found through a profiler with live updates to one that required manual invocation. With **Q2**, we explore our design choice of showing a starting point for data profiling. To answer this question we measure how many insights participants found through our tools versus their own code and their qualitative perceptions of each tool version. Finally, to answer **Q3** we measured how often exports to code are used during analysis and participants' perceptions of this feature.

In order to answer these research questions, we ran a between-subjects user study with two versions of our tool. We elected for a between-subjects design since data analysis requires time to do well and we found during pilots that having participants analyze two separate datasets was infeasible and the quality of analysis on the second task was significantly worse. We also noticed a large learning effect in pilot studies when participants analyzed two datasets back to back.

4.6.3 StaticProfiler

In our study, one condition used AutoProfiler with live profiles, automatic updates, and code exports. For our other condition, participants used a *static* version of the tool which we call StaticProfiler which requires manual invocation. StaticProfiler allows us to test how different levels of automation support continuous data profiling. The interface shows the exact same information as AutoProfiler, however, it must be called manually with `plot(df)` and does not update automatically with data updates. The same profiles for each column are presented in an inline interactive widget with the ability to hand off to code in the notebook. This sort of manual invocation is similar to other Pandas visualization tools in notebooks [95, 120]. A screenshot of the StaticProfiler tool is included in the appendix.

We compare AutoProfiler with StaticProfiler rather than other open source tools since StaticProfiler includes largely the same information as other tools but the UI design is the same as AutoProfiler. Our goal with this comparison was to isolate the effects that live updates have on continuous data profiling (**Q1**) and evaluate **Q2** and **Q3** through logs and interviews across both system versions. We compare AutoProfiler to a non-live updating tool, StaticProfiler, instead of a baseline of no tool since participants could write any extra code in the study notebook and did not have to use the tools. This allowed us to evaluate how different designs impacted tool use and how a tool augmented a typical programming workflow.

4.6.4 Procedure and Task

In both conditions, participants were first shown a demo of the tool version they would be using (AutoProfiler or StaticProfiler). Each participant then analyzed the same dataset during the task. The dataset was a sample of a larger dataset of apartment listings from craigslist [122] with extra "errors" added[2]. The task dataset had 1,942 rows and 13 columns. We sampled the dataset to a smaller size so we could be more confident that our rubric covered the majority of important insights and errors in the data.

We had 13 pre-known insights/errors that we measured to see how well participants could explore the data and find these insights as an inital "rubric" of task performance. Additionally, we included three extra insights and errors that participants found during their exploration. A detailed description of each insight/error that we measured is in Table 4.1. The categories of errors in this dataset were inspired by prior studies that group dataset errors into common types[76]. Our first 10 dataset errors are issues of missing data, inconsistent data, incorrect data, outliers, and schema violations. Inconsistent data refers to

[2]Task dataset: https://github.com/cmudig/AP-Lab-Study-Public

data with inconsistencies like variations in spelling or units; incorrect data is parsed as the wrong data type or has default values like dashes or empty strings. In addition to errors that might jeopardize an analysis if not discovered, we also measured how well participants discovered several broader insights in the dataset. Building off past definitions of dataset insights as unexpected, qualitative findings rooted in the data [116], we broadly considered insights as findings about the data that did not fit into one of the aforementioned error buckets and are important to know before the dataset is used for a downstream task. We initially included three general insights such as the scope of the dataset, realizing skewed distributions, and investigating correlations. While these errors/insights are by no means exhaustive of everything of interest in our dataset, they provide a common "rubric" that we could evaluate participants against. We consider this rubric indicative of things that *should* be found in a proper EDA of the dataset, regardless of the tool being used. With the exception of insight 13 about correlations, all of these findings can be seen in the AUTOPROFILER or STATICPROFILER interfaces.

Participants were asked to explore and clean the data under the guidance that this dataset was recently acquired by a colleague who wants to build a predictive model of apartment prices. Participants were asked to clean and produce a report about the dataset in the notebook that would be handed off to their colleague. Participants were told there were at least 10 errors in this dataset that they should try to find and fix to encourage critical engagement with the data. They were not told what kind of errors these were or what constituted an error.

Participants were given 30 minutes to explore the data with the tool and asked to think aloud about what they were investigating. Participants were asked to write down any insights and findings in their notebooks and voice them aloud. During their analysis, they were free to look up external documentation and use any other python libraries they thought might be helpful. Our research team was present if participants had questions about the task overall, however, did not answer questions about the data. We automatically logged interactions with the tools during the study. Afterward, we conducted semi-structured interviews with each participant and asked them about how they went about the task and how the tool supported their analysis. We examined the findings that participants wrote down in the notebook or voiced aloud from study recordings to quantify how many of the insights on our rubric they had found. In Subsection 4.6.5, Subsection 4.6.6, and Subsection 4.6.7 we discuss findings based on these logs and interview data.

4.6.5 Live profiles do not lead to more insights but make verification easier

In both conditions, participants found a similar number of insights: on average, 6.9 with STATICPROFILER and 7.4 with AUTOPROFILER out of the 16 we measured (P=0.71). Therefore, we did not observe more insights found with AUTOPROFILER (**Q1**). Participants heavily used both versions of the tool as demonstrated by the similar number of unique dataframes and columns explored in Figure 4.5. We suspected the live updates in AUTOPROFILER to encourage *more* tool use which would lead to more insights found but participants found both versions to be helpful during their analysis task, reinforcing the value of automatic visualization. Furthermore, live updates may not have made as much of a difference in a controlled lab setup versus a less well-defined analysis outside of the lab

No.	Type	Category	Origin	Description	Found	From tool
1	Missing	Error	Inherent	Small missingness in 3 columns	56%	100%
2	Missing	Error	Inherent	Large missingness in 3 columns	56%	100%
3	Inconsistent	Error	Added	City has lower & upper case values	69%	91%
4	Inconsistent	Error	Added	Negative prices	69%	100%
5	Incorrect	Error	Inherent	Date could be parsed to DateTime	63%	90%
6	Incorrect	Error	Added	County has default values of "---"	81%	85%
7	Incorrect	Error	Added	Sqft has strings; convert to int	69%	82%
8	Outliers	Error	Inherent	Outliers in sqft	6%	100%
9	Outliers	Error	Inherent	Outliers in price	44%	100%
10	Schema	Error	Added	Duplicate datapoints	38%	100%
11	Distribution	Insight	Inherent	Room_in_apt is almost all 0	56%	100%
12	Scope	Insight	Inherent	Dataset is only apartments in California	31%	100%
13	Correlation	Insight	Inherent	Check any correlations with price	13%	0%
14	Distribution	Insight	Inherent	Data not evenly distributed across years	38%	100%
15	Inconsistent	Error	Inherent	Check year and date cols correspond	19%	67%
16	Inconsistent	Error	Inherent	Price not properly extracted from title	6%	0%

Table 4.1: Description of each of the errors and insights on our "rubric" of participant performance. We include the percentage of participants that discovered each error/insight, noting that some discoveries were found far more often than others. As the same information was present in both AUTOPROFILER and STATICPROFILER, the discovery rate in each condition is largely comparable. The first 13 insights and errors were things we expected participants to discover ahead of time, and the last 3 were valid extra findings discovered by participants.

setting which we explore in Section 4.7.

Participants used both versions of the tools to verify that their code had the expected effect on a dataframe. For example, we observed participants finding an error through the tool, writing code to fix it, and then checking that their code had the expected effect through the tool. We particularly noticed this pattern with users of AUTOPROFILER. For example, P3 noticed error #3 that the city column contained some cities that were spelled with different casings ("Oakland" and "oakland") with the column detail view. They then fixed this error by making all the values upper case with their own Pandas code and verified that the top values were all upper case in AUTOPROFILER. As P3 described:

> *"It was nice to see when I do the upper [casing] and I can just see, oh that worked. When I do the drop duplicates, I can just look and see like, oh that worked. I like that."*

We observed this (1) find a dataset error, (2) fix, and (3) verify in the tool loop for many of our participants. Live updates help facilitate this verification since the updates happen automatically, whereas with the static version of the tool, users would often verify transformations with their own code manually. As P7 (STATICPROFILER) mentioned: "I only want [STATICPROFILER] when I'm ready for it. Because it does take up some screen space. Like I don't want it like suddenly bumping a bunch of things out of the way." Since STATICPROFILER puts visualizations inline in the notebook, multiple invocations can lead to cluttered notebooks.

Both AUTOPROFILER and STATICPROFILER also helped participants quickly discover when they had done a transformation *incorrectly*. For example, P5 used AUTOPROFILER to export the outliers for the beds column to code. However, when they re-assigned their dataframe variable, they assigned `df` to only contain outliers by accident. With AUTOPROFILER they quickly noticed that their dataframe now only contained 12 data points with extreme distributions and were able to fix their error. We observed this pattern of the tool helping find user errors during four different studies, three of which were using AUTOPROFILER.

Using static, inline data profiles is not without its advantages. For one, several users liked the ability to keep a history of past dataframes in their notebook when they called `plot()` with STATICPROFILER. Although some participants felt this led to potentially cluttered notebooks, it can be useful to scroll back to an earlier version of the data. This is not possible in AUTOPROFILER since the visualizations always show the current dataframe in memory.

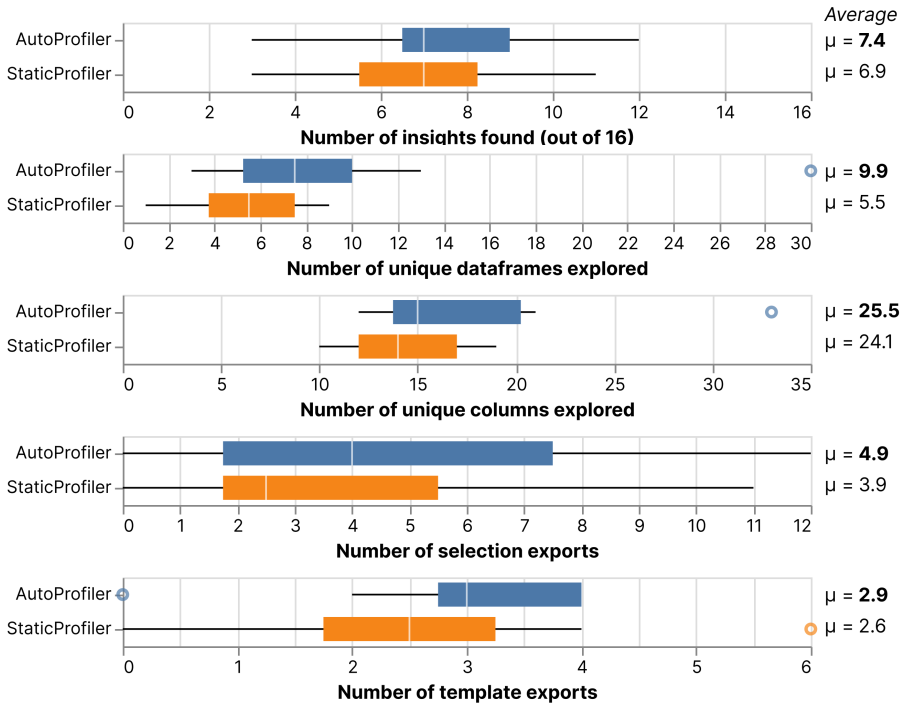

Figure 4.5: Usage and task performance metrics of AUTOPROFILER and STATICPROFILER from our user study.

4.6.6 Automatic visualizations speed up insight discovery

Participants found the tools to be useful both as a first step in analysis, but also to help them understand their data after updates and transforms. We logged interactions during the study and present metrics of interest in Figure 4.5. We measured the unique dataframes explored as the number of unique dataframes toggled open (AUTOPROFILER) or called with plot (STATICPROFILER). This metric captures how often a user returns to a dataframe after it

updates or explores a new dataframe. For example, if a user explores `df`, updates it, then explores `df` again we would count this as two unique interactions. We observed that participants with AUTOPROFILER interacted with slightly more dataframes (9.9 vs 5.5), however, this difference was not statistically significant (P=0.21). Over the course of their analysis, participants were on average inspecting data profiles in AUTOPROFILER for almost 10 different slices or updates to dataframes. One of our participants with AUTOPROFILER actually interacted with 30 unique dataframes during their analysis.

We also measured the number of unique columns (including updates) that participants interacted with and find that they explore largely the same number of columns in each condition, investigating 25.5 unique columns on average with AUTOPROFILER and 24.1 with STATICPROFILER. Since the original dataset had 13 columns, this indicates that participants were not only interacting with the original data but were returning to the profiles as they updated or filtered their data. This continuous interaction is the main goal of continuous data profiling.

Overwhelmingly, participants found their insights **with** the assistance of either tool rather than by manually writing code to get the same information. This means that when a participant said the insight aloud or wrote it down in their notebook, this information was discovered through the tool. Across both conditions, an average of 91% of insights found came from the tool, with a non-significant difference in rates between the two conditions (P=1.0). This means that on average only 9% of insights were found by users writing manual pandas code during the study. This supports that the information contained in the profiles is useful and replicates what participants would have wanted to see anyway without requiring extra code to be written (**Q2**). As P14 (STATICPROFILER) said "it does a lot of the things that I already do, but just in one succinct and easy-to-understand way". By presenting this information automatically, the tools saved participants time and prevented them from having to exit their analysis flow to look up external documentation. As P10 (AUTOPROFILER) described:

> *"I might have known to look for it, but it would have taken me a lot longer to remember how to do it in Pandas."*

When data profiling information is more easily accessible it speeds up the entire analysis loop, making it easier to discover more insights in a shorter amount of time while still being thorough. As P9 (AUTOPROFILER) described:

> "*I would probably try to do similar things that AutoProfiler suggests [on my own], but it would take a much longer time. Like the amount I did in 30 minutes, if I had to do it without AutoProfiler, would have taken hours. And then since it takes longer, my motivation would go down and my focus would go down. So I feel like I would have found far fewer errors than I could with AutoProfiler.*"

We found that not all insights were discovered with the same frequency, with discovery rates between 6% and 81%. In Table 4.1 we see that some errors like #6 were found by 81% of participants; others like #8 or #10 were found by 6% and 38%, respectively. Error

#8 was particularly difficult since the sqft column had to be parsed from a string to an integer (error #7) to get information about the outliers in the profiles. Many participants did not successfully fix this issue during the study time, explaining the low discovery rate. However, duplicate primary keys (error #10) was readily discoverable in the interface by looking at the number of unique values in the post_id column yet few participants found it. We discuss this usage trend in more depth in Section 4.8 about how tools can facilitate users finding information they would have already wanted to investigate, however if they do not know to check for an issue then this information is easily skipped over.

4.6.7 Exports facilitate follow-up analysis and learning

We also measured the number of times that participants exported to code during their analysis. Every participant used code exports at least once, with the total number of exports ranging from 1 to 16, with a mean of 7.1 exports. In Figure 4.5, we detail the average number of exports between the two tools. We see similar trends across both conditions, where participants export more selection exports than template exports. Selection exports refer to exporting a filter from a chart or summary statistic like exporting the selection for `df[df.city == "San Jose"]`. Although these exports are small, they can help make follow-up analysis easier if a user wants to filter since "that's probably the most annoying lines to constantly type is [to] just filter" (P5, using AUTOPROFILER).

Template exports refer to code for authoring a chart or getting outliers. Participants also found this helpful because it helped facilitate tweaking code for follow-up analysis. When describing their reason for using chart exports, P14 (STATICPROFILER) mentioned "It's really nice to just quickly be able to like to copy that and use it, and then I could just make some edits to it." This answers **Q3** that exports facilitate faster feedback loops.

Another unexpected benefit of code exports is the ability to actually learn Pandas better and understand what is going on under the hood of the system when it reports a statistic. As P12 (AUTOPROFILER) said succinctly: "I'm learning as I'm exploring and it's saving me time." Expanding more, P2 (STATICPROFILER) mentioned:

> *"For the educational perspective, that's something I didn't expect...specifically, I [exported] the standard deviation and I could see points inside or outside of 3 [std]. When I saw that code I learned that's the way to do that."*

The ability to teach users how to do common analysis steps is an exciting aspect of systems that support easily linking code and direct manipulation interactions.

4.6.8 Limitations

Our user study is subject to several limitations. First, subjects were explicitly told to explore and clean their dataset and were given 30 minutes to engage with a brand-new dataset. This is a relatively short time span to learn and use a new tool on new data. We also suspect that the explicit instructions to find errors and write down findings in a report might have encouraged better continuous data profiling practices than what actually happens in real-world settings. However, these explicit instructions helped us determine which features specifically aid in continuous data profiling and what kind of errors users commonly find or

miss. Another limitation is that participants analyzed a relatively small dataset. The errors and insights in our dataset were representatives of those found in larger datasets and we believe our findings translate well to other tabular dataset tasks. Finally, we compared two versions of our tool with different levels of automation to understand how they supported continuous data profiling rather than comparing to a baseline with no tool and view this as an area for future work.

4.7 Evaluation: Longitudinal Case Study

To address some of the limitations of our user study, we also evaluated how AUTOPROFILER helps data scientists in a real world environment by working with domain scientists at a US National Lab to integrate AUTOPROFILER into their workflows. These scientists work with large-scale image data collected from beamline X-ray scattering experiments to understand the properties of physical materials [83]. Two different scientists installed AUTOPROFILER into their Jupyter Lab environments and used it over a three month period during their analyses as much as they liked. We were unable to collect log data during this deployment for privacy reasons. We periodically spoke with the scientists during the deployment to make sure the tool was working. At the end of the 3-month period, we conducted in-person observations and interviews with the participants where they showed us the notebooks and datasets where they were using AUTOPROFILER and we asked about how they used the system, and which features they felt supported their workflows.

As a Jupyter Lab extension, AUTOPROFILER fits into the existing workflows of these scientists since they typically did data analysis with Python and had existing libraries for visualizing and manipulating their data. AUTOPROFILER helped improve two different workflows they have for data analysis. The first is for monitoring data outputs and quality *while* an experiment is running. Their experiments last for multiple hours or even days while they collect image readings from a sensor and then process these images into tabular datasets with Python image processing pipelines. As the scientists describe, during these experiments "real-time feedback is important as it shows us whether the experiment is working". The participants mentioned how AUTOPROFILER improved this type of monitoring since it works with any Python-based analysis and "allows [them] to easily notice any anomaly and observe a trend or correlation during experiments."

The second way the participants used AUTOPROFILER was to analyze their results after an experiment completed. In this scenario, the scientists "iteratively sub-selected a relevant set of data, using AutoProfiler as a guide, and then analyzed this subset of data using existing analysis/plotting tools. Thus, AutoProfiler has shown its value in improving data triage, data organization, and serendipitous discovery of trends in datasets". In the remainder of this section, we discuss two high-level patterns of use that emerged from interviews with the participants in our long-term deployment.

4.7.1 Finding and following up on trends

When using AUTOPROFILER to analyze their experimental results, our participants expressed how the tool facilitated finding interesting aspects in their data and then diving deeper into those subsets. In this way, AUTOPROFILER facilitated a faster find-and-verify

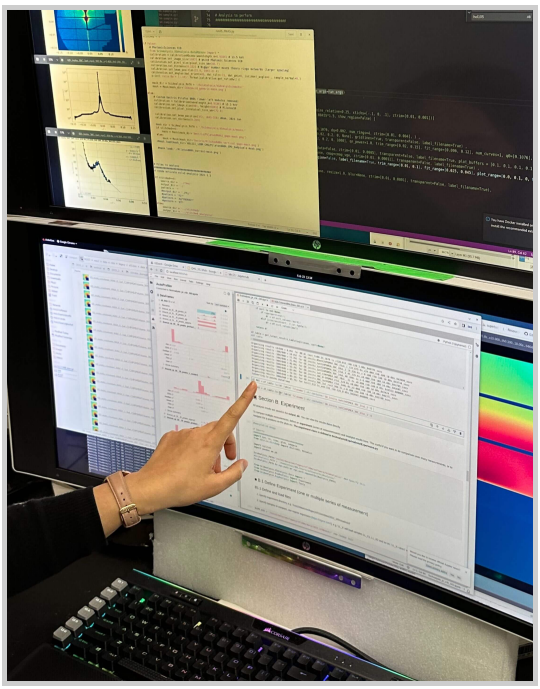

Figure 4.6: AUTOPROFILER integrated into a domain scientist's analysis workflow during our case study. AUTOPROFILER is shown on the bottom screen in the Jupyter notebook.

loop during analysis. The automatic plotting in AUTOPROFILER presented interesting plots in their dataset that helped them find subsets to export and explore further such as by running other analysis code to plot the images corresponding to each data point. They were especially excited about the possibility of incorporating bivariate charts into AUTOPROFILER so they would have to use even less of their own analysis code.

4.7.2 AUTOPROFILER facilitates serendipitous discovery

The scientists used the live version of AUTOPROFILER that updates whenever their data changes. They mentioned that the combination of all three features (automatic visualization, live updates, and code authoring) supported one another to lower the friction of their data analysis and were not enthusiastic about using versions of the tool without all of these features (such as in STATICPROFILER). Furthermore, the participants mentioned that using AUTOPROFILER helped them discover trends or errors they might not have noticed otherwise:

> *"One of the things that I very often notice is if the histogram is completely flat. That means that either all the numbers are exactly the same, or that it's some sort of sequential number. Sometimes that's what I'm expecting, so great. But sometimes, if it's not what I'm expecting, then that immediately stands out as being weird and it draws my attention to it.* ***I would never have noticed if it were not plotted; I would never have thought to plot it.****"*

Our participants described how these unexpected, serendipitous, discoveries were primarily

facilitated by the auto-updating and automatic visualizations of AUTOPROFILER and made the system a valuable part of their workflow.

4.8 Discussion

Data science is messy. There are a combinatorially large number of ways to slice a dataset, trying to find meaningful insights. The goal of continuous data profiling is to augment a human's sense-making ability by automating the analysis feedback loop to be as fast as possible. Previous work has established that automated systems can best facilitate data understanding by automating the need for manual specification [59]. We found that two different versions of automatic profiling help speed up this feedback loop in our user study. Furthermore, we found evidence that the combination of automatic visualization, live updates, and code handoff leads to a smoother, more thorough analysis loop in our long-term deployment where our participants credited AUTOPROFILER with helping them find "serendipitous discoveries" in their dataset.

In real-world tasks, encouraging critical engagement is challenging because analysts must trade off finding insights and errors quickly with a thorough and exhaustive analysis of their data. AUTOPROFILER's design removes friction by saving time and clicks to better facilitate continuous data profiling. Since AUTOPROFILER works with any pandas dataframe, users do not have to write or copy and paste profiling code that might be tightly coupled to a specific dataset. This makes notebooks cleaner and easier to maintain.

Future tools can leverage the benefits of both code and automated visualization for data analysis through linked and deeply integrated data profiles. Automatically presenting a starting set of profiling information and supporting follow-up analysis by enabling code exports helps reduce the feedback time during analysis. This approach differs from other profiling systems that aim to include as much information as possible in the interface without handing off to code [95, 120].

4.8.1 Guiding users towards unknown insights

Beyond making data analysis faster, automated systems like AUTOPROFILER can help users discover insights they might have otherwise missed. These serendipitous discoveries present an interesting opportunity for tools to help users look at their data in new ways. However, this process cannot be fully automated. Automatically presenting data profiles to users gives them the *opportunity* to find insights. Users must still take the time to look at and interpret if an insight or error is noteworthy. Automated systems can augment human expertise, but do not replace it. For example, in our user study, many participants missed important data quality issues like duplicate values, even though this information was readily available in either tool if one knew to check. The most common types of unexpected errors discovered through AUTOPROFILER were strange distributions such as a totally flat distribution or weird frequent values. The distribution information is very visually prominent in AUTOPROFILER, perhaps making it easier to discover in the interface.

Automated assistance in notebooks opens up the design space for further improvements toward guided analysis. One exciting area for future work is the potential to integrate alerts into automatic data profiles to draw user attention to important errors. For example, an alert

could be displayed if a column has a number of null values or outliers greater than some threshold. Alerts must be customizable and designed to minimize alert fatigue, or else a user may totally ignore them [144]. With existing inline, manual profilers [120] these alerts would be re-computed and displayed every time a user updates and re-profiles their data, quickly causing alert fatigue. Tools like AUTOPROFILER present an opportunity for persistent alerts between profiles that can better support continuous data science.

4.8.2 Authoring analysis code for users

Our export to code feature was very popular among participants, with many requests for even more ways to export to code. Part of the benefit of AUTOPROFILER's approach to exports is they are predictable: the system exports the same template code every time, with the dataframe and column names filled in. This is in contrast to generative approaches to code authoring such as Github Copilot [49] where a model might produce different code for the same task depending on the prompt. Users must then take time to understand this new code each time it is exported. The downside to template approaches like ours is that it is less flexible for arbitrary analysis.

In our user study, we frequently observed participants needing to look up the documentation for how to write a certain command with the Pandas library, even if they were experienced users. As tools continue to evolve to automatically write analysis code through text prompting, we think this will make data iteration even faster. The linked, interactive outputs from systems like AUTOPROFILER becomes even more valuable to help users understand their data as the time it takes to write analysis code decreases, perhaps especially when users are not manually writing all of that code and need to understand its effect on their data.

4.9 Conclusion

In conclusion, this chapter presents AUTOPROFILER, a system situated within computational notebooks that uses automatic, live, and linked data profiles to support continuous data profiling during data analysis. Tools like AUTOPROFILER demonstrate the potential for automated tools to support common data analysis tasks like dataset overview and verifying the effect of edits. This helps data practitioners more quickly get feedback on their data while programming.

With AUTOPROFILER, we develop an interactive data profiling tool that:

1. Integrates directly into the environments where users work with their data—computational notebooks
2. Facilitates fast feedback with *live* updates so that data profiles are in sync with the latest data
3. Enables handoffs between the system and code so that users can interact with their data through code and view results in the system, but also return from the system back to code for follow up analysis through code exports

Chapter 5

SOLAS: Adapting Data Profiles Based on Analysis History

This chapter is adapted from the following published paper:

> [39] Will Epperson, Doris Jung Lin Lee, Leijie Wang, Kunal Agarwal, Aditya G. Parameswaran, Dominik Moritz, and Adam Perer. "Leveraging Analysis History for Improved In Situ Visualization Recommendation". *Computer Graphics Forum EuroVIS*. 2022.

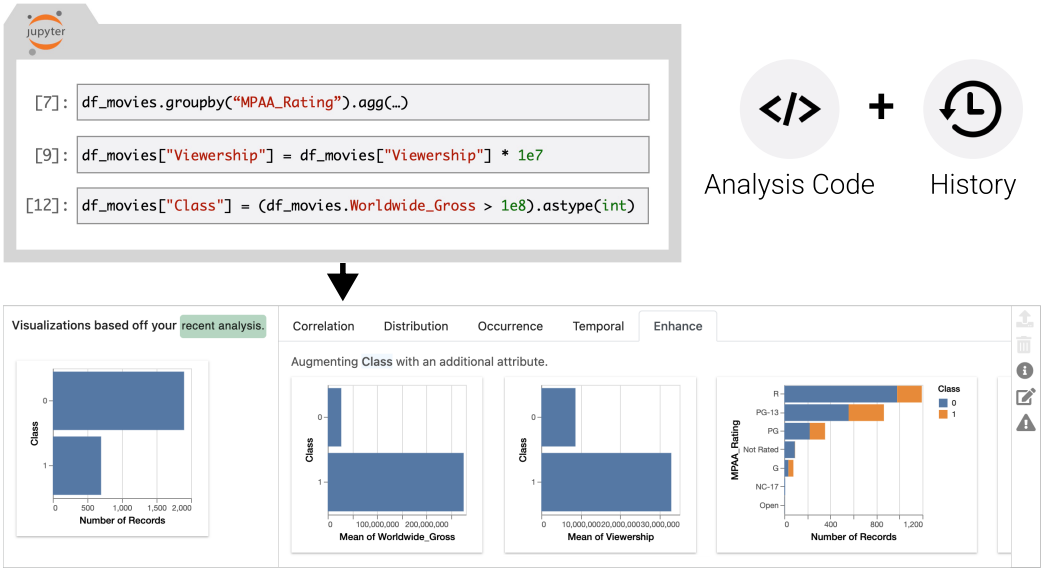

Figure 5.1: SOLAS tracks the history of a user's analysis to provide improved in situ visualization recommendations. In this example, a user has most recently created a new column called Class, so SOLAS profiles this variable in the main view of the interface. Since other recently executed Pandas commands interacted with Worldwide_Gross, Viewership, and MPAA_Rating, SOLAS ranks visualizations in this order compared relative to the Class.

5.1 Summary

In this chapter, we extend the tabular data profiling concepts introduced by AUTOPROFILER by incorporating the user's analysis history into dataset overview visualizations. Existing visualization recommendation systems commonly rely on a single snapshot of a dataset to

suggest visualizations to users. However, actual data work involves a series of related interactions with a dataset over time rather than one-off analytical steps. This chapter presents SOLAS, a tool that tracks the history of a user's data interactions through the analysis code they write, models user interest in each data column, and uses this information to provide visualization recommendations, all within the user's native analytical environment. SOLAS tracks a user's function calls to the Python Pandas library to track how they are manipulating their data and which data properties they most frequently access. The system then uses this analysis history to improve data overview visualizations in three primary ways: task-specific visualizations use the provenance of data to provide improved visual encodings for common analysis functions, aggregated history is used to rank visualizations by our model of a user's interest in each column, and column data types are inferred based on applied operations. We present a usage scenario on how SOLAS can be used for a real world analysis workflow and the results from an online user survey with 87 participants that shows that users significantly prefer the task-specific visual encodings made possible by SOLAS on $3/5$ tasks.

5.2 Introduction

During exploratory data analysis, analysts iteratively explore different methods for cleaning, aggregating, and filtering to make sense of their data [79]. Throughout this exploration, recommended data visualizations can help users understand their analysis and determine next steps by automatically visualizing interesting relationships. However, most existing visualization recommendation systems provide recommendations on a *single* data snapshot that fails to capture the dynamic nature of analysis.

Recognizing this limitation, recent visualization recommendation libraries have started providing dynamically chosen recommendations **in situ** during the iterative process of data analysis [14, 55, 96, 120, 138]. For example, the Lux library generates visualization recommendations when users display Pandas dataframes in a computational notebook. This in situ, dynamic approach to visualization recommendation has seen considerable adoption, and helps analysts identify valuable next steps in analysis [96]. However, existing libraries use a single snapshot of the dataset and/or the last data analysis or transformation step issued by the user, as opposed to the rich **history** of the user's exploration across steps. This history captures not just implicit user interest, but also provides cues into the underlying data semantics.

In this chapter, we show how analysis history can improve recommendation. With analysis history, we can better understand both the *provenance* of data as users apply iterative transformations and which parts of the data users are likely interested in visualizing. For example, if a new column is created, a natural next step may be to visualize the distribution of this column. If an analyst has also recently explored other columns, we can suggest visualizations with these sets of columns to facilitate comparison. Over time, we build a model of a user's interest in each column of their data. Each time a user interacts with their data we update this model to reflect their current interests and use this information to improve visualization recommendations.

By tracking history, we can also preserve data that is no longer in the current dataframe

for visualization. For instance, when a filter is applied to data, we plot the unfiltered distribution alongside the filtered distribution to add context. Systems without the provenance provided by analysis history have no knowledge of how to combine these dataset iterations. Additionally, the operations a user applies to each column provide a hint about the high-level measurement types of the columns. For example, when a multiplication or division operation is applied, a column can be inferred as quantitative. As we track analysis history, we use these operational signals to do better type inference and visualize columns according to this inferred type.

Using analysis history for recommendation is difficult for several reasons. First, dataframes must be instrumented so that history is logged with all relevant parameters and column interactions. This also involves logging parent-child links between dataframes when an operation returns new data and transferring history to the new dataframe. Second, each analysis operation must be interpreted to understand how the operation should be visualized and what data type information can be learned. We need to use the semantics of data returned from certain operations to offer tailored task-specific visualizations. Lastly, combining analysis history into a model of a user's interest in each column is non-trivial. Analysts shift their focus as they learn more about which parts of the data they find interesting. More recent data interactions yield a stronger signal about which parts of the data should be visualized and older interactions become less relevant over time.

We address the aforementioned challenges by integrating history tracking into a visualization recommendation tool, SOLAS. Our tool demonstrates how the extra information from analysis history leads to more insightful and better visualization recommendations. We provide a user evaluation of our task-specific recommendations that demonstrates that Python users find our provided visualizations useful for understanding the returned data from analysis functions. SOLAS is open-sourced and available for use[1].

In summary, our contributions are as follows:

1. We provide an extensible approach for logging, weighing, and combining data interactions during analysis.
2. We demonstrate how to use the semantics of the data returned from specific analytical function calls to visualize them with appropriate encodings. These task-specific visualizations often include data from previous analysis steps.
3. We introduce a method for aggregating over history to model user interest in columns and to update inferred data types based on data transformations.

5.3 Background and Related Work

5.3.1 Interacting with Analysis History

Provenance is often used to describe the history of how data and analysis evolve over time. Data provenance is used to understand analysis history, adapt to user preferences, and suggest next steps in a variety of analysis settings [173]. Graphical histories offer an approach for exploring the analysis history logged from a user's UI interactions with the visualization tool Tableau [60]. B2 logs an analyst's interactions with data as code snippets in a notebook

[1] https://github.com/cmudig/solas

so they can more easily recreate an analysis [171]. Analysis history is also useful in experimentation and versioning so that analysts can track multiple versions of an analysis, switch between versions, and interact with these histories during exploratory programming [79, 80, 81, 164]. SOLAS offers a unique method for tracking analysis history through the code executed and suggesting visualizations from this history.

Aggregated past analyses can also be used to suggest potential next steps. Data scraped from Jupyter notebooks on GitHub has been used to suggest possible function parameters during analysis or to suggest next steps based on a user's current exploration path [131, 174]; similar approaches have been used to recommend related SQL queries during analysis [3]. Data scraped from public repositories only represents a single snapshot of analysis and thus does not contain the full history of the user's exploration; SOLAS captures more detailed data interactions.

5.3.2 Visualization Recommendation

Visualization recommendation typically has two goals: (1) helping analysts follow best practices by creating visualizations that are both expressive and effective, and (2) removing the tedium of crafting visualizations to make the exploration process faster and more robust [59]. These goals manifest in systems that recommend a combination of *design* and *data* variations [166]. Design variation shows data in a variety of visual encoding to a user to allow them to select the best way to visualize their data; data variation shows different combinations and subsets of the data to help users find interesting trends or patterns.

One of the early systems to focus on visualizing *design* variation was Mackinlay's APT system [102]. APT recommended visualizations that satisfied the competing criterion of *expressiveness* (conveying the truth) and *effectiveness* (is the truth readily perceived). Later work also focused on presenting a variety of design encodings to an analyst that satisfy constraints on design best practices [111], or based on user-specified interest [103]. Most similar to our approach is Behavior-Driven Visualization Recommendation (BDVR) which matches a user's patterns of analysis to suggest alternative visual encodings [50]. Our approach is distinct in several ways, namely that we track a user's analysis history through their code rather than direct manipulation and thus impose fewer restrictions on user inputs. By situating SOLAS in the Jupyter ecosystem, we provide visualizations when the alternative is no visualization at all, whereas all exploration in BDVR is visual.

Another complementary approach to visualization recommendation focuses on recommending data variation to the user. Foresight allows users to explore by selecting a guidepost metric of particular interest (such as high correlation) and then view charts with similar statistics [32]. In Zenvisage, users specify a query by sketching the general chart pattern they are looking for, such as a sharply increasing linear curve, and are presented with charts that loosely match this pattern [147].

The Voyager and Voyager 2 recommendation systems are driven by the maxim to "show data variation not design variation" [167, 168]. In these systems, a user specifies an attribute of interest, and the system shows visualizations of this attribute with one other attribute (possibly a wildcard) ranked by perceptual effectiveness scores. Furthermore, the CompassQL recommendation engine underlying these systems supports the partial specifications of visualizations that can fill in reasonable defaults according to best practices [166].

Similarly, SeeDB allows a user to specify a base data query and the system finds interesting visualizations by comparing statistics between charts such as the skew or correlation of the data [158]. Despite their focus on data exploration, data variation systems notably all focus on a single iteration of a dataset as input. However, during their exploration, analysts are transforming, adding, and deleting data. By taking into account the provenance of data, SOLAS uses the history of analysis to provide improved recommendations.

SOLAS is an extension of the popular Lux library that recommends visualizations for Python Pandas dataframes [96]. Lux allows users to center recommendations around a particular attribute or subset of the data through a manually specified *intent*. However, in initial studies of the *Lux* system, users seldom used the intent specifications and found the in-place, immediate recommendations that *Lux* provides to be most helpful [96]. By tracking analysis history, SOLAS is able to automatically infer user intent and recommend appropriate visualizations. History tracking, task-specific visualizations afforded by history, and operational type inference are all unique to SOLAS. SOLAS groups recommended visualizations into the same semantic tabs as Lux such as Correlation, Distribution or Occurrence but sorts the charts in each of these tabs by the model of user column interest. The history tracking capabilities of SOLAS are not tied to Lux and can also be applied to other systems.

5.4 Tracking Analysis History

Our system design brings analysis tracking to users' native data analysis environments so they can use their normal data exploration tool stack. SOLAS tracks history for the popular Python data manipulation library Pandas and presents visualizations directly within Jupyter notebooks. Pandas is the most popular data manipulation library in Python, with hundreds of millions of downloads [119]. Likewise, computational notebooks in Jupyter have become the tool of choice for data science in Python [123]. Due to their widespread adoption, SOLAS focuses on analysis history tracking and visualization in this ecosystem. Users can explore their data using Pandas and SOLAS automatically creates visualizations based on their analysis history.

5.4.1 Logging Python Pandas Function Calls

Most analytic actions in Pandas occur through the DataFrame and Series APIs which are abstractions over data tables and arrays, respectively. To collect analysis history, we override the Pandas API at runtime so that operations applied to dataframes or series are captured. For the user, the API does not change and they can use Pandas functions like normal; behind the scenes, whenever one of the overridden functions is called, we log the interaction to that dataframe's history. Although Pandas supports some unique analytic functions for series or dataframes, the SOLAS user experience is not substantially different depending on the underlying data object so we focus the majority of our examples on dataframes.

For each operation, we collect four pieces of information: the dataframe this operation occurred on, the data columns in the operation, the type of operation, and the time (in terms of execution count) when this operation occurred. In Jupyter, code is organized into cells that can be executed in arbitrary order. The output of the chunk of code in a cell is shown

Task	Code Example	Information Learned
Column Reference	`df["A"]`	Increased interest in *A* column.
Column Assignment	`df["B"] = df["A"]10`	Interest in *A* and *B*. Both are quantitative.
Value Counts	`df.A.value_counts()`	Increased interest in *A*.
Describe	`df.describe()`	Interest in quantitative columns of df.
Groupby, Aggregate	`df.groupby("C")["B"].mean()`	Interest in cols; *C* is nominal and *B* quant.
Aggregate	`df.mean()`	When plotting show error bars.
Filters	`df[df.A > 30]`	Interest in *A*. *A* is at least ordinal.
Null Checks	`df.isna()`	Plot as stacked bar.
Correlation	`df.corr()`	Plot as correlation matrix.

Table 5.1: Common analysis tasks that have accompanying task-specific visualizations in SOLAS. When an operation is performed, it is added to the history of that dataframe.

immediately below the cell. Whenever a cell is executed, the execution count increases by one and thus we use execution count as a time ordering of analysis commands. When `df["Medal"].value_counts()` is run, SOLAS logs that this operation occurred on the dataframe `df`, referenced the `Medal` column, was a value counts operation, and occurred at a certain execution count during the analysis. We discuss how we use this information for improved visualizations in Section 5.5.

SOLAS maintains its own history of operations for each dataframe or series object. This was an intentional design decision since users may have dozens of dataframes in memory so we want to be sure to show relevant visualizations for each dataframe. This also resolves ambiguities when two dataframes have the exact same column names but different data so interest in the `Age` column of one dataframe does not influence interest in the `Age` column of another.

Beyond a single operation, analysis history represents a *sequence* of operations over time. Many analysis steps return new dataframes or change an existing dataframe. Figure 5.2 demonstrates how a filtering operation on `df_movies` at time step 11 returns a new dataframe that is assigned to the variable `filt_df`. By tracking history, we know that `df_movies` is the parent and `filt_df` the child. When an operation returns a new dataframe, this new object inherits the history from its parent. However, subsequent operations only affect either the parent or child, but not both. For example, the column assignment at execution count 12 in Figure 5.2 only affects the interest model of `df_movies` and the mean calculation in time step 14 only affects `filt_df`. By tracking this data provenance, we maintain references to data that would have been lost otherwise and can create unique visualizations that use the data before and after an operation is applied such as showing the background distribution for filtered data.

SOLAS *Tracks Common Pandas Analysis API Calls*

To ensure coverage of commonly used Pandas functions, we scanned the API documentation of Pandas and identified common analysis functions that might be applied to a dataframe or series. We additionally observed over 10 hours of online Pandas tutorials and analysis demonstration videos that showed how people use the API for real-world analysis tasks. Overall, our tracked analysis functions cover the most common *analysis*

functions from previous investigations of Pandas API usage [124]. These operations range from simple variable selections to complex filters, aggregations, and statistical functions. Operations for which we provide a task-specific visualization are presented in Table 5.1. We also track history for additional functions like `df.head()` or `df.tail()`. However, since these functions interact with all columns in a dataframe, they do not provide us with additional information to model user interest in specific columns. We do not track history for table joins or operations that span multiple data tables since we focus our recommendations on visualizing one dataframe (and its history) at a time. Furthermore, joins result in a single dataframe object that can be visualized.

5.4.2 Modeling Column Interest

Analysts' interests shift over time as they explore their dataset. In order to reflect this in our recommendations, we consider more recent data interactions to be more important than older interactions. At the start of an analysis, no history exists and thus all columns are equally interesting. Over time we update our model of an analyst's interest and provide recommendations tailored to their recent analysis.

To accomplish this time-weighting in SOLAS, we use Jupyter's execution count as a time index for each history item. Every time a user executes a code cell, this execution count increases by one. Each operation begins with an initial weight w_0, that corresponds to how much we value this operation in our history. Most operations begin with $w_0 = 1$, with the exception of two operations. Column references begin with $w_0 = 0.5$ and column assignments with $w_0 = 2$. We found that column references are extremely common and happen in almost every single piece of analysis code. Therefore, we begin column references with a weight of $w_0 = 0.5$ to reflect this weaker signal. Likewise, column assignments are rarer and thus should be strongly valued. This is similar to the logic of TF-IDF from natural language processing, where more common words across documents are less interesting [73]. Since column references happen more frequently, they provide us with less signal about a user's interest.

To calculate our model of user interest in each column at a time-step t, we begin by iterating through the items in the dataframe's history in reverse order and decay the weights according to an exponential decay function. This decay function allows us to prioritize data interactions that occurred most recently in the overall execution count as well as within a single cell. The weight of a history item that occurred at time t is $w_t = w_0 \times 0.85^{n-t} \times 0.95^{line\_num}$, where w_0 is the initial weight of the item, $line\_num$ is the *within* cell index (starting at 0), and n is the total number of history items. This involves two hyperparameters: decay *between* execution counts (i.e. in different cells), and decay between operations that occur *within* the same cell. We use a value of 0.85 to decay history between execution counts and 0.95 to decay history items that occurred during the same execution count. After we decay the history, we exclude operations with a decayed weight less than a threshold of 0.25. This allows us to exclude older history items that are likely no longer relevant. Users can customize all three of these hyper-parameters (decay rates and exclusion threshold) through the SOLAS API. These parameters primarily affect how long interactions are considered for recommendation, and we found in practice that the model's column ranking is relatively stable across parameter values. To produce a ranking

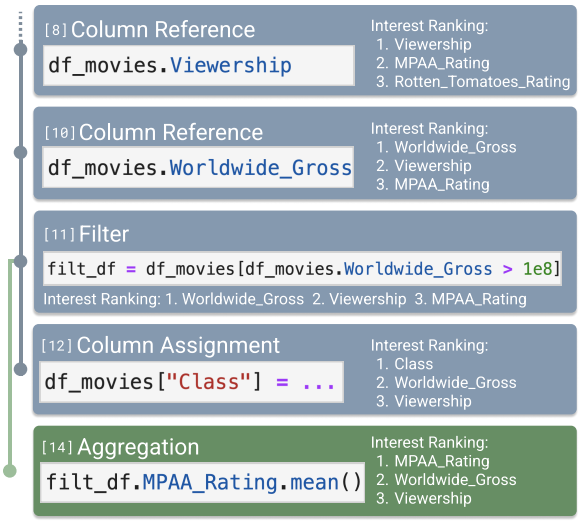

Figure 5.2: The interest rankings demonstrate the model of column interest at different steps during the analysis in Section 5.6. When `Viewership` is referenced in time step 8, it has more interest than other columns from earlier in the analysis. After the filter in `[11]`, `filt_df` inherits the history from `df_movies` and further commands affect the models of each dataframe independently.

of column interest, we sum across the weighed history and sort so that columns with the most cumulative weight are given the highest ranking. This prioritizes columns that are referenced frequently and more recently.

SOLAS uses the model of column interest to visualize columns in the most recent operation relative to columns of interest in the enhance tab and to sort other recommendation tabs. The column interest ranking shown at each time step in Figure 5.2 demonstrates how this history aggregation works in practice. In time step 14 (the green box), the column interest model for `filt_df` ranks `MPAA_Rating` most highly since it was referenced most recently, and the interest in `Worldwide_Gross` has been decayed.

5.5 Visualization Recommendations from Analysis History

Once we have modeled column interest from analysis history, we can use it to improve visualization recommendations in three ways. First, we use the provenance afforded by history to provide task-specific visualizations to visualize data from specific function calls with appropriate encodings. Next, we use the model of column interest to enhance the most recent operation's visualization and sort other recommendation tabs. Lastly, we use the operations that an analyst applies to each column to improve type inference and provide better type-appropriate visualizations.

5.5.1 Improved Task Visualization

The most recent operation a user has applied to their data gives us the strongest signal about their current interest. As described in Subsection 5.4.2, the last operation gets the most weight in our model of a user's column interest. We also provide task-specific visual-

izations catered to the most recent operation in the SOLAS UI. To encode a single operation, we provide visualizations that, in our opinion, best communicate the task that the operation performs. Each of the functions in Table 5.1 is encoded in a specific way that best presents the task this function aims to accomplish. We chose the task-specific encodings to reflect both common practice (e.g. heatmaps for correlation matrices) as well as encodings that follow best practices such as those synthesized in prior work [111].

Our task-specific visualizations fall into two broad categories: those that detect pre-aggregated data and those that use historical data from earlier in the analysis history. Many analysis functions such as value counts return pre-aggregated data. Typical recommendation systems are unaware of this provenance and treat these aggregates as raw values, producing nonsensical visualizations. SOLAS is aware of the function call that produced data to visualize to avoid this pitfall. Other tasks, like filters, benefit from data that is *no longer in the current dataframe* to give additional context. Some of our encodings, such as for `describe` handle pre-aggregated data and use historical data for outliers.

Detecting Pre-aggregated Data

Several analytic functions aggregate the raw data in various ways and return the results. We use the semantics of the returned data to visualize each function in a task-appropriate way.

Value Counts: The value counts function returns the count of each unique value in a column of a dataframe. Existing visualization recommendation systems will encode these counts as raw values; SOLAS knows that they represent category counts and encodes the data as a bar chart.

Correlation: Calls to `df.corr()` return a correlation matrix. SOLAS plots this data as a heatmap over correlations to make it easier to spot columns with low or high correlation. Figure 5.3 shows a Correlation matrix visualization for the Movies dataset discussed in Section 5.6.

Null counts: There are several ways to check how many nulls are in a column in Pandas including `isna`, `isnull`, and `notnull`. Each of these functions returns a Boolean dataframe representing if a value is null. We visualize this data as stacked bar charts showing how many nulls are in each column to help analysts identify columns with many (or few) null values.

Encoding Historical Data

In addition to understanding the semantics of data returned from analysis functions, analysis history allows us to visualize historical data that is no longer in the dataframe. Data from earlier in the analysis lineage proves useful in a variety of analysis tasks from providing overviews with outliers to adding context to filters.

Describe: The describe function returns statistical summaries of each quantitative column in a dataframe such as the count, mean, std, and quartiles. Since the goal of this function is to get an overview of the column, we visualize the data in a boxplot to communicate the distribution of the columns. The power of SOLAS becomes evident here as the data returned by `df.describe()` does not contain enough information to visualize a boxplot. Instead, SOLAS retrieves the parent dataframe (`df`) in order to plot outliers in the

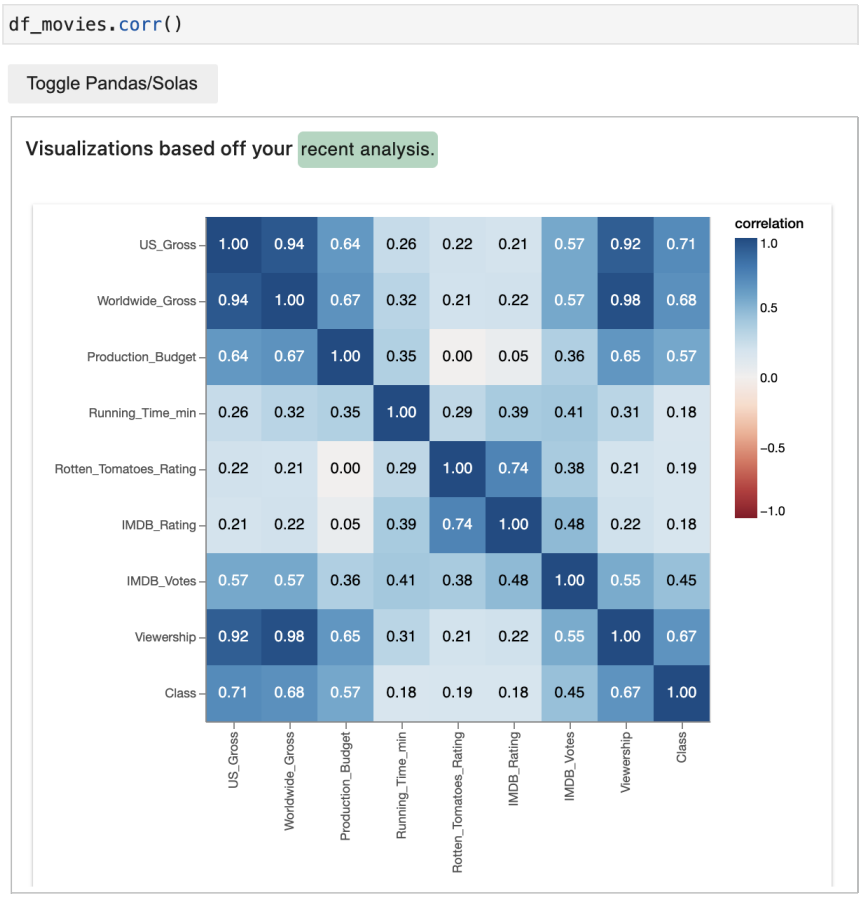

Figure 5.3: By using analysis history, SOLAS better understands the semantics of data. It knows the values returned from `df.corr()` represent a correlation matrix and visualizes this data as a heatmap to highlight columns with high or low correlation.

data needed for the boxplot. Furthermore, the data returned from `df.describe()`, like many analysis operations, is pre-aggregated. Visualizing these aggregates as raw values by treating them as quantitative values results in nonsensical visualizations. Yet without the history captured by SOLAS, a recommendation system would be unaware this data is pre-aggregated. Figure 5.5A shows our boxplot visualization of this function. Without analysis history, visualizing outliers is not possible.

Filters: During data analysis, filtering is extremely common. In Pandas, users can accomplish the same filtering task with the following three commands: `df[df.Age > 30]`, `df.loc[df.Age > 30]`, `df.query("Age > 30")`. To better understand a filter, it can be useful to plot how the returned data compares to the original distributions. Filtering on one column can sometimes have unexpected effects on the distribution of other columns. In SOLAS, we support this comparison between the filtered and original distribution by visualizing these two distributions as overlapping bars or histograms for each column. Users can toggle the background distribution on or off to support focus on only the filtered data or to compare to the background. Additionally, we sort the returned charts to prioritize distributions that shift the most after the filter is applied by calculating the earth mover's distance between the two distributions in the same approach as SeeDB [158]. Sorting in this way

allows users to compare which distributions shift the most because of the filter. Figure 5.5D shows an example filter visualization for filtering the worldwide gross column. The first visualization shows how many points remain in the data after the filter is performed. The following visualizations show that the distribution of `Worldwide_Gross` shifted the most after filtering, followed by `US_Gross`, `Production_Budget`, and so on.

Groupbys and aggregations with mean: When users perform any type of aggregation on the mean, we augment the visualization by plotting error bars with the standard deviation to provide additional context to the mean values. This reflects statistical best practices around plotting mean values. To compute the error bars, SOLAS once again references the parent of the aggregated data to calculate the standard deviation. Example function calls that elicit this visualization include both data aggregations that calculate the mean for any quantitative column in the data (e.g. `df.mean()`) as well as groupbys with mean (e.g. `df.groupby("A").agg({"B":"mean"})`).

For any groupby aggregation, we also update the x-axis name to include the aggregation so that users know how their data was aggregated in the plot. Figure 5.5C shows an example groupby where `Rotten_Tomatoes_Rating` is aggregated by its mean. SOLAS shows error bars for these groups, allowing users to understand the standard deviation in addition to the mean, without having to write any extra code.

5.5.2 Using Column Interest Model

SOLAS uses the model of column interest described in Subsection 5.4.2 in two ways. First, high interest columns are compared in the enhance tab. As the most recent operation has the highest interest, this tab shows visualizations comparing columns in the most recent operation to other recently interacted columns. Second, the recommendations in the other tabs are sorted according to this interest model. Particularly as datasets grow wide, there are many possible visualizations that can be shown to visualize univariate and bivariate distributions. Therefore ranking visualizations becomes increasingly important to show users visualizations that correspond to columns they care about. We group our recommended visualizations into the same task groups as Lux, including tabs for Correlation showing scatterplots and Distribution showing histograms. However, we sort the visualizations in each of these tabs by the column ordering provided by our model. SOLAS thereby shows visualizations most relevant to a user's recent interactions at the front, reducing the time needed to scroll to find relevant charts.

Figure 5.1 demonstrates this sorting after several analysis steps. Since our analyst has most recently interacted with the Class variable it is shown on the left-hand side of SOLAS's UI. Next, SOLAS shows `Class` relative to `Worldwide_Gross`, `MPAA_Rating`, and `Running_Time_min` since these columns were also interacted with during the analysis in decreasing order of inferred interest. Other tabs such as Correlation, Distribution, Occurrence, and Temporal use this same ordering to present the most relevant visualizations to the user first.

Operation	Aggregation	Inferred Type
$=, \neq$	...	Nominal
$<, \leq, >, \geq$	min, max, median	Ordinal
$+, -$	mean, sum	Interval
$*, /, //, \%, **$	prod, std, var, sem, skew	Ratio

Table 5.2: When column operations or aggregations are applied to the data, SOLAS updates the data type if it learns new information from the interaction.

5.5.3 Inferring Data Types from Interactions Through Analysis Commands

The last way that we use history to improve recommendation is by using the operations that an analyst applies to each column to do better measurement type inference. Measurement types refer to the meaning of a column such as nominal, ordinal, interval, and ratio variables as opposed to data types such as int or float.

Type inference in SOLAS happens in two stages. First, we infer types with traditional methods based on dataset statistics and data types. Next, we update these default measurement type inferences based on the operations a user applies to each column. With better types for each column, we are able to visualize data with more appropriate encodings. We infer types for levels of measurement based on the operations supported by each level. Nominal variables only support equality, ordinal variables also support comparison, interval variables support addition and subtraction, and ratio variables also support multiplication and division [152]. Each "higher" level supports all operations below. When we see an operation applied to a column, we know that the column must be *at least* of that level. Table 5.2 shows Python operators and their corresponding level of measurement that we use for type inference. For simplicity in SOLAS, we visualize variables as either nominal or quantitative and therefore group nominal and ordinal inferences into nominal and interval and ratio into quantitative.

Figure 5.4 demonstrates how we can update the type of a variable from interactions and how this affects recommended visualizations. In this example, the `Viewership` column is inferred as nominal by default since it has a low cardinality. However, once a multiplication operation is applied to the column, we learn this column must be quantitative in order to support multiplication. This type update changes the univariate visualization of `Viewership` from a bar chart to a binned histogram, and changes how `Viewership` is visualized in bivariate distributions as well. These type updates only go up the levels. Therefore if a column is inferred to be quantitative by default and we execute `df.col == 45`, we will not change the type to nominal since quantitative columns also support equality.

In addition to mathematical operators, we also learn type information from the aggregation functions presented in Table 5.2. The levels in each of these aggregation functions correspond to the operations required to carry out that functionality. For instance, `median` only requires greater than or less than comparisons so lets us learn ordinal information, whereas calculating the product (with `prod`) of a column requires multiplication and thus tells us this column should be Ratio typed.

There is one exception to the rule of only going up the levels of measurement. When

a user groups by a column, we infer this column to be nominal. We use this as a heuristic since it does not make sense to group by a quantitative column (without binning) and so any column that is used to group should be nominal. With any of these type inferences, there is the possibility that we will update a column's type erroneously. Users are able to override inferred types manually though the SOLAS API by using `df.set_data_type()`.

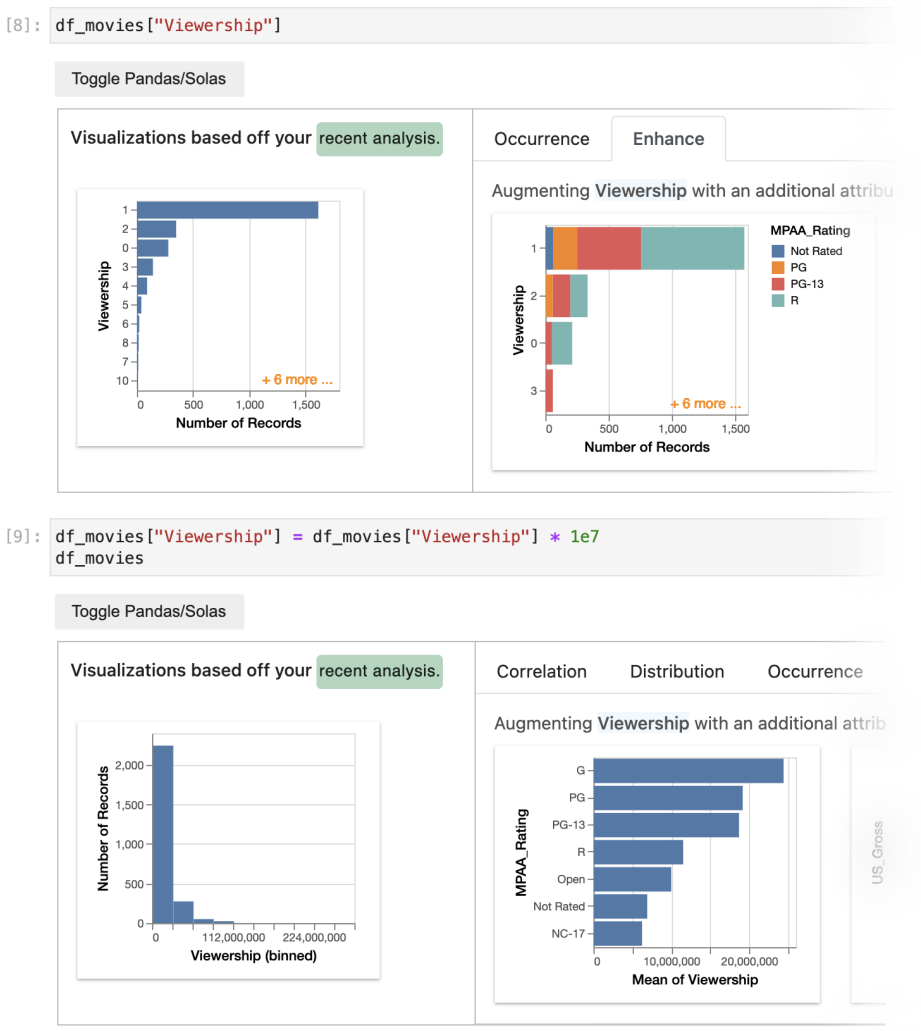

Figure 5.4: Viewership initially represents the count of viewers in 10 millions. Since it has low cardinality, it is visualized as a nominal variable. However, when we re-scale the column by multiplying by 10 million, SOLAS infers that Viewership must be a quantitative column that supports multiplication and visualizes accordingly.

5.5.4 Interacting with History

In SOLAS, all of the history tracking and recommendation happens under the hood. However, we support interactions for users to browse the history of operations that occurred on a dataframe (or its ancestors) to better understand the past operations and visualize them.

When a user clicks on a previous step in the analysis, we show them the visualizations for this specific task. Additionally, users can delete history items if they do not want them to influence their recommendations.

5.6 Usage Scenario

To demonstrate the use of SOLAS, we describe an example analysis scenario where an analyst uses the system to explore a movies dataset to create a model for predicting movie revenue. The dataset contains columns such as the movie title, revenue, rating, viewership, etc. This example demonstrates many features enabled by collecting and reasoning about analysis history.

To begin her analysis, our analyst loads the CSV file with Pandas into her Jupyter notebook and calls `df`. Her dataset contains 3,201 rows and 17 columns. By default, SOLAS shows four different groups of visualizations: correlation, distribution, occurrence, and temporal. Once she begins exploring, SOLAS will be able to use her history to suggest even more visualizations.

5.6.1 Supporting Analysis with Task-Specific Visualizations

To get an overview of her data, she calls `df_movies.describe()`. There are initially eight quantitative columns in the dataset, and the `describe` function returns summary statistics for each column. To visualize this data, SOLAS plots each of these eight columns as a boxplot (Figure 5.5A). Looking at the plots, she notices many high-range outliers on the `US_DVD_Sales`, `Worldwide_Gross`, and `US_Gross` columns. By using data from the parent dataframe, `df`, to plot these outliers, SOLAS is able to provide a visualization that best caters to the overview task of `describe`.

Next, our analyst checks if she needs to clean any columns. She calls checks the nulls and looks at the bar charts to see the results. Most columns have no or very few nulls; however, the `US_DVD_Sales` column is almost all null. She decides to drop this column. Additionally, she filters to only keep non-null rows for `MPAA_Rating`, as she is potentially interested in including this column in her model. `MPAA_Rating` corresponds to movie ratings like PG-13 or R. SOLAS visualizes the returned data as a filter so our analyst can inspect how the `dropna` operation affects the distribution of other columns (Figure 5.5B).

Next, our analyst looks to explore how metrics in the dataset differ across `MPAA_Rating` to see if this column will be helpful for modeling later. First, she calls values counts on the MPAA_Rating column to understand the distribution. SOLAS knows the data returned is pre-aggregated and plots the results in a bar chart. Most movies in this dataset are rated R, followed by PG-13. She then groups by `MPAA_Rating` and aggregates several other columns. Since her earlier exploration revealed the skewed distribution of the `US_Gross` and `Worldwide_Gross` columns, she aggregates them by their median across `MPAA_Rating`. She also calculates the mean of the Rotten Tomatoes Rating and Running Time minutes. When visualizing these results, SOLAS automatically includes error bars for the mean calculations to give more context (Figure 5.5C). Our analyst notices that the Rotten Tomatoes Rating column has similar standard deviations across ratings, except for the Open category, which has a much smaller standard deviation on the visualization.

5.6.2 Improving Visualizations with Type Inference Updates

To continue transforming her dataset, the analyst inspects the `Viewership` column (Figure 5.4 Top). Since this column has low cardinality, SOLAS initially infers the type to be nominal. However, the analyst knows that this column represents the viewership in units of 10 million, so she multiplies the column by 10 million to get the raw viewership count. After this operation, SOLAS has evidence that `Viewership` is a quantitative column and updates the type and visualizations accordingly (Figure 5.4 Bottom). This operation-based type update would not be possible without tracking and reasoning about analysis history.

5.6.3 Surfacing Visualizations with Column Interest Model

Our analyst turns her analysis towards the `Worldwide_Gross` column, since she will be using this column for predictions. Her earlier analysis suggested this column is right skewed, and has a large number of high-value outliers as shown in the boxplot for `describe`. Still, she wants to recheck the distribution again to confirm. She simply references this column in a cell by typing `df_movies.Worldwide_Gross` and SOLAS plots both the distribution of this column as well as `Worldwide_Gross` relative to other columns in the dataset. SOLAS sorts these visualizations so that columns she has recently interacted with appear first such `Viewership` or `MPAA_Rating` (as reflected by the ranking at execution count `[10]` in Figure 5.2).

After looking at the histogram, she creates a predictive model for high-grossing movies. She applies a filter to `Worldwide_Gross`, and SOLAS shows how this filtered data compares to the unfiltered set (Figure 5.5D). She can toggle the background distribution on and off to inspect more closely the returned data with or without this additional context. Once again, this background context would be impossible without SOLAS's history of her analysis. She iterates on her filter and decides on a value of $100M for her threshold since the SOLAS filter plots revealed that about a quarter of movies earn more than this much. Our analyst then creates a new binary variable called `Class` for whether or not a movie makes more than $100M. She will be using this variable as the prediction target for a binary classifier. She visualizes her data once again by displaying the dataframe with a call to `df_movies` (Figure 5.1). Since she has most recently created the `Class` variable, this action is highlighted in SOLAS and a bar chart for `Class` is shown. Furthermore, by looking at the enhance tab, she can see `Class` relative to other variables in the dataset. These recommendations are sorted by variables she has interacted with recently so `Class` vs `Worldwide_Gross` is shown first followed by `Class` vs `Viewership` and so on.

Finally, our analyst calls `df.corr()` to see how the other columns in her data are correlated with her new `Class` column (Figure 5.3). She notices several features have a strong correlation with the `Class` such as `US_Gross`. In contrast, others like `IMDB_Rating` have a weaker correlation, so they likely provide less predictive value. With this, our analyst is happy with her data exploration and is ready to begin modeling. By using SOLAS, she was able to spend less time thinking about how to visualize her data and more time focusing on the insights of her analysis.

Figure 5.5: SOLAS tracks history throughout an analysis to provide improved visualizations. We show four snapshots from an example analysis that demonstrate the recommendations from SOLAS. (A) When an analyst calls `describe`, we visualize the returned data as a boxplot by using information that is no longer present in the returned data to plot outliers. (B) After cleaning their data by dropping columns and nulls, SOLAS shows how the distribution of other columns change. (C) For groupbys and aggregations, we use history information to add better x-axis labels and include error bars when plotting the mean. (D) When filtering, SOLAS shows the background distribution of each column from the parent data relative to the filtered data. Users can toggle the background distribution on and off.

5.7 Evaluation

To evaluate how well SOLAS suggests visualizations that users find helpful on real-world tasks, we ran a survey to assess if users preferred the task-specific encodings provided by SOLAS versus the default encodings shown in Lux. We use Lux as an example of visualizations that will be presented by a state-of-the-art recommendation tool that does *not* use history. Our evaluation demonstrates the value of incorporating analysis history into visualization recommendation as SOLAS suggests preferred encodings for several tasks. We chose to evaluate the task-specific encodings since they were most easily assessed by crowd workers and do not require an entire analysis context to be useful; we demonstrate the utility of our model of column interest in Section 5.6.

We recruited 87 participants from the crowd-working site Prolific who attested to having some experience working with Python and the Pandas library. For participants to be eligible, they were required to correctly answer at least one quiz question assessing their familiarity with the Pandas API.

Participants were introduced to an example analysis task analyzing data about athletes from the 2016 Summer Olympics. This dataset has 13,688 rows and 14 columns, such as the athlete's height, weight, age, country, sport, and whether or not they won a medal. The survey was split into five sections where participants were shown a function call for the `describe`, `corr`, `groupby`, `isNull`, or `filter` tasks along with a preview of the data returned from this function. Participants were asked which of two recommendations they preferred for this data: the SOLAS task-specific encoding, or the default Lux encoding that did not leverage analysis history. The ordering of the visualization choices was randomized.

Participants' preferences are summarized in Figure 5.6. We conducted t-tests to assess if the fraction of responses was significantly different than 0.5 (which would indicate no preference). For `describe`, `corr`, and `groupby` participants significantly preferred the SOLAS encodings. For `describe`, participants preferred how SOLAS's boxplot matched the descriptive statistics: "[Solas] actually shows a distribution to the descriptive statistics, so we can see if there's any skew/outliers/etc." (P30). Interestingly, for `describe` Lux re-aggregates the data and presents a misleading histogram that assumes the data is a normal quantitative column. However, 21% of participants still preferred the histogram since it "Just seems easier to analyze and see" (P26). This underscores the importance of communicating data with appropriate encodings since users will interpret the chart even if it is *visualizing irrelevant data.*

Participants preferred the correlation matrix shown by SOLAS because they found the heat map "helps to see trends where they might not be obvious" (P64) and found it "Cleaner to have it in a single visualization, and the correlation matrix makes it easier to compare values " (P67). The correlation matrix has higher information density; most users prefer having the data communicated in a single visualization with higher information density. However, others still preferred bar charts showing the correlations relative to a single variable since they found it "easier and faster to read" (P27). The groupby visualizations were very similar except SOLAS's included error bars for mean charts and more descriptive axis labels. As indicated in survey responses, many participants preferred these subtle differences.

For the `isNull` task, users were ambivalent about which encoding they preferred with

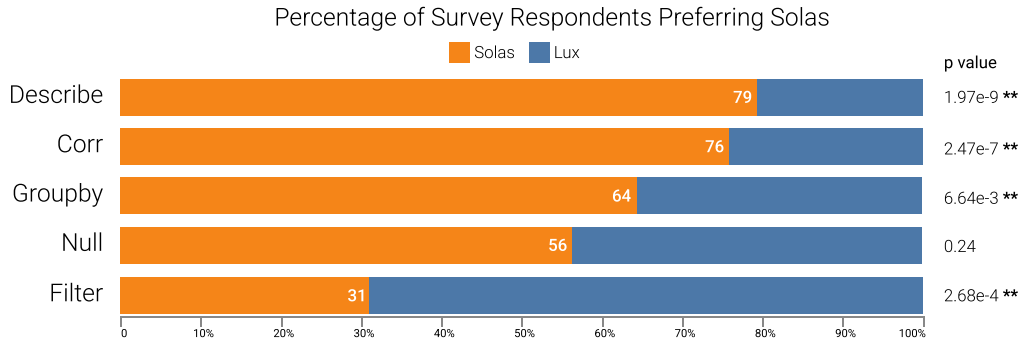

Figure 5.6: Survey participants (N=87) significantly preferred SOLAS's encodings for `describe`, `corr`, and `groupby`. For `isNull`, they found either encoding equally acceptable. For `filter`, participants significantly preferred the Lux encoding and are given the option to toggle the background distribution on and off in SOLAS. P values marked with (**) are below 0.05 and considered significant.

a non-significant difference in preferences. The SOLAS visualizations are subtly different than those provided by Lux. SOLAS visualizes the data as a stacked bar for each column showing the amount of null values; Lux shows a bar chart of the sum of True and False values. Participants preferring SOLAS remarked "if the context is apparent, why use true or false? In [Solas] we know the emphasis is on the number of missing records" (P93). However, others preferred Lux's encoding since it is more faithful to the data rather than the task at hand, "I think it's more clear in [Lux] approximately how many True/False nulls there are, while [Solas] is a relative comparison" (P30).

Finally, for the filter task, users were shown data from a filter that selected only athletes who participated in the Athletics event at the Olympics. Participants significantly preferred the Lux filter encoding to the SOLAS encoding that showed the results relative to the background distribution. In the actual SOLAS system, users can toggle the background distribution on and off so users can view both of these encodings. Users that preferred SOLAS claimed "This chart shows me how much the Athletics population is part of the overall population and their metrics as compared to others. Really cool chart" (P60). However, most participants found the extra context unhelpful: "I think it's better to focus on the extracted data rather than have it being compared to the entirety of the dataset" (P13). Future investigations might explore when this additional context is most helpful. By supporting toggling the background distribution on and off, we believe the design of SOLAS addresses many of the concerns from participants.

5.8 Limitations

This system and study is subject to several limitations. As a system, SOLAS currently only tracks history for Pandas dataframes and works in Jupyter (or similar) notebooks. However, we believe the ideas of using analysis history tracking to augment visualization recommendation are applicable beyond Python and Pandas programming. Our evaluation focuses on how crowd workers successfully understand SOLAS's task-specific visualizations. How-

ever, further studies might explore how systems augmented with history tracking like SOLAS help analysts explore their data on more in-depth analysis tasks.

5.9 Discussion

SOLAS demonstrates how history-based visualization recommendations can improve the experience of users as they iterate during exploratory data analysis. We believe that integrating history tracking into other visualization tools can provide similar benefits. Even in tools where users are not writing code, they still take actions similar to those accomplished through Pandas for SOLAS such as looking at an overview of their dataset, applying filters, and aggregating. Future work can use the same task-specific visualizations from SOLAS from history tracking and recommend in other settings such as no-code tools or other programming languages such as SQL or R.

5.9.1 Preventing Erroneous Findings

In building and evaluating SOLAS, we noticed trends around how users interact with their data and analysis histories. During our evaluation, some users still preferred the poor encoding of the aggregated data even though the chart communicated false findings such as re-aggregating pre-aggregated data into a histogram. By ensuring proper task-specific visualizations, SOLAS can help make sure that data analysts engage with their data truthfully and are not led astray by poor encoding. This finding echoes similar research from the XAI community about how users trust interpretability visualizations of a machine learning model even if the results are false [77].

5.9.2 Next Step Recommendation

Once we have detailed information about an analyst's steps during their data exploration, we can use this information beyond visualization recommendation. By aggregating across multiple analyses, we can begin to recommend potential next steps during recommendation with accompanying visualizations. Existing work in this area typically mines Jupyter notebooks found on Github to understand how users go about their analysis [131, 174], however, notebooks found on Github are often incomplete, or do not run [140]. Furthermore, they do not represent the full breadth of analysis since only one snapshot is uploaded that may not contain previous analysis paths that have been deleted from the notebook. By using SOLAS, we can track detailed information about the full breadth of a user's analysis and use this data to provide improved next step recommendations.

5.9.3 Capturing Interest Across Multiple Analyses

In addition to next step recommendations, we can use aggregated analysis histories to better understand how analysts typically interact with a particular data source. Many teams interact with (versions of) a remotely stored data source. Each of these analyses can be tracked through SOLAS to build a model of how users interact with that data *across* analyses. When

a user begins a new analysis, we can help them bootstrap their exploration by demonstrating how people typically explore or interact with that data source. We could even develop analysis templates based on common practices for single data sources or within a domain.

5.10 Conclusion

In this chapter, we introduced SOLAS, a system that enhances tabular data profiling by incorporating users' analysis histories into visualization recommendations. By tracking interactions with the Python Pandas library, SOLAS models user interests across different data attributes and provides task-specific data overviews and prioritized visualizations tailored to ongoing analytical tasks. Recognizing that data analysis involves iterative queries, SOLAS adapts visual overviews to fit the user's workflow, ensuring that visualizations are relevant and context-aware. This adaptive approach facilitates more effective data profiling compared to traditional methods that rely solely on static data types.

SOLAS extends the ideas of Interactive Data Profiling to develop a tool that:

1. Like AUTOPROFILER, is situated directly in the computational notebook environment
2. However, rather than only considering the current data in an analysis, SOLAS tracks the full history of the analysis. We show how to implement history tracking by extending a data manipulation library, then demonstrate applications of history tracking for task-specific visualizations, ranking overview visualizations, and inferring data types

Chapter 6

TEXTURE: Structured Exploration of Text Datasets

This chapter is adapted from the following paper:

> [37] Will Epperson, Arpit Mathur, Dominik Moritz, and Adam Perer."Texture: Structured Exploration of Text Datasets". *In Submission to IEEE VIS*. 2025.

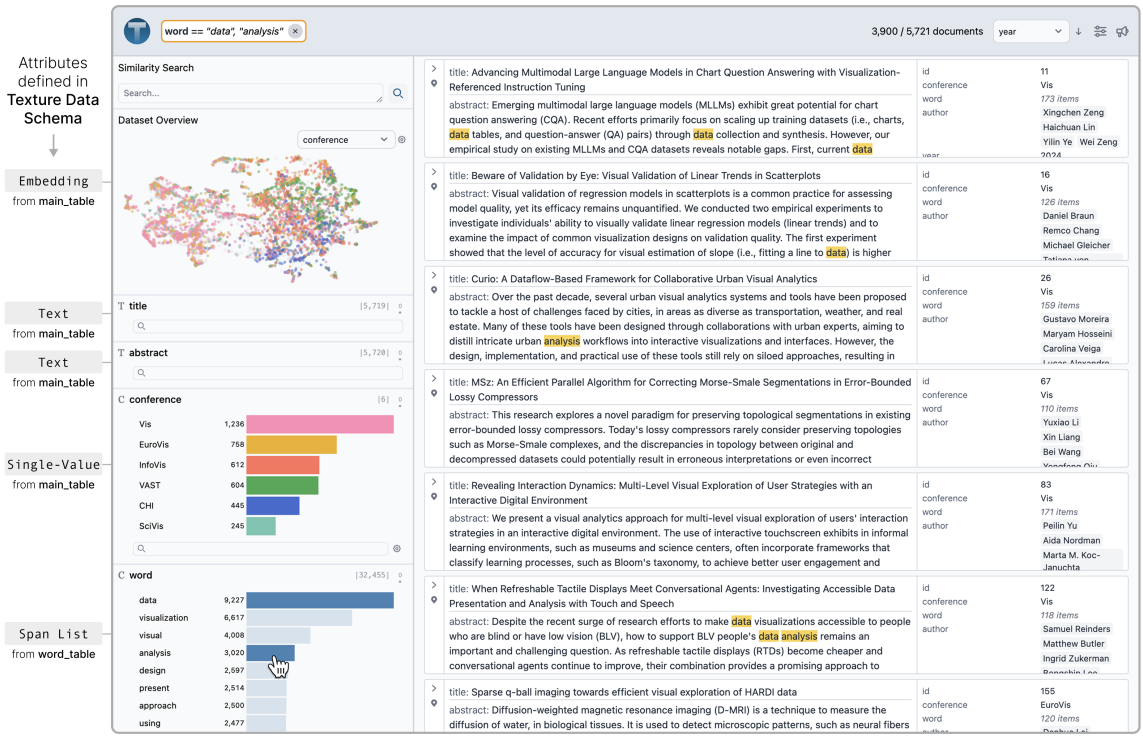

Figure 6.1: TEXTURE helps users explore text datasets through structured descriptive attributes. Its configurable data schema supports attributes at any level of granularity in the text, such as document-level attributes like the conference and embedding or word-level counts shown in this example analysis of an abstract corpus. The system organizes list attributes like words with multiple values per document into new tables and then joins tables to enable scalable filtering. TEXTURE helps users explore their data through attribute overview visualizations, interactive filtering, embedding overview and search, and contextualizing filters in the document text.

6.1 Summary

Exploratory analysis of a text corpus is essential for assessing data quality and developing meaningful hypotheses. Text analysis relies on understanding documents through structured attributes spanning various granularities of the documents such as words, phrases, sentences, topics, or clusters. However, current text visualization tools typically adopt a fixed representation tailored to specific tasks or domains, requiring users to switch tools as their analytical goals change, slowing down analysis and introducing unnecessary friction. This presents an opportunity to design general-purpose, lightweight interactive tools that can be used across text datasets to increase the feedback cycle during analysis in the spirit of Interactive Data Profiling systems. However, the Interactive Data Profiling tools developed up to this point in this thesis, AUTOPROFILER and SOLAS, only target tabular data and thus cannot support the full range of attributes needed to understand unstructured text datasets.

To address these limitations, we present TEXTURE, a general-purpose interactive text profiling and exploration tool. TEXTURE introduces a configurable data schema for representing text documents enriched with descriptive attributes. These attributes can appear at arbitrary levels of granularity in the text and possibly have multiple values, including document-level attributes, multi-valued attributes (e.g., topics), fine-grained span-level attributes (e.g., words), and vector embeddings. The system then combines existing interactive methods for text exploration into a single interface that provides attribute overview visualizations (inspired by those presented in AUTOPROFILER), supports cross-filtering attribute charts to explore subsets, uses embeddings for a dataset overview and similar instance search, and contextualizes filters in the actual documents. We evaluated TEXTURE through a two-part user study with 10 participants from varied domains who each analyzed their own dataset in a baseline session and then with TEXTURE. TEXTURE was able to represent all of the previously derived dataset attributes, enabled participants to more quickly iterate during their exploratory analysis, and discover new insights about their data. Our findings contribute to the design of scalable, interactive, and flexible exploration systems that improve users' ability to make sense of text data.

6.2 Introduction

Understanding collections of text documents is a fundamental task across many disciplines. In the social sciences, researchers programmatically analyze trends from social media posts, news articles, and government reports [40, 47, 163], while in NLP researchers use text datasets to train and evaluate new AI models [74, 136]. With the rise of large language models (LLMs), text data is becoming increasingly available and important in many high-stakes applications [133, 141]. Whether for research, model development, or decision-making, users have the need to quickly understand the text in their datasets, and evaluate the results of their analyses on the text [133, 157].

Since manually reading all documents in a corpus is infeasible, analysts must transform text data into shorter, more interpretable representations [35]. Many techniques exist to derive these shorter representations, ranging from extracting words and phrases, to tagging

documents with topics or classes [99]. While some descriptive attributes like words or topics are generally useful across datasets, many meaningful representations are task and dataset specific. Over the course of an analysis, users often experiment with different representations of their data as their questions and goals evolve.

However, many prior text visualization tools focus on particular domains or tasks where the descriptive attributes are predetermined. For example, domain-specific systems for social media content analysis [4, 16], academic literature review [7, 114], or NLP model evaluation [74, 148] all extract predetermined representations. Likewise, task-specific systems for word or topic understanding operate on a pre-determined level of granularity [41, 133]. These specialized tools enable deep analysis of specific questions, however lack the flexibility needed to adapt to varied datasets or shifting analytical goals during exploratory analysis [71]. Nevertheless, many exploratory interactions—such as filtering data subsets, searching for textual patterns, or exploring document embeddings—are common across tasks and domains. This fragmented landscape motivates our central research question: *How can we build a general-purpose, interactive text exploration tool?*

To address this question, we first introduce a configurable data schema for describing text data and associated descriptive attributes and then present an interactive interface for exploring a text dataset through its configured attributes. Our schema describes attributes along two primary dimensions: their relationship to the document (i.e., Does the attribute correspond to the entire document or only particular spans?) and the cardinality of the attribute (i.e., Does the attribute have a single value per document or multiple values?). Accordingly, our schema categorizes attributes into five distinct types: text documents, single-value attributes, list attributes, span list attributes, and vector embeddings. These types cover common text representations from different levels of granularity in the text like topics (a *single-value* attribute that describes a document), authors (a *list* attribute with multiple values per document), or frequent words and n-grams (*span list* attributes that correspond to parts of the text). Additionally, the proposed data schema prescribes how to split multi-valued list attributes into new relational tables to enable interactive filtering and exploration at scale.

We then designed TEXTURE, an interactive text exploration tool based on this configurable data schema. TEXTURE combines interactions from previous text analysis systems into a single general-purpose tool. TEXTURE is designed around three primary interactions: the ability to provide an **overview** of a text dataset through structured attribute visualizations, **filter** attribute visualizations for exploration, and link visualizations back to the raw documents to **contextualize** results. TEXTURE automatically generates interactive overview visualizations for each attribute in a dataset that support cross-filtering to explore insights across attributes. Attribute visualizations link to a view of the raw documents, helping users contextualize summaries and filters in the actual document texts. Finally, TEXTURE incorporates embedding-based operations such as similarity search and dimensionality reduction overviews to enable fuzzy search and a dataset summary. TEXTURE integrates seamlessly with Python-based data workflows, allowing analysts to programmatically define and manipulate descriptive attributes before exploring them interactively. While prior text visualization systems contain subsets of these features, TEXTURE's novelty is the combination of features with a configurable data schema that enables exploration

of many different types of text datasets. TEXTURE is open-sourced and available for use [1].

We evaluate TEXTURE through a two part user study with 10 participants. Each participant brought their own dataset they had previously analyzed to ensure higher validity of our study. In the first session, they walked us through a baseline analysis of their data using their current workflow. Participants used many different kinds of structured attributes to understand their text. However, they only inspected small samples of the text in their dataset and were slowed down by the need to manually define charts through code and their inability to link them back to the text.

In the next session, each participant analyzed their data using TEXTURE. Across all 10 distinct datasets, our results show that TEXTURE helped users effectively explore their data and uncover new insights. Participants quickly explored different hypotheses and iteratively developed meaningful analysis questions with TEXTURE. Our system and study inform the design of configurable, general-purpose tools that better support users in exploring text datasets across diverse domains. In summary, this chapter contributes:

1. A configurable data schema for describing text attributes from arbitrary levels of document granularity and cardinality.
2. TEXTURE, an interactive text exploration tool that helps users explore their data through overview visualizations, filtering, and contextualizing descriptive attributes.
3. Results from a user study that show how TEXTURE is expressive enough to analyze datasets from 10 different tasks/domains and helps each user effectively explore their data.

6.3 Related Work

Our paper builds on related work on interactive systems for exploratory data analysis and text visualization.

6.3.1 The Need for Exploratory Text Analytics

Text data inherently lacks the structure necessary for straightforward visual analysis and must be processed into meaningful, structured attributes for exploration [133]. Recent studies underscore the necessity of understanding both individual documents and their descriptive attributes within both local (individual document) and global (full corpus) contexts [57]. Even basic document metadata, such as the most frequent words, n-grams, or domains, can yield significant insights into a corpus and its quality [35]. This form of corpus *profiling* is essential both for analytical purposes as well as AI model training, where data quality is often undervalued and prior analyses have found that popular NLP benchmark datasets still contain poor quality or mislabeled data [141, 155].

Profiling dataset metadata attributes is a core aspect of EDA workflows where analysts summarize their data with visualizations to generate initial hypotheses for further analysis [156, 165]. Prior interactive visualization systems for tabular data facilitate EDA by automatically constructing visual representations of the data and enabling interaction to

[1] https://github.com/cmudig/Texture

quickly filter and find meaningful data subsets [76, 167]. Automatic visualization combined with interaction enables a fast feedback loop between asking questions about data and interpreting the results [36]. While these approaches craft effective visual summaries of attributes, they lack the ability to contextualize attribute summaries in the actual documents needed for EDA over text.

6.3.2 Exploring Text Through Structured Attributes Across Levels of Granularity

Understanding a text corpus requires navigating information across multiple levels of granularity. We review prior approaches for understanding documents across three common levels—words, documents, and the entire corpus—and their influence on the design of TEXTURE.

Word Frequency and Highlighting

Words offer a natural starting point for summarizing text documents into a more concise representation. Techniques such as word clouds and parallel tag clouds display frequently occurring words or sequences across a dataset, potentially faceted by structured attributes [23, 42, 160]. Since the significance of individual words often depends on their surrounding context, systems frequently use highlighting to link words back to the surrounding context of the original document [25, 34, 112, 117, 153]. Word-level tags like entities or part-of-speech are also common attributes for analysis. For example, in Jigsaw users investigate the relationship between entities across documents in a dataset [151], and Automatic Histograms presents a technique for clustering entities into meaningful semantic groups using LLMs [133]. Beyond individual words, other research summarizes text documents through short phrases that match a particular query or linguistic pattern [132, 169]. TEXTURE builds on prior word analysis techniques by abstracting these into a span list attribute where values correspond to any arbitrary segment of the text and individual occurrences are highlighted.

Document-Level Attributes

Beyond words, other research considers how to navigate a corpus through attributes assigned to each document, most commonly topics. Topic modeling techniques represent each document as one or more topics, each summarized by characteristic words. Latent Dirichlet Allocation (LDA) is a foundational method using this approach [15]. Interactive systems help users explore a corpus through topics by showing the most frequent topics, filtering to documents that match a particular topic, and understand how topics evolve over time [2, 41, 98]. Domain-specific systems like PaperLens linking topic visualizations directly to a predefined set of structured attributes [94]. With word-based topic techniques, users often struggle to interpret abstract topics represented by lists of potentially ambiguous words. Recent methods like LLooM improve interpretability by framing topics through clear inclusion criteria generated by LLMs, such as direct questions (e.g., "Does this text discuss sports?") [92]. TEXTURE generalizes techniques for understanding topics to any document-level attribute by showing frequent values and helping users explore how document-level attributes correspond to common words or phrases.

Corpus Overview Methods and Embedddings

At the highest abstraction level, corpus visualization techniques enable exploratory analysis across entire document collections. Many of these systems incorporate structured metadata alongside raw text. Leam provides an interface for applying basic text data transformations (e.g., text length extraction) then visualizing the results [51]. TextTile defines three fundamental analysis operations—filter, split, and summarize—allowing users to compare keyword summaries and attribute visualizations across facets of their dataset [43]. Systems like LLMComparator and Vitality help contextualize dataset attributes by linking document views and structured attribute visualizations for particular types of text data like LLM outputs or paper abstracts [74, 114]. Our work extends these prior corpus-level approaches by considering a broader set of attribute types, then enables similar interactions for users to filter and explore subsets.

In addition to structured attribute summaries, document embeddings are commonly used to understand a corpus. Representing documents with high dimensional embedding vectors has become a common technique to capture both the syntax and semantics of the document in a single representation [33, 121, 134]. These embeddings can then be used for many useful analysis tasks, such as projecting the embeddings down to two dimensions to enable an overview visualization using techniques like UMAP [106], or finding nearby instances in the embedding space. Prior interactive systems help users explore documents through embeddings. For example, DocuCompass helps users link subsets of the embedding space to inline structured attribute overviews [62]. Angler, an interactive visualization tool for prioritizing machine translation errors, uses UMAP projections to provide an overview of the data while overlaying the scatterplot with additional information, such as usage logs [136]. Similarly, WizMap tightly integrates a projection view with automatically-generated multi-resolution summaries to help users navigate through the large embedding space of documents [162]. TEXTURE adopts similar techniques to these prior approaches by including embeddings as a fundamental data type, used for a projection overview and finding similar instances. Like prior work, TEXTURE helps users make sense of embeddings through the structured attributes in the corpus.

6.4 TEXTURE: Interactive Exploratory Text Analysis

TEXTURE builds on prior work for understanding text at different levels of granularity such as words, document-level attributes, and structured corpus metadata in two primary ways. First, TEXTURE presents a configurable data schema for representing different kinds of structured attributes. Unlike prior systems that use a fixed set of attributes or only consider attribute types relevant for a particular analysis task, TEXTURE's attribute schema is highly configurable to analyze different types of descriptive attributes across datasets. Second, TEXTURE enables exploratory analysis through interactions designed around these descriptive attributes. These interactions build on methods from prior work to enable users to explore their data through attribute overview visualizations, filter and compare filters on attributes, use document embeddings, and relate attributes back to the document text. We first describe the data schema underlying TEXTURE, then our system.

6.4.1 Configurable Attribute Schema

Attribute Type	Description	Examples
`Text`	The text documents	Paper abstracts, news articles, LLM prompts
`Single-Value`	Descriptive attribute that only has one value per document and is *numeric*, *categorical*, or *temporal*	Publication date, document sentiment, topic
`List`	Descriptive attribute with multiple values per document	List of authors, keywords, or topics
`Span List`	Descriptive attribute with multiple values per document where each value maps directly to a *span* of the text	Words, tokens, phrases, part of speech tags
`Embedding`	High dimensional embedding + a 2D projection	SBERT embeddings [134] with UMAP projection [106]

Table 6.1: The semantic data schema used in TEXTURE describes different kinds of attributes and their relationship to text documents.

TEXTURE formalizes attribute definitions with a configurable data schema. This schema supports common representations of text at different levels of granularity—including words, phrases, n-grams, topics, or document tags—along with arbitrary user-defined attributes. Expressing attributes in the TEXTURE data schema makes the system highly configurable for exploring different datasets.

The TEXTURE data schema describes attributes according to their cardinality and how they correspond to the text. The five types in this schema are presented in Table 6.1. The first type is `Text` data which describes the documents in a dataset. Text data can be documents of any length from social media posts to books. A single instance might have multiple text attributes. For example, paper titles and abstracts, Spanish to English translation pairs, or LLM prompts and responses.

The other four types in the schema categorize descriptive attributes. Single-value attributes have a single value per document and are numerical, categorical, or temporal. These describe document-level attributes such as a single topic tag, publication date, or sentiment score. This is distinct from multi-valued attributes that are lists. Many common descriptive attributes are lists. For example, each document can be divided into a list of words or tokens. Likewise, a single document may have a list of topics or authors. Any attribute with multiple values is a `List` attribute. If each attribute item corresponds directly to a span of the text it is a `Span List` attribute. Words, tokens, or phrases are all examples of span list attributes since they correspond to a specific span segment of the document. Word-level tags such as part of speech tags or entity tags can also be represented as span list attributes. Span list attributes enable TEXTURE to capture the hierarchical nature of text. Documents can be broken down into arbitrary levels of granularity and then annotated with other possible data. All that is necessary is to maintain which index the segment comes from.

The last type of our data schema are `Embedding` representations of the text. Representing embeddings as an atomic type in TEXTURE allows users to choose the model and projection method they prefer, and then analyze the results in the system. Currently, TEX-

TURE only supports a single embedding per instance but could be extended in the future to support multiple.

TEXTURE does not automatically derive any descriptive attributes from the text; instead, it aims to decouple attribute derivation from representation and exploration. Meaningful attributes are often dataset and task-specific and comprise a huge space of possible attributes. TEXTURE is un-opinionated about how attributes are derived, allowing users to write whatever code or use whichever models best fit their task. Once they have attributes, they can describe them with the TEXTURE data schema and then explore their results in the system.

Representing Attributes in Relational Tables

Our semantic schema prescribes how data should be formatted in physical tables to enable exploration in TEXTURE. A key goal of this representation is to support the different data types from the schema in Table 6.1, while also enabling scalable interactions through relational query languages like SQL. While modern DataFrame libraries like Python Pandas [119] permit arrays or tuples in columns, most relational databases require that tables are in first normal form and each attribute has only a single value [22]. Normalizing attributes into different tables makes it easier to visualize attributes and perform interactive filtering at scale with SQL [61].

Therefore TEXTURE splits list attributes into new data tables that are linked back to the documents. Figure 6.2 shows what this representation looks like for an example input dataset with different attribute types. In this example, we have five attributes: one text, two span lists, one list, and one single value. To achieve normalized tables, each of these list attributes is split into a new table while text and any single valued attributes stay in the same table. In this example, each document and topic tag remain in the main table. However the word, part of speech tags (POS), and authors are all lists which are parsed into new tables.

These tables enable representing hierarchy in the data. Each row in the main table has a unique id while the list attributes maintain a foreign key relationship to the ids. Furthermore, for span list attributes, each table maintains the spans of the original document that the entry corresponds to. These span indices are used for highlighting the text in the interactive system. Regular list attributes maintain the array index.

6.4.2 *Overview:* Automatic Attribute Visualizations

Providing a dataset and schema to TEXTURE enables immediate exploration in the UI. Figure 6.1 shows TEXTURE with the VisPubs dataset that was also used by one of the participants in our user study to understand visualization abstracts and is used throughout our examples [93].

The first component of this UI is the automatic visualization of each attribute into an interactive overview visualization. These visualizations enable quick overviews of each attribute and support interactive filtering to enable users to explore documents in different subsets of the data. Once formatted into separate tables, each attribute becomes tabular data. We therefore designed the attribute overview visualizations similar to previous tabular

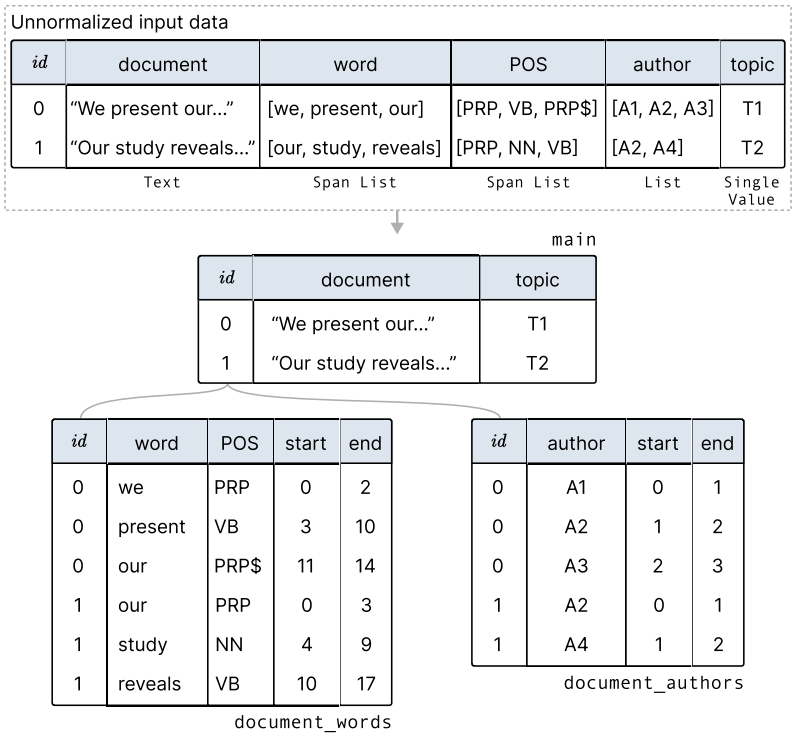

Figure 6.2: Following the TEXTURE data schema requires placing list attributes into new tables that map back to the documents.

data overview tools and text visualization systems that show structured attribute visualizations [36, 74, 114].

TEXTURE automatically visualizes attributes according to their data type: text, quantitative, categorical, or temporal. Example attribute visualizations are shown in Figure 6.3. Text columns are displayed in the main document view whereas categorical attributes are visualized. Users can change the data schema to configure how attributes are displayed. We produce a summary visualization for each attribute according to its data type:

- **Quantitative** attributes are visualized as binned histograms.
- **Categorical** attributes are visualized as sorted bar charts of the counts of the 10 most frequent values (more on-demand)
- **Temporal** attributes show line charts of the count over time.

While each of the attributes might come from different tables, TEXTURE shows the visualizations as a flat list to facilitate comparison between attributes while filtering. This allows users to make corpus-level observations from list attributes. For example, observing the most frequent words across the entire corpus or the most frequent author from lists of authors.

In addition to individual attribute visualizations, TEXTURE provides a dataset overview plot through a scatterplot of 2D embedding projections. This sort of dataset overview has become a common data overview technique for unstructured data like text in both research and commercial systems [62, 115, 162]. The projection view can also be colored by any of

the single value categorical attributes in the dataset to help users plot clusters in the dataset. For example, the embedding overview in Figure 6.4 is colored by the conference attribute.

6.4.3 *Filter:* Exploration Through Linked Visualizations

After attribute visualizations provide an initial overview, TEXTURE enables users to explore their data by filtering to different subsets. Users can filter their data in several ways in the interface. First, they can apply selections to visualizations to filter to a particular value or range of values. As shown in Figure 6.3, users can apply selections to categorical summary charts like the `word` chart by selecting one or more bars, in this case the words "data" and "analysis". For quantitative or temporal charts, users can filter to a certain range with a brush.

Attribute visualizations are cross-linked, meaning that filters applied to one visualization filter the data in all the other visualizations, including the projection overview chart. This enables insights beyond single attribute summaries by exploring interactions between attributes. For example, after filtering to certain values of `word` in Figure 6.3, a user can understand how these words are distributed across the `conference` and `year`. Or a filter brushed across different years in the `year` chart shows how the top words or conferences change across the years. Filters can be applied to multiple charts at once, enabling questions about increasingly specific subsets of the data.

Users can also apply filters to the dataset overview chart to understand frequent attribute values for different regions of the projected embedding space. Prior embedding visualization systems have directly labeled charts with frequent words or metadata values in different regions of the embedding space [62, 162]. TEXTURE does not label the embedding view directly, however enables a similar insight by cross-linking the embedding view to attribute visualization charts. This allows users to interpret the embedding space through any of the attributes in their data, rather than just words.

Beyond filters on charts, TEXTURE also supports search and embedding-based similarity search. With the search bar, a user can search for a specific phrases in any of the text (or other) attributes and once again cross-filter the data. Similarity search is supported in two ways, shown in Figure 6.4. The first is open-ended similarity search where after a user enters a query, TEXTURE calls a user-specified model to compute the embedding for the query, and calculates the cosine distance to each instance's embedding. The second form is similar instance search where a user clicks the show similar button on an instance, then the system computes the cosine distance from this instance to all other in the corpus. The result from either of these interactions is a new attribute that shows the similarity search result. Like any other quantitative attribute, users can brush to different regions of similarity or sort their dataset by this value to find the most, or least, similar instances.

Since the values for `List` and `Span List` attributes in TEXTURE are stored in different tables, TEXTURE joins tables when relevant attributes are involved in a cross-filter selection. This feature enables insights across attributes from different levels of a document hierarchy such as the aforementioned example about filters over a span list attribute `word` with a document level attribute `year`.

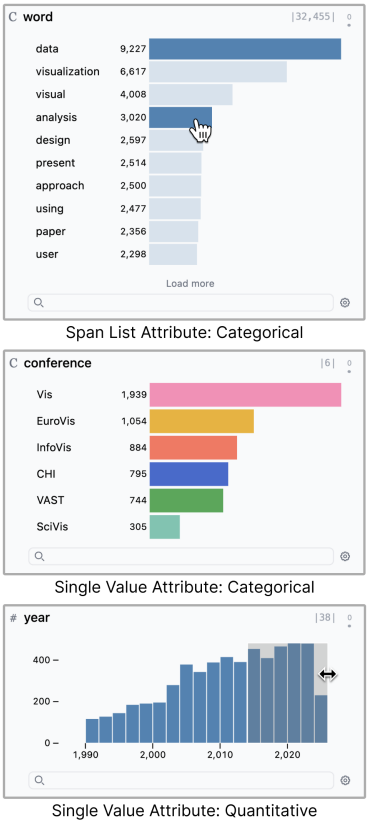

Figure 6.3: All attributes are automatically visualized according to their data type (quantitative, categorical, or date) regardless of if they are lists or single-valued. Attribute visualizations support interactive cross-filtering and can color the projection overview.

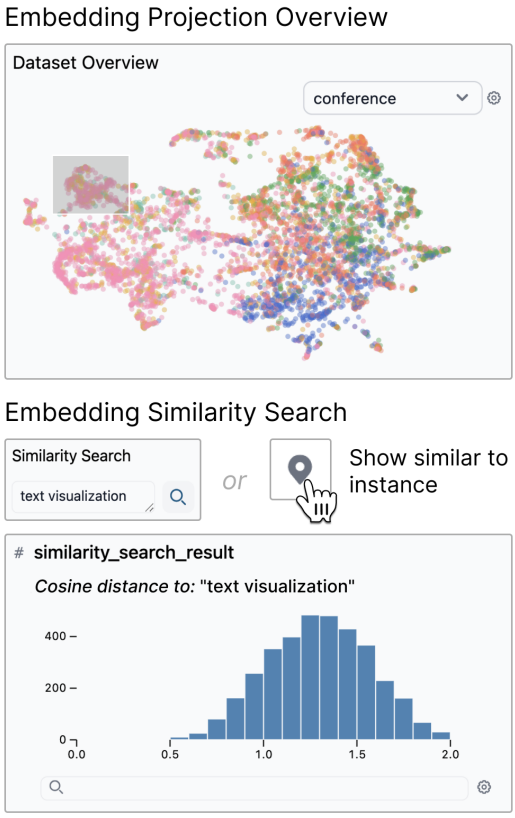

Figure 6.4: Document embeddings enable a projection overview and similarity search.

6.4.4 *Contextualize:* Linking Attributes to Documents

While attribute visualizations and filters help users make sense of the structured representations of their text, they also need to be able to contextualize attributes in the actual documents. By design, a document table view occupies most of the screen in TEXTURE. This allows users to quickly inspect individual instances and their attributes. The display table is scrollable with an entry for each instance that shows the text data and attributes. The document view also allows sorting by each attribute. Each instance shows the first five lines of text by default and can be toggled to show the entire document and other text attributes.

When filters are applied through attribute charts, the table view is filtered to the subset of documents that match the current filter. Combined with attribute cross filtering, this enables users to quickly understand the results of filters both in terms of other attributes and individual documents. For example, Figure 6.5 shows how a user might apply filters to different attributes like year or words to inspect the subset of documents with the word "data" or "analysis" between the years 2015 and 2025.

Similar to prior word visualization systems [2, 34], TEXTURE highlights spans in the text when filters are applied to span-list attributes. The span indices maintained in the TEXTURE data tables (discussed in Table 6.4.1) ensure accurate highlights and disambiguate between substring matches. For example, if a user filters to `word == "won"`, the span of this word is important to properly highlight only word matches. This way "we won the wonderful match" is highlighted and not "we won the wonderful match".

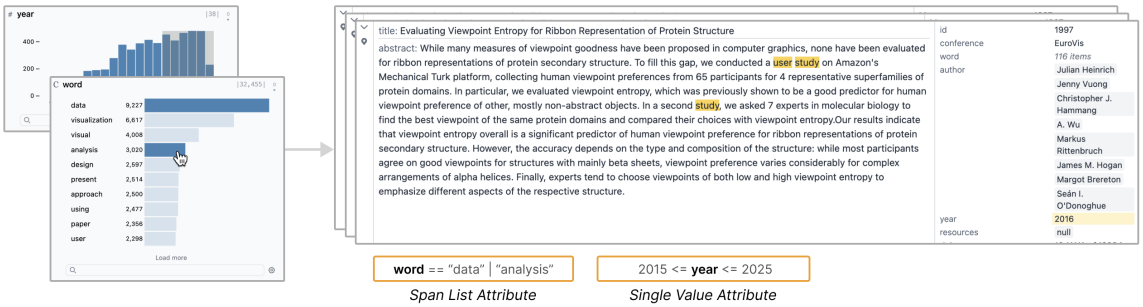

Figure 6.5: TEXTURE helps users **contextualize** attribute filters in the actual documents by showing documents that match current filters and highlighting the spans of text for filtered span list attributes.

6.4.5 Implementation & Integration into Python Ecosystem

The TEXTURE interface runs in the browser and is built with a Svelte frontend and a Python FastAPI backend. DuckDB SQL queries are generated by the frontend with queries coordinated by Mosaic [61]. DuckDB queries run in the backend. TEXTURE also uses a vector database, LanceDB, to store embeddings and calculate similarity distances between embedding vectors [30].

This Python-based implementation allows TEXTURE to integrate into the Python data science ecosystem. Python and computational notebook tools like Jupyter have become the most popular tools for programming with data [150]. Users can format their text data in Python, derive structured attributes to represent their data, then launch the TEXTURE UI either from a Python script or computational notebook.

6.5 User Study

To evaluate how well TEXTURE helps users performing different tasks understand their text data, we ran a user study. We had two primary research questions for this study:

Q1. *Expressivity:* How well can users working across different tasks represent their text datasets and descriptive attributes in the TEXTURE data schema?

Q2. *Effectiveness:* Does TEXTURE improve users' ability to understand their text, explore subsets, and find similar documents compared to their current workflows?

6.5.1 Procedure & Methods

To answer these questions, we designed a user study where users explored their own data in TEXTURE. Our study involved two separate one hour sessions: a baseline session and an experiment session.

The baseline sessions were structured as a semi-structured think-aloud interview and live-coding session, where each participant walked us through an analysis on their own text dataset and explained their analysis goals. Participants first completed a background questionnaire on their text analysis experience and listed out their initial analysis questions about the data. They then walked us through their analysis, showing us the kinds of visualizations and analysis steps they took to understand their dataset. Afterwards, we conducted a semi-structured interview to probe about their experience.

In the next part of our procedure, each of these 10 participants also explored their data in TEXTURE in a separate session. Participants analyzed the same dataset with TEXTURE as their initial exploration. This session consisted of a semi-structured exploration, where participants were asked to think aloud while exploring their data analysis goals in TEXTURE. Each participant sent their data to the research team before the session where we formatted the data into tables that fit the TEXTURE data schema. We included the same attributes that participants explored in their initial sessions, derived words from the text attributes with the spans if words were discussed in the baseline interview, and added document embeddings for the text using the OpenAI text-embedding-3-small model [118] if participants did not already provide them. Each session lasted one hour, including a short demo of the features of TEXTURE on a different dataset, 30 minutes for the participants to use the system, and an exit survey and interview. In the survey, participants compared their experience using TEXTURE to their baseline workflow. They then provided open-ended responses elaborating on their ratings during a post-task interview.

Participants were compensated $40 for the study, and the study protocol was approved by our institution's IRB. Both sessions were conducted over video call where audio and screen was recorded. Interview transcripts and recordings of each participants' analysis actions were analyzed using thematic analysis to identify common patterns.

6.5.2 Participant Details

To be eligible for the study, participants had to have experience working with text data and programming in Python, and be able to bring a text dataset they had previously used in their work or research. Participants were recruited through connections at our institution and advertisements on social media.

Table 6.2 details our study participants, their backgrounds, and high level analysis tasks. Participants had between 4 and 13 years of experience programming with Python (mean of 7.8 years), with self-reported expertise ranging from intermediate to expert. Eight out of ten of our participants worked with text data daily or weekly, with the other two working with text data on a monthly basis. Almost all of our participants brought datasets they had worked with in the past week, with only one participant bringing a dataset they had not worked with in over a year for confidentiality reasons.

Our participants primarily came from research backgrounds, analyzing research ques-

ID	Role	Data and Task	Attributes	Size	Med. # Words
P1	VIS Researcher	Paper abstract corpus for literature review	2 text, 10 descriptive	5700	164
P2	NLP Researcher	LLM prompts and responses with constraints	2 text, 3 descriptive	1250	201
P3	NLP Researcher	AI agent responses	2 text, 1 descriptive	100	207
P4	NLP Researcher	LLM reasoning traces on benchmark	3 text, 4 descriptive	2600	1402
P5	HCI Researcher	Paper abstract corpus for literature review	3 text, 4 descriptive	3500	215
P6	VIS Engineer	Song lyric corpus analysis	3 text, 8 descriptive	900	991
P7	Humanities Researcher	Historical book corpus analysis	1 text, 17 descriptive	600	4218
P8	AI Engineer	LLM chatbot user queries and responses	3 text, 5 descriptive	16000	431
P9	AI Engineer	Reddit social media post analysis	1 text, 10 descriptive	*15000	18
P10	NLP Researcher	LLM fine-tuning dataset creation	3 text, 8 descriptive	*5000	1024

Table 6.2: Participants in our study analyzed text data from a wide variety of domains and formats. We summarize the attributes in each dataset along with the size and median number of words in each document. *Indicates sample from a larger dataset.

tions about text datasets or text-based AI models. Some of their analysis goals were explicitly exploratory such as understanding themes in a corpus of song lyrics (P6) or getting a better sense of the literature in a field (P1, P5). Other participants were seeking to better understand text datasets as part of a larger model building or evaluation task (e.g., P2, P3, P4, P10).

Participants analyzed datasets varying significantly in terms of content, structure, and scale. For instance, P7 analyzed the richest dataset, which included extensive descriptive attributes accumulated over multiple prior projects. Conversely, P3's dataset had minimal metadata, specifically a single categorical attribute identifying agent names for a multi-agent analysis. The datasets varied considerably in size, ranging from as few as 100 documents to as many as 16,000. Two of our participants (P9 and P10) analyzed samples of larger training corpora.

6.6 Baseline Text Exploration Tasks and Challenges

For their baseline analysis, all participants were asked to show us how they explore their data through code and so spent most of the session stepping through their code-based EDA workflow. Participants also used tools like the huggingface built in data viewer [69] or spreadsheets to get an overview of their data. Two participants also used custom-built tools for exploration. P1 built a custom UI to inspect their data; P2 used the Zeno platform to

inspect and filter their data [18]. We describe some of the common analysis patterns and pain points from these baseline analysis sessions.

6.6.1 Inspecting Small Sample of Documents

A common task across domains involved inspecting a sample of individual data instances to generate hypotheses for further validation through exploration. 8 of the 10 participants examined a few examples of text, often the first few rows of the dataset, to identify initial patterns within the data:

> "*I think this is probably like the most common thing that I've done with almost every data set I've ever processed, which is you look at the first 30 rows and just read through these ones and you see a few things.*" - P8

However, with larger datasets, it becomes difficult to make sure these sample documents are a meaningful subset of the entire dataset. Since hypotheses are generated by examining only a few instances, analysts might overlook important details in their documents.

6.6.2 Exploration Through Structured Attributes

Although participants were analyzing the *text* in their data, each relied on different kinds of structured attributes to facilitate understanding their text. As shown in Table 6.2, every participant used at least 1 structured descriptive attribute in their data, with up to 17. The counts in this table represent the number of attributes that were in the data when passed to TEXTURE, so includes both attributes in the original dataset as well as attributes derived over the course of the baseline session we observed. For example, the abstract corpora analyzed by P1 and P5 each came with list attributes like the paper's authors, publication year, and conference. The song lyric corpus analyzed by P6 included attributes like the song's artist, year, and popularity. P7 described how their dataset and attributes were the combination of many prior projects where they had worked with the same collection of books.

Participants also derived new attributes to facilitate exploration. Four participants derived the frequent words or n-grams in their dataset. Three participants used the TextBlob package to quickly compute sentiment scores for their text [101]. Six participants mentioned having previously used, or the desire to use, LLMs as a way to easily derive attributes for things that are hard to define in code. For example, P10's dataset included three text attributes for LLM fine-tuning: a prompt, and two different model responses. They had previously used an LLM to extract the user-specified constraints on the output in each prompt and thus also had a list attribute for the constraints in each prompt.

Deriving useful attributes is an iterative process, with new attributes derived as new questions arise. However, with each new attribute participants described how it can become hard to actually contextualize attribute summaries in the documents. As P6 said when asked about what is currently the hardest part of their analysis:

> "*I think with text, it's converting words to numbers a lot. You're clustering, you're binning, you're counting, you're aggregating,* ***but you can lose the context very quickly.*** *So I would have to add a lot more code to just take one topic*

and then [look at] the lyrics for that topic. But if I could just select a topic and then say, 'Show me five songs that identify strongest with that topic'...then I could read it if I wanted to" – P6

6.6.3 Filtering Primarily Through Keyword Search

Filtering is a fundamental analytical task in data analysis, and a similar pattern emerged during the baseline exploration. Participants applied filtering operations to both derived attributes (e.g., using word counts to exclude instances with short text) and metadata attributes (e.g., removing publications with low citation counts).

Keyword search was a common and easy form of filtering the text used to validate previously generated hypotheses. Participants filtered individual instances based on the presence of specific keywords. However, keyword search was not always sufficient to fully address the hypotheses, and participants expressed a need for other search approaches when exact matches are hard to find:

"*I guess I can search for 'predefined category', but it's fuzzy and kind of hard to use regular expressions here. I just need to understand how often people do this type of sentence or a variant of this sentence trying to constrain the classification results.*" – P2

6.6.4 Barriers to Using Embeddings

The use of document embeddings for text similarity or overviews was not a common analysis task. Only one participant (P7) had previously computed embeddings for the dataset they brought. Participants expressed interest in incorporating word embeddings into their workflows but cited the effort required use these embeddings for analysis as a barrier to adoption. For example, P4 described:

"*Embeddings are great, but they're just hard to use...I think, honestly, [we have] TF-IDF and these simple things and word counts for a reason. So I just like quick and dirty.*"

6.7 TEXTURE Usage Results

In this section, we discuss themes about participant's usage of TEXTURE to explore their same dataset and how it compared to their baseline workflow.

6.7.1 TEXTURE Is Expressive Enough to Support All Participant Attributes

All of the diverse descriptive attributes from participants' baseline analysis could be represented in the TEXTURE schema and analyzed with TEXTURE. Most attributes captured straightforward, document-level information and were modeled as single-value attributes. However, eight out of ten participants also used list-like attributes with their datasets. Most of these were span list attributes for words from the different text attributes, but others included document-level lists such as authors (for P1 and P5) or lists of information extracted

from the text (for P2). This ability to handle varied structured attributes underscores the flexibility and expressivity of TEXTURE.

Exploring text data through the lens of structured attributes often inspired ideas for other useful attributes to add to the dataset. During the exploration session, six participants added new attributes to their data and then explored the results by writing code in the Python notebook. These were all simple attributes like the length of documents or number of words. However, participants also mentioned how they might add other attributes like document topics, use LLMs to quickly derive new attributes for exploration, or find other datasets online to add to their current exploration given more time.

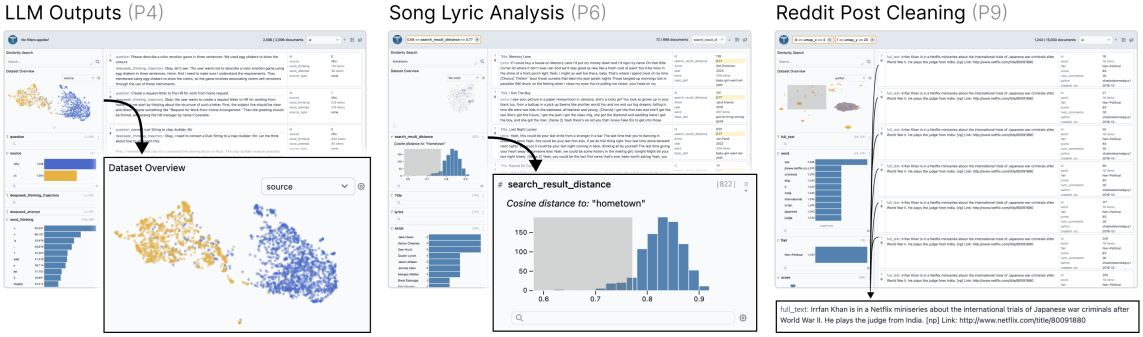

Figure 6.6: Participants used TEXTURE to explore a wide variety of datasets including LLM outputs, song lyrics, and Reddit posts.

6.7.2 TEXTURE Makes Prior Analyses Faster and Enables New Types of Analyses

With TEXTURE, participants were able to perform the same analysis actions as in their baseline exploration, but more quickly and easily. For example, every participant inspected attribute summary charts and applied multiple filters to attribute charts to explore subsets of their data. This involved the keyword searches typical in the baseline sessions (also used by $7/10$ participants in TEXTURE), along with filters over multiple attribute charts.

The automatic attribute visualizations in TEXTURE sped up the analysis compared to creating charts by writing code in Python. As P7 said, "This just gets me there so much faster". Overall, participants agreed that TEXTURE made it easier to understand their text (mean rating $4.4/5$) and filter to subsets of their data (mean rating $4.8/5$, see Figure 6.7). In interviews, participants often attributed this improved ability in understanding their text to the fact that TEXTURE encouraged them to read through more instances and made it easy to contextualize filters in the actual data:

> "*[Texture] makes me feel like I have more visibility into my data set...I'm immediately reading actual samples for like half the time, which I'm never doing in a notebook.*" – P9

In addition to making prior workflows faster and easier, TEXTURE also **enabled participants to explore their data in new ways**. In our baseline sessions, only one participant had analyzed their dataset with embeddings for a projection overview and no participants

leveraged similarity search. However, after we calculated embeddings for their data and uploaded them to TEXTURE, participants found them to be a useful tool for understanding their data.

All 10 of our participants used the embedding projection chart to get an overview of their dataset and explore different subsets of the embedding space. Even participants like P4, who mentioned in the baseline session that embeddings are too hard to use to be valuable, found immediate value from them in the tool when they noticed that the two main types of prompts in their data were clearly separated in the embedding plot (shown in Figure 6.6):

> "*That's great! The fact that it's already there. Literally all of what I did last meeting you just [see here]. This is exactly what I wanted to show*" – P4

Once again, the ability to link filters across parts of the interface helped participants better understand their data. For example, for P7 their core analysis task was to analyze historical books to find counterfeit publications in the late 1600s. Using embeddings, their goal therefore was to find "unexpectedly similar documents". To do this, they colored the projection overview chart by different attributes like the political affiliation of a book and then looked for outliers–i.e., documents with similar embeddings but different political affiliations. When they found an outlying book, they were able to then use TEXTURE to read the text of this book and then compare the instance to others with similar attributes to better understand if it was actually counterfeit.

Whereas no participants used similarity search in their baseline explorations, the majority did while exploring their data in TEXTURE. Similar instance search was more common, used by seven participants whereas only two performed open ended text similarity searches. Participants rated their ability to find similar instances as far easier in TEXTURE than their baseline workflow ($4.6/5$).

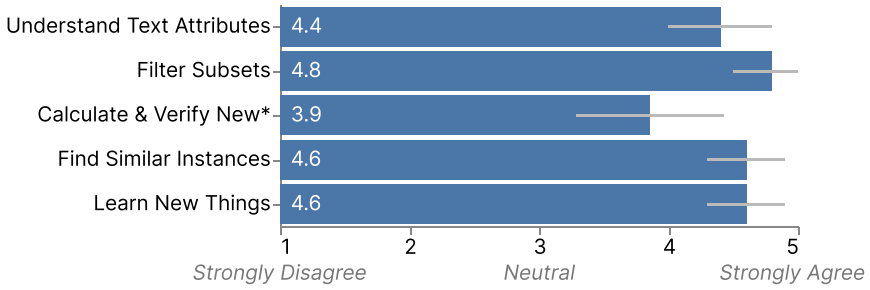

Figure 6.7: Participants rated if Texture made it easier to perform certain actions relative to their baseline. Mean scores shown along with 95% CI. *Calculate and verify new attribute reflects the seven participants who provided ratings for this aspect.

6.7.3 Exploring with TEXTURE Leads to New Insights

TEXTURE provided flexibility for participants to explore data from multiple perspectives—transitioning between top-down and bottom-up modes of exploration. Top down exploration involved looking at attribute summaries then contextualizing these attributes in the

documents; bottom up involved finding an interesting document and then searching for the insight more broadly among documents with similar metadata or through similarity search. These different analysis modes helped participants learn new things about their data they did not uncover in their prior analyses. Overall, participants rated that they learned new things about their data with TEXTURE as $4.6/5$ on average.

For example, P9 and P10 both found major data quality issues they were previously unaware of and did not discover in their baseline analyses. P9 was working with a dataset collected from the social media platform Reddit that contained posts over a 4 year period. They analyzed 15k posts in TEXTURE out of their full 200k corpus. When initially reading through some of the instances to get a sense of the data, they noticed that one particular post seemed to re-occur. Later, while inspecting different outlying clusters in the projection overview they noticed there was a large central cluster and many outlying clusters. Inspecting these outlying clusters more, they noticed they were all the exact same post that comprised almost one third of their data (shown in Figure 6.6). They returned to their notebook, filtered out these duplicate points, then continued to analyze the remaining 10k posts in TEXTURE. They were previously unaware that this post existed so many times in their dataset.

Similarly, P10 identified cases of near repetition in their training prompts by inspecting outlier clusters in the embedding visualization. By inspecting the data instances that corresponded to the different clusters, they noticed these were not exact duplicates but near duplicates where each prompt contained the same starting text and the end was slightly different. These near duplicates comprised around 30% of their overall dataset. P10 emphasized the importance of dataset diversity for their training goals and noted this quality issue as something to address in future training. When asked about how TEXTURE helped them discover this new insight, they remarked:

> "*One huge value here is that AI people hate looking at data. Like I hate looking at data because it's just such a pain. But this makes it actually kind of fun to look at the data and definitely easy. So I'm pretty excited just using this now because I'm like, wow, I can actually see the data that I'm training models on.*"
> – P10

Another theme across participant remarks was the speed with which they were able to explore and pivot between different analysis questions in TEXTURE. For example, P6 was analyzing a corpus of country song lyrics to see if they might contain useful signal around self-perceptions of rural identity. Their analysis was inherently exploratory, where they were trying to better understand the data to see if it would even be useful for this task. In their baseline exploration they had tried different analysis strategies like searching for keywords, looking at a song's sentiment, or basic keyword based topic models. They began their analysis in TEXTURE by inspecting different subsets of these same attributes, which then inspired them to do a similarity search for the term "hometown" (shown in Figure 6.6). From this, they found a particular song whose lyrics captured elements of home and country and then used this song to once again find similar instances. These results contained many promising matches that they continued to explore for the rest of the analysis. Reflecting on their analysis with TEXTURE, they commented:

> "*I think the tool was really good at suggesting new questions, actually. Like, I had some, which I kind of talked to you about [in the first session], but it helped me narrow down the questions.*" – P6

6.7.4 Study Limitations

Our study and findings are subject to potential limitations. Our participants were recruited from a convenience sample and thus might not be representative of the larger population of text analysis practitioners. In particular, our participants skewed towards computer science experts working in NLP.

Participants analyzed the same dataset in the first baseline session and the experiment session, which means additional insights could come simply from additional exposure. However, since each participant analyzed a familiar dataset—previously explored in their own analysis or publication even before the baseline—we expect minimal learning effects from repeated exposure to the data.

6.8 Discussion

6.8.1 The Value of General-Purpose, Configurable Tools

Many text visualization tools target specific analysis questions and methods from a particular domain or task. Such tools are highly valuable when participants are at the phase of analysis where they have decided on their research question and method. However, with TEXTURE we sought to explore a different goal of developing a general exploration tool across tasks as a way to support open-ended exploratory analysis. With this approach, TEXTURE **decouples the derivation of attributes from their exploration**. This allows users to try different techniques—deriving common words, n-grams, topics, LLM-calculated topics, or another arbitrary method—and then explore the results in the same tool. This approach is complementary to domain and task specific systems. Exploratory tools for text like TEXTURE facilitate fast first impressions of a data, verifying new attributes in a dataset, and, importantly, quickly determining if an analysis question is worthwhile before moving on to custom visual and analytical tools for a particular sub-task. Several of the participants in our study commented on the value of this handoff, and how they would like to continue exploring a subset of the data or a question *that they discovered in* TEXTURE.

6.8.2 EDA in the AI Coding Era

In recent years, AI has begun to transform how people write code and interact with data through code [49, 176]. This surfaced in our study as well, where many participants used AI coding assistants to help write the code for their EDA or mentioned how they would like to process their text data further with LLMs.

What role do interactive data exploration systems serve in the era of AI coding? Our results offer some clues. Interactive systems can enable users to **discover interesting questions** about their data, which complements the ability of AI coding assistants to generate

code to answer a specific question. Much of the challenge and value of exploratory analytics from the start has been to develop enough intuition about the data to know what questions are worth asking [157]. Despite using AI coding assistants in their baseline sessions, our study participants still reported how TEXTURE enabled them to quickly explore their data and come up with better analysis questions than they were able to do on their own.

These approaches—interactive systems and AI assistance—need not be in conflict but rather can complement one another. Historically, this has been explored through paradigms like mixed initiative interfaces where a system can both allow user interactions and suggest actions [68]. Exploratory text analysis tools like TEXTURE can enable these sort of interactions at the workflow level, where an AI coding assistant suggests analysis code and then interactive tools help users to quickly inspect the results. This combines the flexibility of writing code with the speed of interactive interfaces for data exploration, an observation noted in prior interactive data programming tools [82].

6.8.3 Towards Rapid and Flexible Attribute Derivation

A key element of data exploration, particularly for text data, is the availability of meaningful attributes to summarize and filter the data. In our study, many participants had previously spent considerable time constructing such attributes with off-the-shelf libraries or custom methods for things like topics or sentiment. Many participants expressed the desire for easier ways to derive task-specific attributes catered to their current analysis. Particularly with the power of general-purpose LLMs for processing text, several participants expressed how it should be straightforward to derive custom attributes by transforming their data with a LLM.

We envision two different ways where interactive systems can better support such an LLM-assisted new attribute derivation workflow: (1) interactively deriving and verifying new attributes, and (2) suggesting interesting questions for derivation. By integrating the ability to derive new attributes directly into systems like TEXTURE, users could quickly derive an attribute, verify the results, and then use this attribute for further analysis. Future research might investigate how to design such interactions to make this feedback loop as fast as possible, and how analysts use them in practice. The second, and perhaps more difficult extension, would be to automatically generate potential questions for derivation, in the spirit of mixed-initiative interactions. Recent systems like Automatic Histograms have begun to move in this direction by automatically grouping text entities into dataset overviews using LLMs [133]. TEXTURE's data model offers a potential starting point for thinking about a broader set of attributes that LLMs can derive that correspond to the entire document or portions of the text. Such methods would continue to increase the speed with which analysts can quickly structure their text data and then use interactive systems to inspect subsets and find interesting insights in their data.

6.9 Conclusion

In conclusion, we present the design, implementation, and evaluation of TEXTURE—a configurable and general-purpose interactive system for exploratory text analytics. TEXTURE

is built on top of a configurable data schema for representing different kinds of descriptive attributes alongside text. We demonstrate the expressivity of this data model to represent 10 different real-world text datasets and associated descriptive attributes. Participants in our study using TEXTURE for analysis were able to more easily understand their data from both top-down and bottom-up analysis paths, using attribute overviews to summarize their dataset then drilling down to find interesting subsets and instances that inspired future analysis directions and questions. Our system and study lay the groundwork for developing expressive and effective interactive systems for exploratory text analytics.

TEXTURE shows how to develop Interactive Data Profiling tools that:

1. Support profiling text datasets by profiling different data representations. This includes both traditional tabular attributes, along with multi-valued list attributes that may correspond to spans of the text, and vector embeddings.
2. Enable fast feedback for exploration through interactive cross filtering across attribute charts that link back to views of the raw documents.
3. Support both bottom-up and top-down exploration through embeddings where users can view a corpus overview, or find an interesting instance and then search for similar instances.

Chapter 7
Discussion & Conclusions

7.1 Summary of Research Contributions

1. **Chapter 3**: Based on an interview study and survey with 149 professional data scientists, we describe five strategies that data scientists use for sharing and reuse of analysis code: personal analysis reuse, personal utility libraries, team shared analysis code, team shared template notebooks, team shared libraries.
2. **Chapter 3**: We report our participants' determinants and obstacles for code reuse, and discuss how this motivates the need for tools to help users more easily reuse code for repetitive analysis tasks like EDA while still maintaining the ability to customize their exploration.
3. **Chapter 4**: We define continuous data profiling as the ability to immediately see interactive visual summaries of a dataset while programming. We then show how our system, AUTOPROFILER, enables continuous data profiling by integrating directly into computational notebooks, summarizing datasets in memory with overview visualizations that *live* update as the data updates, and enables handoffs back to code through exports.
4. **Chapter 4**: We evaluate AUTOPROFILER in a controlled study with 16 participants that demonstrates how continuous profiling helps analysts discover insights in their data and supports their workflow without the need to write extra code. In our study, 91% of user-generated insights come from the tools rather than manual profiling code written by users.
5. **Chapter 4**: We also present a longitudinal case study demonstrating how scientists using AUTOPROFILER in their workflows were able to make *serendipitous* discoveries about their data when the system plotted data they would not have thought to check manually.
6. **Chapter 4**: AUTOPROFILER is released as an open-source system at https://github.com/cmudig/AutoProfiler.
7. **Chapter 5**: We show how to leverage a user's analysis history to create interactive data profiles beyond the current data in memory. Our system, SOLAS, provides an extensible approach for logging user analysis code, weighing, and combining data interactions during analysis.
8. **Chapter 5**: We demonstrate how to use the semantics of the data returned from specific analytical function calls to visualize them with task-specific data profiles. These task-specific visualizations often include data from previous analysis steps. We present the results from an online user survey with 87 participants that shows these encodings are preferred by users.
9. **Chapter 5**: We introduce a method for aggregating over history to model user inter-

est in columns and to update inferred data types based on data transformations.

10. **Chapter 5**: SOLAS is released as an open-source system at https://github.com/cmudig/solas.
11. **Chapter 6**: We describe how to profile text datasets through different data representations. This includes both traditional tabular attributes, along with multi-valued list attributes that may correspond to spans of the text, and vector embeddings. We present a configurable data schema for describing text attributes from arbitrary levels of document granularity and cardinality.
12. **Chapter 6**: We present TEXTURE, an interactive text exploration tool that helps users profile and explore their text data through overview visualizations, filtering, and contextualizing descriptive attributes.
13. **Chapter 6**: We present results from a user study with 10 expert participants that shows how TEXTURE is expressive enough to analyze datasets from these 10 different tasks and helps users learn new things about their data and make new insights.
14. **Chapter 6**: TEXTURE is released as an open-source system at https://github.com/cmudig/Texture.

7.2 Discussion and Future Directions

This thesis has presented a series of systems that develop an approach for building interactive data visualization tools designed to help users quickly explore and make sense of their data while programming. These systems allow users to concentrate on writing code to manipulate and model their data, then use readily interactive tools to quickly interpret the results. We reflect on this approach and review opportunities for future research.

7.2.1 Complementing Data Programming with Interactive Interfaces

Each of the tools in this thesis help users perform data actions in UIs that are either hard or just tedious to accomplish through code. Manually cross filtering data with code is quite hard; plotting the same histogram after each data change is tedious. AUTOPROFILER, SOLAS, and TEXTURE automate some of the common yet tedious steps of data exploration like manually specifying charts, and then make it possible for users to interactively explore their data to ask follow-up questions.

However code is not without its merits—many programming libraries are highly expressive and configurable, making it easier to do custom analyses that would be difficult to fully support in UIs. Where should we draw the line between interactive interfaces that can speed up common analysis tasks vs configurable coding tools? One way to conceptualize this is through the *cliff of expressivity* inherent in interactive UIs. Data interfaces are designed and implemented for specific tasks. For example, the interactive data profiling tools in this thesis display automatic data overviews and support predefined methods of data interaction. However, users might desire alternative ways of visualizing their data or wish to apply a unique filter not currently supported in the UI. In other words, users will inevitably encounter the limits of a tool's expressivity.

Many traditional data analysis platforms tackle this expressivity gap with a monolithic design—cramming every conceivable way to interact with data into a single UI. However,

that all-in-one approach often backfires: the constant context-switching makes them hard to adopt and integrate into varied workflows. In contrast, interactive data profiling tools are designed to address common, repetitive tasks and to fit within a larger ecosystem of programming environments. Integrating such tools into platforms like Jupyter notebooks holds considerable potential for creating a central, interconnected ecosystem. However, this integration can be fragile. Promising research directions, such as projects like anywidget [105], are addressing these challenges by developing interface specifications. For a truly interconnected ecosystem of tools, we need both robust interface specifications and common data specifications, enabling different tools to operate on shared data abstractions and link together seamlessly.

To further address the "cliff of expressivity," interactive tools should facilitate a smooth handoff back to code, even through very simple interactions. For example, AUTOPROFILER facilitates the export of code back into the programming context, and prior research on APIs like Mage formalize ways of navigating this handoff [82]. Regardless of the specific handoff mechanism, we argue that research in interactive tools should not only consider the interactions within the tool itself but also how these tools fit into larger analytical workflows and facilitate seamless transitions to and from other systems, particularly code-based environments.

7.2.2 Hierarchical Data Profiling for Unstructured Data

Future research could explore the extension of interactive data profiling to other data modalities. In particular, unstructured datasets beyond text such as collections of images or videos, stand to benefit significantly from interactive data profiling tools. Building on the design principles of TEXTURE, an image profiling tool could allow users to define different structured representations of their images (e.g. color, objects, lighting) and then filter, sort, and explore the data through these derived attributes.

The concept of hierarchical data, central to TEXTURE, is particularly promising for making sense of rich, unstructured data collections. For instance, with lengthy text documents or PDFs, different types of information might be derived at various levels of hierarchy, such as paragraphs or chapters. These hierarchical summaries could then be used to navigate to interesting subsections of the document. This same concept could be applied to images or videos that possess semantic sub-units relevant for analysis like areas of an image or a segment of a video. Future research can build upon the methods developed in this thesis and in TEXTURE to both automatically derive representations at each of these hierarchical levels (potentially using advanced AI models) and to develop interactive interfaces for navigating this complex information.

7.2.3 Verifying AI-Authored Code with Interactive Interfaces

The interactive data profiling tools presented in this thesis rely on the user's initiative to conceptualize and create meaningful data attributes through code, which are then analyzed in the system. Recently, the advent of AI and agentic code authoring tools is changing how individuals code and work with data, suggesting that an increasing proportion of this code will be authored by AI models [49, 72].

As AI assistants write an ever-larger share of data-processing code, interactive data profiling tools become even more crucial for users to verify the results produced by their AI assistants. For example, interactive data profiling systems can help users make sense of results from analysis and verify that AI authored code does not introduce unintentional errors into datasets. As the time it takes to write code drops, lightweight and integrated interfaces can help users profile their data without slowing down the analysis process.

7.2.4 AI-Powered Data Profiles

Beyond authoring analysis code, powerful AI models offer potential in enhancing data profiling directly through their ability to generate meaningful data overviews. For example, AI models can be used to reason about semantic concepts within a dataset to identify which subsets are particularly interesting or relevant for exploration [133].

More mechanically, AI models are already being employed to transform unstructured data into shorter, structured forms that are more conducive to data profiling [10, 113]. Many insightful data columns are difficult to derive precisely from unstructured datasets using code alone. For instance, deriving the number of users mentioned in a series of paper abstracts might require numerous complex string matches if done through code, but can often be extracted trivially with LLMs. This presents an opportunity to rapidly transform datasets using LLMs and then employ interactive data profiles to make sense of the results, further speeding up the feedback loop between manipulating and exploring data.

7.3 Conclusion

In conclusion, if you read this far—thank you! The overarching goal of this thesis has been to make it easier for people to understand their data while programming. This thesis has introduced the design, implementation, and study of a series of systems for Interactive Data Profiling. AUTOPROFILER, SOLAS, and TEXTURE have been used by participants across numerous lab and deployment studies, scientists, and the open-source community. Looking forward, as the volume and complexity of data continues to grow, the importance of efficient and intuitive data profiling tools will only increase, making such systems a critical piece of effective data exploration and analysis.

References

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 385–395.

[2] E. Alexander, J. Kohlmann, R. Valenza, M. Witmore, and M. Gleicher, "Serendip: Topic model-driven visual exploration of text corpora," in *2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*, IEEE, 2014, pp. 173–182.

[3] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi, "A collaborative filtering approach for recommending olap sessions," *Decis. Support Syst.*, vol. 69, no. C, 2015. DOI: 10.1016/j.dss.2014.11.003.

[4] J. Alsakran, Y. Chen, Y. Zhao, J. Yang, and D. Luo, "Streamit: Dynamic visualization and interactive exploration of text streams," in *2011 IEEE Pacific Visualization Symposium*, 2011, pp. 131–138. DOI: 10.1109/PACIFICVIS.2011.5742382.

[5] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. A. Hearst, "Futzing and moseying: Interviews with professional data analysts on exploration practices," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, pp. 22–31, 2019. DOI: 10.1109/TVCG.2018.2865040.

[6] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19, IEEE Press, 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042.

[7] H. An, A. Narechania, E. Wall, and K. Xu, *Vitality 2: Reviewing academic literature using large language models*, Presented at the NLVIZ Workshop, IEEE VIS 2024, 2024. arXiv: 2408.13450 [cs.HC].

[8] Anaconda Foundation, *The state of data science 2020: Moving from hype toward maturity*, https://www.anaconda.com/state-of-data-science-2020, Accessed 06-2023, 2020.

[9] Apache Arrow, *Pyarrow - apache arrow python bindings*, https://arrow.apache.org/docs/python/index.html, Accessed 06-2023, 2023.

[10] S. Arora *et al.*, "Language models enable simple systems for generating structured views of heterogeneous data lakes," *Proc. VLDB Endow.*, vol. 17, no. 2, pp. 92–105, Oct. 2023. DOI: 10.14778/3626292.3626294.

[11] P. D. Bailis, E. Gan, K. Rong, S. Suri, and S. InfoLab, "Prioritizing attention in fast data: Principles and promise," in *8th Biennial Conference on Innovative Data Systems Research (CIDR 17)*, 2017.

[12] L. Battle and A. Ottley, "What exactly is an insight? a literature review," *2023 IEEE Visualization and Visual Analytics (VIS)*, pp. 91–95, 2023.

[13] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 12–23.

[14] F. Bertrand, *Sweetviz*, https://github.com/fbdesignpro/sweetviz.

[15] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[16] H. Bosch *et al.*, "Scatterblogs2: Real-time monitoring of microblog messages through user-guided filtering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2022–2031, 2013.

[17] M. Bostock. "Introduction to imports." (2018).

[18] Á. A. Cabrera *et al.*, "Zeno: An interactive framework for behavioral evaluation of machine learning," *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023.

[19] R. Chang, C. Ziemkiewicz, T. M. Green, and W. Ribarsky, "Defining insight for visual analytics," *IEEE Computer Graphics and Applications*, vol. 29, no. 2, pp. 14–17, 2009.

[20] E. K. Choe, B. Lee, *et al.*, "Characterizing visualization insights from quantified selfers' personal data presentations," *IEEE computer graphics and applications*, vol. 35, no. 4, pp. 28–37, 2015.

[21] W. Cleveland, *The Elements of Graphing Data*. AT&T Bell Laboratories, 1994.

[22] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. DOI: 10.1145/362384.362685.

[23] C. Collins, F. B. Viegas, and M. Wattenberg, "Parallel tag clouds to explore and analyze faceted text corpora," in *2009 IEEE Symposium on Visual Analytics Science and Technology*, 2009, pp. 91–98. DOI: 10.1109/VAST.2009.5333443.

[24] P. Contributors, *Pytorch*, https://pytorch.org/, Accessed: 2024-12, 2024.

[25] M. Correll, M. Witmore, and M. Gleicher, "Exploring collections of tagged text for literary scholarship," *Computer Graphics Forum*, vol. 30, no. 3, pp. 731–740, 2011. DOI: https://doi.org/10.1111/j.1467-8659.2011.01922.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.01922.x.

[26] T. Dasu and T. Johnson, *Exploratory data mining and data cleaning*. John Wiley & Sons, 2003.

[27] Databricks. (2021).

[28] Databricks. "Notebook workflows." (2021).

[29] T. H. Davenport and D. Patil, "Data scientist: The sexiest job of the 21st century," *Harvard business review*, vol. 90, no. 5, pp. 70–76, 2012.

[30] L. DB, *Lance db*, https://lancedb.com/, Accessed 12-2024, 2024.

[31] R. DeLine and D. Fisher, "Supporting exploratory data analysis with live programming," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 111–119. DOI: 10.1109/VLHCC.2015.7357205.

[32] Ç. Demiralp, P. J. Haas, S. Parthasarathy, and T. Pedapati, "Foresight: Recommending visual insights," *Proc. VLDB Endow.*, vol. 10, no. 12, 2017. DOI: 10.14778/3137765.3137813.

[33] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/V1/N19-1423.

[34] A. Don *et al.*, "Discovering interesting usage patterns in text collections: Integrating text mining with visualization," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, 2007, pp. 213–222.

[35] Y. Elazar *et al.*, "What's in my big data?" *arXiv preprint arXiv:2310.20707*, 2023.

[36] W. Epperson, V. Gorantla, D. Moritz, and A. Perer, "Dead or alive: Continuous data profiling for interactive data science," *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 1, pp. 197–207, 2024. DOI: 10.1109/TVCG.2023.3327367.

[37] W. Epperson, D. Mathur Arpit Moritz, and A. Perer, "Texture: Structured exploration of text datasets," *IEEE Transactions on Visualization and Computer Graphics (under submission)*, 2025.

[38] W. Epperson, A. Y. Wang, R. DeLine, and S. M. Drucker, "Strategies for reuse and sharing among data scientists in software teams," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22, Association for Computing Machinery, 2022, pp. 243–252. DOI: 10.1145/3510457.3513042.

[39] W. Epperson *et al.*, "Leveraging analysis history for improved in situ visualization recommendation," *Computer Graphics Forum*, vol. 41, no. 3, pp. 145–155, 2022. DOI: https://doi.org/10.1111/cgf.14529.

[40] J. A. Evans and P. Aceves, "Machine translation: Mining text for social theory," *Annual Review of Sociology*, vol. 42, no. Volume 42, 2016, pp. 21–50, 2016. DOI: https://doi.org/10.1146/annurev-soc-081715-074206.

[41] E. Fast, B. Chen, and M. S. Bernstein, "Empath: Understanding topic signals in large-scale text," in *Proceedings of the 2016 CHI conference on human factors in computing systems*, 2016, pp. 4647–4657.

[42] C. Felix, S. Franconeri, and E. Bertini, "Taking word clouds apart: An empirical investigation of the design space for keyword summaries," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 657–666, 2018. DOI: 10.1109/TVCG.2017.2746018.

[43] C. Felix, A. V. Pandey, and E. Bertini, "Texttile: An interactive visualization tool for seamless exploratory analysis of structured data and unstructured text," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, pp. 161–170, 2017.

[44] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, "Interactions with big data analytics," *Interactions*, vol. 19, no. 3, pp. 50–59, May 2012. DOI: 10.1145/2168931.2168943.

[45] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The space of developer productivity: There's more to it than you think.," *Queue*, vol. 19, no. 1, pp. 20–48, 2021. DOI: 10.1145/3454122.3454124.

[46] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[47] M. Gentzkow, B. Kelly, and M. Taddy, "Text as data," *Journal of Economic Literature*, vol. 57, no. 3, pp. 535–574, 2019.

[48] Git. "Git." (2021).

[49] Github, *Github copilot - your ai pair programmer*, https://github.com/features/copilot, Accessed 12-2024.

[50] D. Gotz and Z. Wen, "Behavior-driven visualization recommendation," in *Proceedings of the 14th International Conference on Intelligent User Interfaces*, ser. IUI '09, Association for Computing Machinery, 2009. DOI: 10.1145/1502650.1502695.

[51] P. Griggs, C. Demiralp, and S. Rahman, "Towards integrated, interactive, and extensible text data analytics with leam," in *Proceedings of the Second Workshop on Data Science with Human in the Loop: Language Advances*, E. Dragut, Y. Li, L. Popa, and S. Vucetic, Eds., Association for Computational Linguistics, Jun. 2021, pp. 52–58. DOI: 10.18653/v1/2021.dash-1.9.

[52] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-user debugging strategies: A sensemaking perspective," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 19, no. 1, pp. 1–28, 2012.

[53] M. L. Griss, "Software reuse: From library to factory," *IBM systems journal*, vol. 32, no. 4, pp. 548–566, 1993.

[54] A. Groce *et al.*, "You are the only possible oracle: Effective test selection for end users of interactive machine learning systems," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 307–323, 2013.

[55] S. D. Group, *Dataprep*, https://github.com/sfu-db/dataprep.

[56] P. J. Guo, *Software tools to facilitate research programming*. Stanford University, 2012.

[57] S. Gururaja, N. Gandhi, J. Milbauer, and E. Strubell, "Beyond text: Expert needs in document research," *ACL*, 2025.

[58] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, Association for Computing Machinery, 2019, pp. 1–12. DOI: 10.1145/3290605.3300500.

[59] J. Heer, "Agency plus automation: Designing artificial intelligence into interactive systems," *Proceedings of the National Academy of Sciences*, vol. 116, 2019. DOI: 10.1073/pnas.1807184115.

[60] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala, "Graphical histories for visualization: Supporting analysis, communication, and evaluation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, 2008. DOI: 10.1109/TVCG.2008.137.

[61] J. Heer and D. Moritz, "Mosaic: An architecture for scalable & interoperable data views," *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 1, pp. 436–446, 2024. DOI: 10.1109/TVCG.2023.3327189.

[62] F. Heimerl, M. John, Q. Han, S. Koch, and T. Ertl, "Docucompass: Effective exploration of document landscapes," in *2016 IEEE conference on visual analytics science and technology (VAST)*, IEEE, 2016, pp. 11–20.

[63] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *International Conference on Software Reuse*, Springer, 2011, pp. 207–222.

[64] J. M. Hellerstein, "Quantitative data cleaning for large databases," *United Nations Economic Commission for Europe (UNECE)*, 2008.

[65] F. Hermans, B. Jansen, S. Roy, E. Aivaloglou, A. Swidan, and D. Hoepelman, "Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 56–65. DOI: 10.1109/SANER.2016.86.

[66] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *2016 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, IEEE, 2016, pp. 162–170.

[67] F. Hohman, K. Wongsuphasawat, M. B. Kery, and K. Patel, "Understanding and visualizing data iteration in machine learning," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20, Association for Computing Machinery, 2020, pp. 1–13. DOI: 10.1145/3313831.3376177.

[68] E. Horvitz, "Principles of mixed-initiative user interfaces," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 159–166.

[69] Huggingface, *Huggingface datasets*, https://huggingface.co/datasets, Accessed 03-2025, 2024.

[70] J. Hullman and B. Shneiderman, "The purpose of visualization is insight, not pictures: An interview with ben shneiderman," *ACM Interactions*, 2019.

[71] A. Ittoo, A. van den Bosch, *et al.*, "Text analytics in industry: Challenges, desiderata and trends," *Computers in Industry*, vol. 78, pp. 96–107, 2016.

[72] C. E. Jimenez *et al.*, "Swe-bench: Can language models resolve real-world github issues?" In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024.

[73] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, 1972. DOI: 10.1108/eb026526.

[74] M. Kahng *et al.*, "Llm comparator: Interactive analysis of side-by-side evaluation of large language models," *IEEE Transactions on Visualization and Computer Graphics*, 2024.

[75] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer, "Enterprise data analysis and visualization: An interview study," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2917–2926, 2012. DOI: 10.1109/TVCG.2012.219.

[76] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer, "Profiler: Integrated statistical analysis and visualization for data quality assessment," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, ser. AVI '12, Association for Computing Machinery, 2012, pp. 547–554. DOI: 10.1145/2254556.2254659.

[77] H. Kaur, H. Nori, S. Jenkins, R. Caruana, H. Wallach, and J. Wortman Vaughan, "Interpreting interpretability: Understanding data scientists' use of interpretability tools for machine learning," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20, Association for Computing Machinery, 2020. DOI: 10.1145/3313831.3376219.

[78] D. Keim, G. Andrienko, J.-D. Fekete, C. Görg, J. Kohlhammer, and G. Melançon, *Visual analytics: Definition, process, and challenges*. Springer, 2008.

[79] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting exploratory programming by data scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17, Association for Computing Machinery, 2017, pp. 1265–1276. DOI: 10.1145/3025453.3025626.

[80] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2018. DOI: 10.1109/VLHCC.2018.8506576.

[81] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2018. DOI: 10.1145/3173574.3173748.

[82] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel, "Mage: Fluid moves between code and graphical work in computational notebooks," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20, Association for Computing Machinery, 2020, pp. 140–151. DOI: 10.1145/3379337.3415842.

[83] M. H. Kiapour, K. G. Yager, A. C. Berg, and T. L. Berg, "Materials discovery: Fine-grained classification of x-ray scattering images," *IEEE Winter Conference on Applications of Computer Vision*, pp. 933–940, 2014.

[84] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 96–107.

[85] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: State of the art and challenges," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, 2018. DOI: 10.1109/TSE.2017.2754374.

[86] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: State of the art and challenges," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, 2018. DOI: 10.1109/TSE.2017.2754374.

[87] Y. Kim and E. A. Stohr, "Software reuse: Survey and research directions," *Journal of Management Information Systems*, vol. 14, no. 4, pp. 113–147, 1998.

[88] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09, Association for Computing Machinery, 2009, pp. 1569–1578. DOI: 10.1145/1518701.1518942.

[89] A. J. Ko *et al.*, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, Apr. 2011. DOI: 10.1145/1922649.1922658.

[90] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[91] T. Kulesza, M. Burnett, W.-K. Wong, and S. Stumpf, "Principles of explanatory debugging to personalize interactive machine learning," in *Proceedings of the 20th international conference on intelligent user interfaces*, 2015, pp. 126–137.

[92] M. S. Lam, J. Teoh, J. A. Landay, J. Heer, and M. S. Bernstein, "Concept induction: Analyzing unstructured text with high-level concepts using lloom," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, Association for Computing Machinery, 2024. DOI: 10.1145/3613904.3642830.

[93] D. Lange, *Vispubs: A visualization publications repository*, 2024. DOI: 10.31219/osf.io/dg3p2.

[94] B. Lee, M. Czerwinski, G. Robertson, and B. B. Bederson, "Understanding research trends in conferences using paperlens," in *CHI'05 extended abstracts on Human factors in computing systems*, 2005, pp. 1969–1972.

[95] D. J. L. Lee *et al.*, "Lux: Always-on visualization recommendations for exploratory dataframe workflows," *Proc. VLDB Endow.*, vol. 15, pp. 727–738, 2021.

[96] D. J. L. Lee *et al.*, "Lux: Always-on visualization recommendations for exploratory data science," *VLDB*, 2022. DOI: 10.14778/3494124.3494151.

[97] X. Li, Y. Zhang, J. Leung, C. Sun, and J. Zhao, "Edassistant: Supporting exploratory data analysis in computational notebooks with in situ code search and recommendation," *ACM Trans. Interact. Intell. Syst.*, vol. 13, no. 1, 2023. DOI: 10.1145/3545995.

[98] S. Liu *et al.*, "Tiara: Interactive, topic-based visual text summarization and analysis," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 3, no. 2, pp. 1–28, 2012.

[99] S. Liu *et al.*, "Bridging text visualization and mining: A task-driven survey," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 7, pp. 2482–2504, 2018.

[100] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

[101] S. Loria, "Textblob documentation," *Release 0.15*, vol. 2, 2018.

[102] J. Mackinlay, "Automating the design of graphical presentations of relational information," *ACM Trans. Graph.*, vol. 5, no. 2, 1986. DOI: 10.1145/22949.22950.

[103] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, 2007. DOI: 10.1109/TVCG.2007.70594.

[104] J. H. Maloney and R. B. Smith, "Directness and liveness in the morphic user interface construction environment," in *ACM Symposium on User Interface Software and Technology*, 1995.

[105] T. Manz, N. Abdennur, and N. Gehlenborg, "Anywidget: Reusable widgets for interactive analysis and visualization in computational notebooks," *Journal of Open Source Software*, vol. 9, no. 102, p. 6939, Oct. 2024. DOI: 10.21105/joss.06939.

[106] L. McInnes, J. Healy, N. Saul, and L. Grossberger, "Umap: Uniform manifold approximation and projection," *The Journal of Open Source Software*, vol. 3, no. 29, p. 861, 2018.

[107] T. Menzies, "How not to do it: Anti-patterns for data science in software engineering," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, Association for Computing Machinery, 2016, p. 887. DOI: 10.1145/2889160.2891047.

[108] T. Menzies, E. Kocaguneli, F. Peters, B. Turhan, and L. L. Minku, "Data science for software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, IEEE Press, 2013, pp. 1484–1486.

[109] Microsoft. "Kusto query overview." (2021).

[110] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: A review of industrial studies," *Empirical Software Engineering*, vol. 12, pp. 471–516, 2007.

[111] D. Moritz *et al.*, "Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, 2019. DOI: 10.1109/TVCG.2018.2865240.

[112] A. Muralidharan and M. A. Hearst, "Supporting exploratory text analysis in literature study," *Literary and linguistic computing*, vol. 28, no. 2, pp. 283–295, 2012.

[113] A. Narayan, I. Chami, L. Orr, and C. Ré, "Can foundation models wrangle your data?" *Proc. VLDB Endow.*, vol. 16, no. 4, pp. 738–746, Dec. 2022. DOI: 10.14778/3574245.3574258.

[114] A. Narechania, A. Karduni, R. Wesslen, and E. Wall, "Vitality: Promoting serendipitous discovery of academic literature with transformers & visual analytics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, pp. 486–496, 2022. DOI: 10.1109/TVCG.2021.3114820.

[115] Nomic AI, *Nomic atlas*, https://github.com/nomic-ai/nomic, Accessed 12-2024, 2024.

[116] C. North, "Toward measuring visualization insight," *IEEE Computer Graphics and Applications*, vol. 26, no. 3, pp. 6–9, 2006. DOI: 10.1109/mcg.2006.70.

[117] D. Oelke, D. Spretke, A. Stoffel, and D. A. Keim, "Visual readability analysis: How to make your writings easier to read," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 5, pp. 662–674, 2011.

[118] OpenAI, *Openai vector embeddings*, Accessed 03-25, 2025.

[119] Pandas, *Pandas: Python data analysis library*, https://pandas.pydata.org.

[120] Pandas-Profiling, *Pandas-profiling*, https://github.com/pandas-profiling/pandas-profiling.

[121] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[122] K. Pennington, *Bay area craigslist posts, 2000 - 2018*, https://www.katepennington.org/data, Accessed 06-2023.

[123] J. M. Perkel, *Why jupyter is data scientists' computational notebook of choice*, 2018.

[124] D. Petersohn *et al.*, "Towards scalable dataframe systems," *Proc. VLDB Endow.*, vol. 13, no. 12, Jul. 2020. DOI: 10.14778/3407790.3407807.

[125] J. Piazentin Ono, J. Freire, and C. T. Silva, "Interactive data visualization in jupyter notebooks," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 99–106, 2021. DOI: 10.1109/MCSE.2021.3052619.

[126] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019. DOI: 10.1109/MSR.2019.00077.

[127] P. Pirolli and S. Card, "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis," in *Proceedings of International Conference on Intelligence Analysis*, 2005.

[128] Polars, *Polars, lightning-fast dataframe library*, https://www.pola.rs/, Accessed 06-2023.

[129] Project Jupyter, *Project jupyter*, https://jupyter.org/, Accessed: 2024-09-05, 2024.

[130] M. Raasveldt and H. Mühleisen, *Efficient SQL on Pandas with DuckDB — duckdb.org*, https://duckdb.org/2021/05/14/sql-on-pandas.html, Accessed 06-2023, 2021.

[131] D. Raghunandan *et al.*, "Lodestar: Supporting independent learning and rapid experimentation through data-driven analysis recommendations," *Proceedings of the 2021 IEEE Conference on Visualization and Visual Analytics*, 2021.

[132] E. Reif, M. Kahng, and S. Petridis, "Visualizing linguistic diversity of text datasets synthesized by large language models," in *2023 IEEE Visualization and Visual Analytics (VIS)*, 2023, pp. 236–240. DOI: 10.1109/VIS54172.2023.00056.

[133] E. Reif, C. Qian, J. Wexler, and M. Kahng, "Automatic histograms: Leveraging language models for text dataset exploration," *arXiv preprint arXiv:2402.14880*, 2024.

[134] N. Reimers and I. 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.

[135] Rill Data, *Rill developer*, https://github.com/rilldata/rill-developer, Accessed 06-2023.

[136] S. Robertson, Z. J. Wang, D. Moritz, M. B. Kery, and F. Hohman, "Angler: Helping machine translation practitioners prioritize model improvements," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23, Association for Computing Machinery, 2023. DOI: 10.1145/3544548.3580790.

[137] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" In *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 255–265.

[138] A. Rose, *PandasGUI*, https://github.com/adamerose/pandasgui.

[139] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the android market," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, 2012, pp. 113–122.

[140] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2018. DOI: 10.1145/3173574.3173606.

[141] N. Sambasivan, S. Kapania, H. Highfill, D. Akrong, P. Paritosh, and L. M. Aroyo, ""everyone wants to do the model work, not the data work": Data cascades in high-stakes ai," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21, Association for Computing Machinery, 2021. DOI: 10.1145/3411764.3445518.

[142] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)*, 2017. DOI: 10.1109/tvcg.2016.2599030.

[143] H. Seltman, *Experimental design and analysis*. Carnegie Mellon University, 2018.

[144] S. Shankar, R. Garcia, J. M. Hellerstein, and A. G. Parameswaran, "Operationalizing machine learning: An interview study," *ArXiv*, vol. abs/2209.09125, 2022.

[145] B. Sheil, "Datamation®: Power tools for programmers," in *Readings in artificial intelligence and software engineering*, Elsevier, 1986, pp. 573–580.

[146] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996, pp. 336–343. DOI: 10.1109/VL.1996.545307.

[147] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, "Effortless data exploration with zenvisage: An expressive and interactive visual analytics system," *Proc. VLDB Endow.*, vol. 10, no. 4, 2016. DOI: 10.14778/3025111.3025126.

[148] C. Sievert and K. Shirley, "Ldavis: A method for visualizing and interpreting topics," in *Proceedings of the workshop on interactive language learning, visualization, and interfaces*, 2014, pp. 63–70.

[149] H. A. Simon, "Designing organizations for an information-rich world," in *Computers, Communications, and the Public Interest*, M. Greenberger, Ed., The Johns Hopkins Press, 1971, pp. 38–72.

[150] G. Staff, *Octoverse: Ai leads python to top language as the number of global developers surges*, Published 10-29-24, 2024.

[151] J. Stasko, C. Gorg, Z. Liu, and K. Singhal, "Jigsaw: Supporting investigative analysis through interactive visualization," in *2007 IEEE Symposium on Visual Analytics Science and Technology*, IEEE, 2007, pp. 131–138.

[152] S. S. Stevens, "On the theory of scales of measurement," *Science*, vol. 103, no. 2684, 1946. DOI: 10.1126/science.103.2684.677.

[153] H. Strobelt, D. Oelke, B. C. Kwon, T. Schreck, and H. Pfister, "Guidelines for effective usage of text highlighting techniques," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 489–498, 2015.

[154] Sveltejs, *Svelte: Cybernetically enhanced web apps*, https://svelte.dev/, Accessed 06-2023, 2016.

[155] S. Swayamdipta *et al.*, "Dataset cartography: Mapping and diagnosing datasets with training dynamics," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, Nov. 2020, pp. 9275–9293. DOI: 10.18653/v1/2020.emnlp-main.746.

[156] J. W. Tukey, *Exploratory data analysis / John W. Tukey.* (Addison-Wesley series in behavioral science). Addison-Wesley Pub. Co., 1977.

[157] J. W. Tukey, "We need both exploratory and confirmatory," *The American Statistician*, vol. 34, pp. 23–25, 1980. DOI: 10.2307/2682991.

[158] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis, "Seedb: Efficient data-driven visualization recommendations to support visual analytics," *Proc. VLDB Endow.*, vol. 8, no. 13, Sep. 2015. DOI: 10.14778/2831360.2831371.

[159] V. Vasudevan, B. Caine, R. Gontijo Lopes, S. Fridovich-Keil, and R. Roelofs, "When does dough become a bagel? analyzing the remaining mistakes on imagenet," *Advances in Neural Information Processing Systems*, vol. 35, pp. 6720–6734, 2022.

[160] F. B. Viégas and M. Wattenberg, "Timelines tag clouds and the case for vernacular visualization," *interactions*, vol. 15, no. 4, pp. 49–52, 2008.

[161] A. Y. Wang, W. Epperson, R. A. DeLine, and S. M. Drucker, "Diff in the loop: Supporting data comparison in exploratory data analysis," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22, Association for Computing Machinery, 2022. DOI: 10.1145/3491102.3502123.

[162] Z. J. Wang, F. Hohman, and D. H. Chau, "WizMap: Scalable Interactive Visualization for Exploring Large Machine Learning Embeddings," *arXiv 2306.09328*, 2023.

[163] F. Wanner *et al.*, "State-of-the-art report of visual analysis for event detection in text data streams.," in *EuroVis (STARs)*, 2014.

[164] N. Weinman, S. M. Drucker, T. Barik, and R. DeLine, "Fork it: Supporting stateful alternatives in computational notebooks," *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021. DOI: 10.1145/3411764.3445527.

[165] K. Wongsuphasawat, Y. Liu, and J. Heer, "Goals, process, and challenges of exploratory data analysis: An interview study," *ArXiv*, vol. abs/1911.00568, 2019.

[166] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Towards a general-purpose query language for visualization recommendation," in *ACM SIGMOD Human-in-the-Loop Data Analysis (HILDA)*, 2016. DOI: 10.1145/2939502.2939506.

[167] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager: Exploratory analysis via faceted browsing of visualization recommendations," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2016. DOI: 10.1109/TVCG.2015.2467191.

[168] K. Wongsuphasawat *et al.*, "Voyager 2: Augmenting visual analysis with partial view specifications," in *ACM Human Factors in Computing Systems (CHI)*, 2017. DOI: 10.1145/3025453.3025768.

[169] T. Wu, K. Wongsuphasawat, D. Ren, K. Patel, and C. DuBois, "Tempura: Query analysis with structural templates," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–12.

[170] Y. Wu, J. M. Hellerstein, and A. Satyanarayan, "B2: Bridging Code and Interactive Visualization in Computational Notebooks," in *ACM User Interface Software & Technology (UIST)*, 2020. DOI: 10.1145/3379337.3415851.

[171] Y. Wu, J. M. Hellerstein, and A. Satyanarayan, "B2: Bridging code and interactive visualization in computational notebooks," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, 2020. DOI: 10.1145/3379337.3415851.

[172] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo, "Why reinventing the wheels? An empirical study on library reuse and re-implementation," *Empirical Software Engineering*, vol. 25, no. 1, pp. 755–789, Jan. 2020. DOI: 10.1007/s10664-019-09771-0.

[173] K. Xu, A. Ottley, C. Walchshofer, M. Streit, R. Chang, and J. E. Wenskovitch, "Survey on the analysis of user interactions and visualization provenance," *Computer Graphics Forum*, vol. 39, 2020. DOI: 10.1111/cgf.14035.

[174] C. Yan and Y. He, "Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1539–1554. DOI: 10.1145/3318464.3389738.

[175] A. X. Zhang, M. J. Muller, and D. Wang, "How do data science workers collaborate? roles, workflows, and tools," *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, 2020.

[176] A. Ziegler *et al.*, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022, Association for Computing Machinery, 2022, pp. 21–29. DOI: 10.1145/3520312.3534864.