VizSmith: Automated Visualization Synthesis by Mining Data-Science Notebooks

Rohan Bavishi
University of California, Berkeley
rbavishi@cs.berkeley.edu

Shadaj Laddad
University of California, Berkeley
shadaj@cs.berkeley.edu

Hiroaki Yoshida
Fujitsu Research of America
hyoshida@fujitsu.com

Mukul R. Prasad
Fujitsu Research of America
mukul@fujitsu.com

Koushik Sen
University of California, Berkeley
ksen@cs.berkeley.edu

***Abstract*—Visualizations are widely used to communicate findings and make data-driven decisions. Unfortunately creating bespoke and reproducible visualizations requires the use of procedural tools such as matplotlib. These tools present a steep learning curve as their documentation often lacks sufficient usage examples to help beginners get started or accomplish a specific task. Forums such as StackOverflow have long helped developers search for code online and adapt it for their use. However, developers still have to sift through search results and understand the code before adapting it for their use.**

We built a tool called VIZSMITH which enables *code reuse* for visualizations by mining visualization code from Kaggle notebooks and creating a database of 7176 *reusable* Python functions. Given a dataset, columns to visualize and a text query from the user, VIZSMITH searches this database for appropriate functions, runs them and displays the generated visualizations to the user. At the core of VIZSMITH is a novel metamorphic testing based approach to automatically assess the reusability of functions, which improves end-to-end synthesis performance by 10% and cuts the number of execution failures by 50%.

I. Introduction

Visualizations are increasingly being used across various domains, including academic research, journalism, and business intelligence, to communicate insights and enable data-driven decision making [1], [2]. The need for bespoke visualizations and reproducible analytical workflows [3] requires the use of powerful procedural visualization tools such as ggplot and matplotlib [4], [5]. However these tools also have a steep learning curve for novices and domain experts with little programming background. Tool documentation pages function well as a reference but often lack sufficient snippets or examples to help beginners get started.

This has led to a huge surge in popularity of technical Q&A forums such as StackOverflow and social programming platforms like GitHub as they facilitate *code reuse* [6]. Analysts can search for usage-examples or even complete recipes [7], [8] to incorporate directly into their workflow.

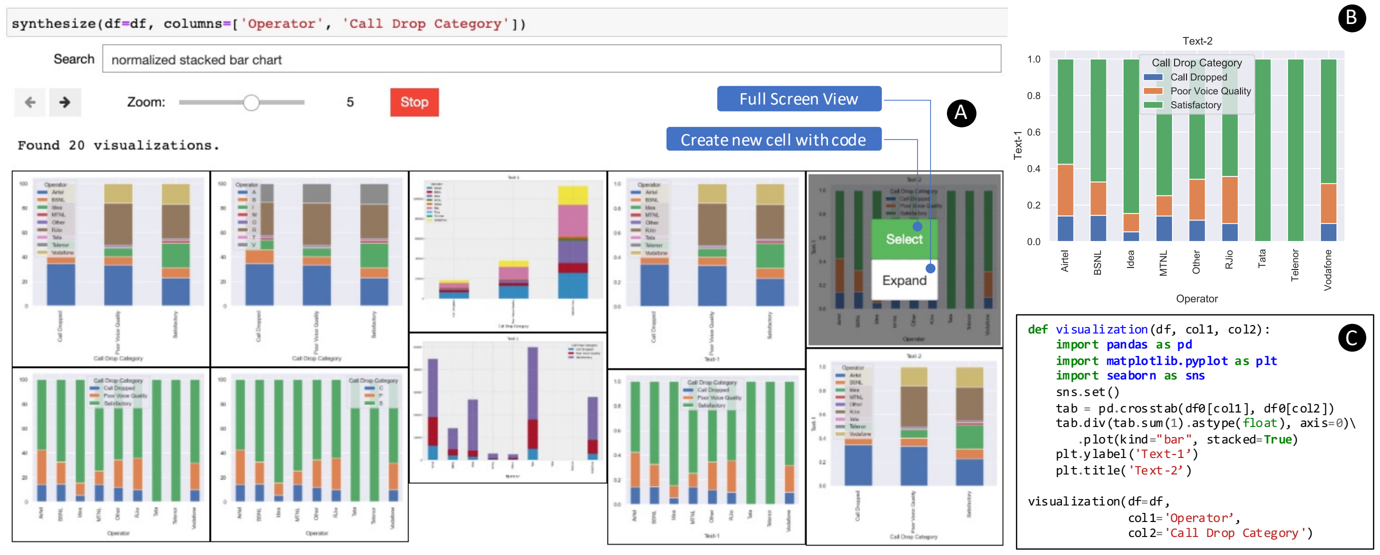
In practice, however, code reuse in software development has largely been sub-optimal [8], [9] due to two main reasons. First, the code results returned by StackOverflow may be incomprehensible to relatively new users, making it difficult for them to modify and reuse that code [8]. Second, there is a proliferation of similar questions on StackOverflow which ends up pushing the burden of selecting the right solution to the end user, who may not be familiar with the specifics of the visualization tools.

Facilitating better code reuse has been a subject of active research [9]–[16]. This includes improving the quality of search results [14], as well adapting the code using additional specifications such as test-cases or type signatures of target methods [9], [11]. None of these are however applicable in the context of visualizations. Wang et al. [17], [18] use a synthesis-powered approach to generate visualization programs in a limited DSL given *partial* or incomplete visualizations. However, this can be insufficient when a helpful partial visualization is difficult to provide, such as when visualizing the correlation matrix of a large table.

In this paper, we present and evaluate an approach for facilitating code reuse in generating visualizations. We leverage the fact that machine learning platforms such as Kaggle [19] host scores of executable data science notebooks that also include the raw dataset. We developed a tool VIZSMITH that analyzes these notebooks and mines a knowledge base of *visualization functions*, which are Python functions that take an input table and the set of columns to visualize as input and produce a visualization as output. VIZSMITH provides a frontend where users can provide a dataframe and the columns to visualize along with a text query. VIZSMITH finds, ranks, and executes the functions best matching the query, and displays the synthesized *bespoke* visualizations.

At the heart of VIZSMITH lies a novel analysis for determining the quality or *reusability* of a mined visualization function. The analysis allows it to discard low-quality code upfront which greatly helps in improving both quality and speed of synthesis. To the best of our knowledge, we are the first to provide a precise conceptual definition of *reusability* in the context of visualization code. We also develop a novel approach based on metamorphic testing that approximates this definition, for automatically evaluating reusability of any arbitrary visualization function. In summary, our contributions within VIZSMITH are as follows:

1) A framework for mining visualization functions from Kaggle that yields a knowledge base of 7126 *reusable* functions mined across 1280 notebooks and 10 competitions.



```
synthesize(df=df, columns=['Operator', 'Call Drop Category'])
```

```
def visualization(df, col1, col2):
    import pandas as pd
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set()
    tab = pd.crosstab(df0[col1], df0[col2])
    tab.div(tab.sum(1).astype(float), axis=0)\
       .plot(kind="bar", stacked=True)
    plt.ylabel('Text-1')
    plt.title('Text-2')

visualization(df=df,
              col1='Operator',
              col2='Call Drop Category')
```

Fig. 1: VIZSMITH's Jupyter notebook frontend. VIZSMITH is provided with a table as a Pandas dataframe along with columns to visualize as input. It has a search bar to input text queries. (A) shows how Alice uses VIZSMITH to search for normalized stacked bar charts for her call quality dataset. (B) and (C) show the visualization selected by Alice and its code respectively.

2) A conceptual definition of reusability in the context of visualizations along with a novel decision procedure based on metamorphic testing that achieves 73% precision and 71% recall with respect to a ground truth obtained via manual inspection.
3) A synthesis engine that takes as user input a dataframe and the columns to visualize. In a cross-project experiment, the target visualization is contained in the top-10 results returned by VIZSMITH for 56% of our benchmarks.
4) A publicly available front-end and demo at https://github.com/rbavishi/vizsmith-demo.

II. MOTIVATING EXAMPLE

Alice is a researcher working on a project on analyzing the voice call quality dataset released by the Indian government [20] containing customer ratings. As part of her project, Alice needs to build a visual dashboard that updates every time new data comes in. She has heard about rich data transformation and visualization libraries in Python such as `pandas` and `matplotlib` and decides to use them for this purpose.

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

Alice promptly writes code to load the dataset into a `pandas` dataframe. Unsure about how to create a stacked bar chart, she visits the `matplotlib` gallery entry for this chart [21] only to find it insufficient for her needs. She is also uncertain about exactly how to *transform* her dataframe in order to create the bar chart. She turns to StackOverflow for help and browses the results for the query "`matplotlib pandas normalized stacked bar chart`".

The top result [22] contains a visualization close to what Alice needs, but she has trouble understanding the code, let alone adapting it for her data. This experience is in line with the findings of previous work [8].

Figure 1 demonstrates how Alice uses VIZSMITH to find the visualization of her choice along with code to produce it. First, Alice fires up the frontend of VIZSMITH, which is implemented as a Jupyter notebook [3] widget. Alice provides VIZSMITH with her dataframe as well as the columns she wants to visualize. VIZSMITH then presents a search bar where Alice provides the same query as before.

VIZSMITH then consults its knowledge base of *visualization functions* that it has mined from the machine learning notebooks written by data scientists on Kaggle. VIZSMITH utilizes dynamic program analysis and metamorphic testing to construct these functions. These visualization functions are regular Python functions that take a dataframe as an argument along with column arguments and produce a visualization after performing any necessary dataframe transformations. VIZSMITH indexes these functions using the names of the API functions and their keyword arguments, along with the natural language comments found in the Kaggle notebooks. Given Alice's keywords, VIZSMITH finds the best matching functions, runs them and presents the resulting visualizations in a gallery view as shown in Figure 1. VIZSMITH allows Alice to expand a particular visualization to a full-screen view as well as study the code for the visualization.

Alice finds her desired visualization in this list right away, shown in (B) in Figure 1. VIZSMITH also produces many similar visualizations with small styling variations. The code for the visualization is shown in (C) and illustrates the inherent complexity of the task as it needs a combination of three `pandas` functions, namely `crosstab`, `div` and `sum` followed

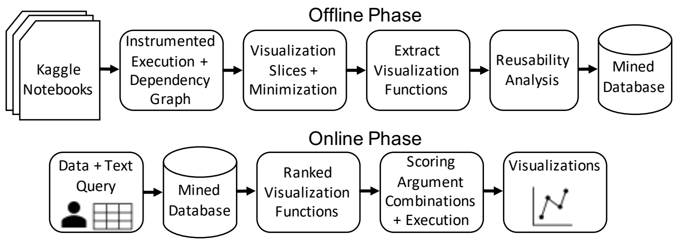

Offline Phase
Kaggle Notebooks
Instrumented Execution + Dependency Graph
Visualization Slices + Minimization
Extract Visualization Functions
Reusability Analysis
Mined Database
Online Phase
Data + Text Query
Mined Database
Ranked Visualization Functions
Scoring Argument Combinations + Execution
Visualizations

Fig. 2: Overview of VIZSMITH.

by the call to `plot`. Alice copies the code into her workflow, and adjusts the title and y-axis labels.

Thus, VIZSMITH enables *better code reuse* by eliminating the burden of understanding and adapting code found online.

III. Overview of VizSmith

Figure 2 presents a high-level overview of VIZSMITH. In the offline phase, VIZSMITH collects and mines visualization functions from Python notebooks hosted on the machine learning platform Kaggle [19] (Section IV). VIZSMITH also analyzes the functions using a novel metamorphic testing scheme (Section V) to discard functions ill-suited for synthesis. In the online phase, VIZSMITH receives from the user, a dataframe as well as the set of columns in the dataframe that participate in the desired visualization along with a search query. VIZSMITH first uses the query to collect a ranked list of functions to explore (Section VI-A). Then it finds appropriate arguments to the parameters of each function, and executes them all. Finally, VIZSMITH collects and displays the generated visualizations in a Jupyter notebook user interface.

IV. Mining

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

A. Collecting and Replaying Notebooks

We sort the list of competitions on Kaggle by the number of teams participating in the competition. From the top-50 such competitions, we pick those where the dataset corresponds to a *single* csv/tsv file. We additionally include the `titanic` and `house-prices` competitions as they are the most well-known classification and regression tasks on Kaggle respectively, resulting in a total of 10 competitions (Table II).

Within these competitions, we only select kernels that have an associated Docker image ID which can be downloaded from Kaggle's GCR repository[1]. To conserve resources, we ignore GPU-based kernels and impose a timeout of 10 minutes on each kernel run. For competitions with large datasets (>50k rows), we take a sample of the dataset in order to reduce the execution time.

[1]https://gcr.io/kaggle-images/python

B. Instrumentation and Execution

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

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

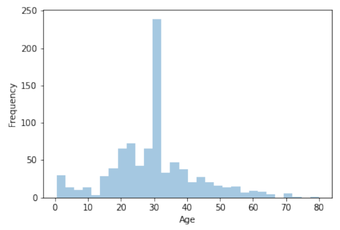
C. Visualization Objects and Visualization Slices

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

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

Definition 1 (Visualization Slice)**.** We define the visualization slice of a program P with respect to a visualization object ν, denoted as *VizSlice*(P, ν), as a program P' that can be obtained by removing statements from P such that, when executed, P' produces the same visualization ν and *only* ν.

Thus, a visualization slice contains all the statements in a program necessary for recreating a particular visualization. Figure 3 contains a slice of the linked Kaggle notebook for



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

Fig. 3: Example of a visualization and a corresponding slice extracted from Kaggle.

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

Fig. 4: Minimized version of visualization slice in Figure 3.

the shown visualization. Furthermore, the slice should only produce a single visualization.

We use standard dynamic program slicing [23] to obtain a visualization slice. Specifically, we remove all statements in P that are *not reachable* via a backward-traversal of the dependency graph of P starting from any of the statements in in the set $\mathit{VizStmts}(P, \nu)$ defined below:

Definition 2 ($\mathit{VizStmts}(P, \nu)$)**.** $\mathit{VizStmts}(P, \nu)$ is the set of all statements in the program P that *directly* create/modify the visualization object ν.

In Figure 3, the statements in lines 5-6 correspond to the set returned by *VizStmts* for the program corresponding to the parent notebook and the visualization object being the actual plot at the top of Figure 3. The first creates the distribution plot, while the second sets the labels of the axes. The remaining statements in Figure 3 modify the style, load the dataframe and modify it before visualization and are hence included in the slice.

D. Minimizing Visualization Slices

Recall that our dependency graph construction is not precise as we use coarse specifications for third-party libraries. As a result, the visualization slice obtained via dynamic program slicing may still contain irrelevant statements whose removal will not affect the visualization. Consider the slice in Figure 3. The call to `set_style` in line 3 is unnecessary as the style is `"white"` by default. It is included in the slice because it writes to an internal styling dictionary which is then read in the call to `distplot` thereby establishing a dependency. Taking the subset of columns in line 6 is also unnecessary as `distplot` only receives the target column anyway. We can remove both these operations to yield a simpler, *minimized* visualization slice, as shown in Figure 4.

How do we obtain the minimized visualization slice in Figure 4 from the slice in Figure 3? Note that it is not enough to simply remove or delete code as one might do if they were using delta-debugging [24]; removing lines 3 and 6 in Figure 3 would lead to an undefined variable error for `df`. Essentially, we need transformations that go beyond code removal.

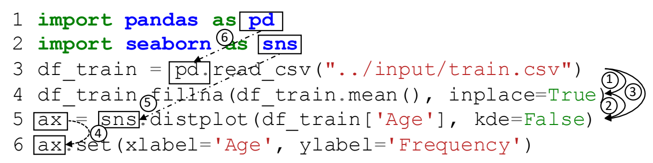
We instantiate the generalized syntax-guided program reduction framework developed in PERSES [25] to enable this minimization. In particular, we use standard statement-level delta-debugging to remove top-level statements whose removal does not change the generated visualization. Additionally, we use a transformation where we replace a usage of a variable holding a dataframe with the usage of a previously defined variable, also holding a dataframe. We keep alternating between these two transformations until the slice cannot be minimized further without altering the visualization. Alternation helps here because one transformation may introduce minimization opportunities for another. Algorithm 1 describes this procedure.

Algorithm 1 Minimization Algorithm Pseudocode

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

We walk through how the algorithm minimizes the slice in Figure 3. In the first iteration, delta-debugging (line 5) would remove the call to `set_style` in line 3, Figure 3. Then we iterate over variants returned by DFVARREPLACE. DFVARREPLACE replaces a use of a variable holding a dataframe by a use of another previously defined variable holding a different dataframe. If there are many possibilities, *DfVarReplace* explores variants in the descending order of the *gap* between the original and replacing definitions of the variables, measured in the number of statements. The variant where `df` is replaced with `df_train` is retained. Then line 6 in Figure 3 gets removed in the second iteration, and the algorithm exits after the third iteration as no further minimization occurred, successfully returning the desired slice in Figure 4.

The reasons behind selecting these two transformations are two-fold. First, data-science code has minimal control flow. Hence, focusing on top-level statements is sufficient. Secondly, data-transformation logic almost always involves applying API functions on variables holding the data (dataframes). Since



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

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

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

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

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

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

Fig. 6: Visualization functions extracted from slice in Figure 3.

visualization slices can be slow to execute as they use heavy-weight libraries, our restricted set of transformations strike a balance between scalability and quality of minimization.

E. Extracting Visualization Functions

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

A visualization function is formally defined as follows:

Definition 3 (Visualization Functions)**.** A visualization function f is a Python function with a single dataframe parameter *df* and m column parameters $col_1, \dots, col_m$ that produces a visualization.

Note that while the above definition restricts a visualization function to a single dataframe parameter, our technique has no such inherent restriction. We adopt this definition to simplify the discussion and the notation used throughout the paper.

At a high level, visualization functions can be extracted from a visualization slice by converting variables holding references to dataframes into parameters and abstracting concrete references to columns into column parameters. The body of the function contains only the statements from the slice required to reproduce its visualization given the new dataframe argument. Figure 6 shows two visualization functions from the visualization slice in Figure 3. Each of them has a single column parameter `col1`. Both produce a visualization containing the distribution plot of the supplied column, with the function in Figure 6a performing an extra imputation step to replace missing values by the mean of their respective columns. We call the slice P_ν from which a visualization function f is obtained as the *parent slice* of f.

Algorithm 2 formalizes the idea. Given a visualization slice P_ν producing visualization ν and its dependency graph G, for every program point b between the top-level statements of the slice (line 4) , and every variable *var* holding a reference to a dataframe object val_{df} that is in scope at b (line 6), we extract a visualization function as follows. We set the body of the function to be a subset of the statements in P_ν, with the variable *var* renamed to *df* (the dataframe parameter). This subset is the smallest such that if the function is executed with the initial value of *df* as val_{df} in the exact same state it was at program point b in the slice, the resulting visualization is the same as ν. This subset is obtained using backward slicing (lines 7-10), but on a subgraph G_r of G. G_r has the same set of nodes as G, but does not contain any dependency edges in G that originate before the boundary and that arise because of the use of the variable *var* or the dataframe val_{df}. This helps us pick only the statements necessary to reproduce visualization ν if *var* is already assigned to val_{df} to begin with.

Algorithm 2 Extracting Visualization Functions

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

```
1:  function ExtractVizFunctions($P_\nu$, $\nu$, $G$)
2:    $\langle s_1, \dots, s_k \rangle \leftarrow$ top-level statements in $P_\nu$
3:    funcs $\leftarrow \emptyset$
4:    for each program point $b \in [1, k]$ do
5:      $S_b \leftarrow \{s_1, \dots, s_b\}$
6:      for each $(var, df_{var}) \in$ GetVarDFs$(P_\nu, b)$ do
7:        $E_r \leftarrow \{e | e \in$ Edges$(G) \wedge$ src$(e) \in S_b$
                  $\wedge$ IsDataFrameEdge$(e, var, df_{var})\}$
8:        $G_r \leftarrow$ induced subgraph of $G$ by removing edges in $E_r$
9:        root $\leftarrow$ VizStmts$(P_\nu, \nu)$
10:       body $\leftarrow \{s_i \mid s_i \in \{s_1, \dots, s_k\}$
                  $\wedge$ Reachable$(s_i, G_r$, root$)\}$
11:       $S_{forbid} \leftarrow \{s | s \in S_b \wedge var$ is used in s$\}$
12:       if $S_{forbid} \cap body = \emptyset$ then
13:         $f$.df_param $\leftarrow$ df
14:         $f$.body $\leftarrow$ RenameVar(body, $var$, df)
15:         $f$.col_params, $f$.body $\leftarrow$ InferColParams$(f, df_{var})$
16:         if Verify$(f)$ then
17:           funcs $\leftarrow$ funcs $\cup \{f\}$
18:   return funcs
```

For example, suppose b=3 and *var*=`df_train` and the slice under consideration is the one in Figure 3. The graph G_r would *not* contain the edges ① and ③ in Figure 5 as they originate right after the statement at line 3 (before b), and arise due to the use of the dataframe variable `df_train`. The edge ② is *included* as it originates *after* b.

Lines 11-12 confirm that the selected statements which appear before the selected program point b do not involve

```python
def visualization(df):
  import seaborn as sns
  sns.heatmap(df.corr())
```

Fig. 7: A visualization function taking no arguments.

the use of variable *var*. This prevents any dependency on a possibly *stale* version of val_{df}. Then, we infer column parameters by simply replacing all string constants that correspond to a column name in val_{df} with parameter variables (line 15). In Figure 3, this corresponds to the string `"Age"` in lines 5 and 6. We also rewrite attribute based column-accesses of dataframes, such as `df.Column` as `df["Column"]` prior to applying this procedure. We denote the mapping from these column parameters to the string constants as $\textsc{OrigCols}(f)$. We refer to the selection of val_{df} as $\textsc{OrigDF}(f)$.

Finally, in line 16, we verify if running the visualization function with val_{df} i.e. $\textsc{OrigDFs}(f)$ and $\textsc{OrigCols}(f)$ reproduces the visualization from the parent visualization slice. Figure 6 contains the two visualization functions extracted from the visualization slice in Figure 3. Observe that no choices for a dataframe variable would be available if we pick the program point b as either 1 or 2.

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

F. Participating Columns vs. Column Parameters

Visualization functions have dataframe and column parameters. It is important to note that column parameters do not necessarily correspond to the exact subset of columns that actually *participate* in the visualization. For example, the function in Figure 7 accepts no column arguments, but produces a correlation heatmap of all the numeric columns in the passed dataframe. We call such columns *implicitly* participating columns. Consequently, we call a column as *explicitly* participating if it is passed as a column argument.

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

We denote the set of columns visualized (explicit or implicit) by f for the dataframe $\textsc{OrigDF}(f)$ as $\textsc{OrigParticipatingCols}(f)$. This notion of participation is at the heart of the reusability analysis as well as visualization synthesis as we shall see next.

V. Analysis of Mined Visualization Functions

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

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

Fig. 8: A visualization function with hard-coded values.

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

Fig. 9: A visualization function using a specialized predicate.

Consider the visualization function in Figure 8. Note that the data-values passed to `ax.pie` in line 12 are hard-coded in the function. That is, regardless of the dataframe and categorical column passed to the function, the produced visualization will be exactly the same. The produced visualization is thus not meaningful. If this function is used in a visualization synthesis setting, its resulting visualization would most likely make no sense to the user, and could undermine trust in the system. Thus we deem this function to be *non-reusable*. This also illustrates why a successful execution of a function does not necessarily entail a meaningful visualization.

In contrast, we consider the function in Figure 7 as "good" or *reusable*. It will correctly produce a correlation heatmap for the class of dataframes that have at least one numeric column. This class clearly includes a wide variety of dataframes and hence we consider this function *reusable*.

Figure 9 presents a much more subtle scenario. The function plots a histogram of the values in `col2`, but only considers the rows where the value corresponding to `col1` is `1`. This filtering criteria is quite arbitrary and only meaningful for dataframes that contain a `1`. We thus deem this function non-reusable..

A. Defining Reusability

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

Definition 4 (Reusable Visualization Function)**.** We consider a visualization function f *reusable* if there exists a set S_{df} of dataframes such that:

1) f produces a meaningful visualization for every dataframe *df* in S_{df}, given an appropriate assignment of *df*'s columns to f's parameters. A meaningful visualization is non-empty and represents all the information in *df* or a filtered view of *df* where the filtering criterion is independent of the concrete data values in *df*.
2) S_{df} can be characterized using high-level properties of a dataframe and its columns including types of columns and types of data values, but excluding properties relying on arbitrary constants or values in the data.

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

Ideally, we would like to be able to *automatically* classify our mined visualization functions as reusable and non-reusable and discard the non-reusable functions. However it is hard to automatically check if a visualization function is reusable according to Definition 4 as we do not have access to S_{df}. Essentially, we are faced with the problem of a missing *test-oracle* [26]. We present a novel approach of using metamorphic-testing [27] to alleviate this issue.

B. Metamorphic Testing for Checking Reusability

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

Definition 5 (MR for Approximating Reusability)**.** Visualizations produced by f on mutated copies of its original dataframe i.e. ORIGDF(f) must all be different from each other as well as the original visualization of f.

These mutated dataframes are produced using column-level *type-aware* mutation operators. Definition 5 along with these mutation operators approximates the concept of reusability in Definition 4 in two ways. First, these operators only modify one column, and take the column type (categorical, quantitative etc.) into account. This helps increase the likelihood of staying within the class of dataframes f is appropriate for. It also ensures that this class is characterizable using simple properties like column types. Second, the mutations applied are large enough to warrant a change in the visualization if f truly produces a visualization that represents all the information in the dataframe or a meaningful subset of it. This helps catch cases like Figure 8 and Figure 9

Algorithm 3 formalizes our metamorphic testing strategy. For every *visualized* column $c \in$ ORIGPARTICIPATINGCOLS(f), we check if there exists a mutation operator for which the metamorphic relation is satisfied for the mutated dataframes it generates. Every mutation operator has a guard that must be true for it to be applicable (line 7).

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

Algorithm 3 Checking Reusability using Metamorphic Testing

```
 1: function IsReusable(f)
 2:     $df_{orig} \leftarrow$ OrigDF($f$); $\nu_{orig} \leftarrow$ OrigViz($f$)
 3:     for each $c \in$ OrigParticipatingCols($f$) do
 4:         success ← false
 5:         for each mutation operator $m$ for ColType($c, df_{orig}$) do
 6:             s ← initialize $m$
 7:             if Guard($m, df_{orig}, c, s$) then
 8:                 $df_1, \ldots, df_k \leftarrow m(df_{orig}, c, s)$
 9:                 $\nu_1, \ldots, \nu_k \leftarrow$ viz produced by $f$ on $df_1, \ldots, df_k$
10:                 if $(\forall i.\ \nu_i \neq \nu_{orig}) \wedge (\forall i \neq j.\ \nu_i \neq \nu_j)$ then
11:                     success ← true
12:                     break
13:         if success is false then
14:             return false
15:     return true
```

a) Categorical Columns: The visualization may be a function of either (1) the individual category labels in the column, or (2) the count distribution of categories or (3) whether a value is a NaN (missing value). Note that the visualization may represent a *function* of these properties, which may not necessarily be identity. The first operator in Table I selects a fixed subset of values and replaces them with one or more unseen categories. Thus, if a function relies on hard-coded values or too arbitrary a filtering process, the resulting visualizations should be the same and thus fail the check. The second operator enables the check of whether the visualization is sensitive to whether values are NaNs or not, rather than their concrete values themselves. It also has a guard which checks whether substituting the same missing values with different categories yields the same result as the original. This ensures that cases like Figure 9 do not pass the check.

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

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

TABLE I: Column Mutation Operators

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

VI. Visualization Synthesis

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

A. Search

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

Given a search query, we rank documents according to their similarity with the search query using BM25 [28]. To obtain a ranked list of visualization functions, we simply map the documents back to their respective visualization functions.

B. Generating Visualizations

VizSmith adapts the ranked visualization functions to the user-provided dataframe using the Instantiate function in Algorithm 4. It takes as input the mined visualization function f, the user supplied dataframe *df* and the columns that must participate in the visualization *vcols*.

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

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

We consider a column eligible to participate implicitly (IsImplicitCand), if there exists a column in the original set

Algorithm 4 Instantiating Visualization Functions

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

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

of implicitly participating columns of f which has the same column-type and data-types. The rationale is that if a column participates implicitly, the criteria determining its participating is most often a function of the column and data types.

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

VII. Evaluation

We focus on three main research questions (RQs) to evaluate VizSmith. In RQ1, we analyze the diversity of the mined visualization functions. Specifically, we explore the distributions over the size of the functions, the APIs explored, and whether a function performs data pre-processing. RQ2 evaluates our metamorphic testing approach to computing reusability against a ground truth established via a manual study. Finally, in RQ3, we evaluate end-to-end synthesis performance of VizSmith.

A. RQ1: How diverse is the collective functionality of all visualization functions?

As it is infeasible to manually examine each function and classify its functionality, we approximate it as the set of API functions used in the body of the visualization function. Note that we only consider functions classified as reusable

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

competition	# viz. funcs (passed quality assurance)
LANL-Earthquake-Prediction	90 (86)
covid19-global-forecasting-week-1	397 (212)
house-prices	3832 (2772)
mercari-price-suggestion-challenge	120 (67)
mercedes-benz-greener-manufacturing	95 (64)
otto-group-product-classification	68 (68)
santander-customer-satisfaction	39 (23)
santander-value-prediction-challenge	64 (47)
titanic	4745 (3604)
tmdb-box-office-prediction	290 (233)
total	9740 (7176)

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

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

by VIZSMITH. We find that all mined functions collectively exercise a total of 289 API functions across 12 third-party libraries. We further bucket each API function manually into four categories using simple criteria, namely (a) plotting if it draws a visualization, (b) transformation if it involves reshaping or filtering operations such as transpose, groupby and dropping null rows, (c) computation if it involves mathematical operations such as correlation and skew and (d) styling if it only modifies the look of a visualization or the text inside it.

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

B. RQ2: How accurate is our metamorphic testing approach?

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

We sampled 50 reusable and 50 non-reusable visualization functions as judged by our metamorphic testing approach. We then designed an interface that displays these 100 functions one-by-one in a random order. Three of the authors labelled each function as reusable or non-reusable as per Definition 4. We computed the ground-truth label via majority vote. In particular, the authors try to assess the intent of the visualization, the class of dataframes where a similar visualization would be meaningful and whether the implementation would be able to produce that visualization without any modifications.

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

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

We find the accuracy of the metamorphic approach to be 71%, with a precision of 73% and recall of 71%. There were 7 false positives (ground-truth non-reusable, classified reusable) and 22 false negatives. We categorized these cases in Table IV. The category column summarizes the reason for the misclassification of the metamorphic testing approach. Examples of these categories are shown in Figure 10.

The 7 false positives occur because our mutation operators are only applied on one column at a time (category A), or the code performs overly specific transforms that are not triggered by mutations (category B), and hence pass metamorphic testing check. The majority of the false negatives occur because of over-aggressive mutation (category D). The example in Figure 10 uses a log function that throws an error when our mutation introduces negative values. In 4 cases, the design choice of using bucketing or changing the axis limits led to the same visualizations being produced despite the mutations (category C). Finally, there were 4 cases in Category E where the filtering was not arbitrary (all positive values), but was judged to be the case by our approach. All categories except E can be handled by a more sophisticated mutation scheme or finer-grained operators. Category E would require a pre-defined notion of what is an adequately general filtering criterion.

C. RQ3: Effectiveness of Synthesis Approach

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

Ⓐ

```
def visualization(df, col1, col2):
  import matplotlib.pyplot as plt
  train_df = df.drop(
   df[
      (df[col1]>4e3) & (df[col2]<3e5)
     ].index
  )
  plt.scatter(train_df[col1],
              train_df[col2])
```

Ⓑ

```
def visualization(df, col1):
  import seaborn as sns
  df[col1]=df[col1].fillna("S")
  df[col1]=df[col1].map({"S":0,"C":1,"Q":2})
  sns.heatmap(df.corr(), annot=True)
```

Ⓒ

```
def visualization(df, col1):
  import matplotlib.pyplot as plt
  df[col1].hist(bins=5, grid=False)
  plt.xlabel(col1)
```

Ⓓ

```
def visualization(df, col1, col2):
  import numpy as np
  import matplotlib.pyplot as plt
  df[col1]=np.log(df[col1])
  plt.scatter(df[col1],df[col2])
```

Ⓔ

```
def visualization(df, col1):
  import seaborn as sns
  ms = df[df[col1] > 0]
  sns.barplot(ms.index, ms[col1])
```

Fig. 10: Examples of each category in Table IV.

consider reusable functions as benchmarks. This yields 3284 benchmarks in total.

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

We find that both VIZSMITH and VIZSMITH$_{\text{ALL}}$ have a top-10 accuracy of 5%. That is, both have an *exactly* matching visualization in the top-10 for only 5% of the cases. There are two possible reasons for this low performance: (a) the quality of the natural language query is poor and (b) styling variations such as color schemes, rotation of tick labels and legend positions will fail the matching visualization test. In a separate manual study of a sample of 100 visualization functions with associated natural language comments, we found only 17% to actually describe the kind of plot and the columns being visualized. Thus (a) is a distinct possibility. To mitigate the effects of (b), we sample 50 benchmarks and examine the results of both the tools manually. In particular, we ignore stylistic variations such as color schemes, rotations of tick labels, legend positions etc. while comparing the visualizations with the ground truth. The top-10 accuracy in this case is 56% and 46% for VIZSMITH and VIZSMITH$_{\text{ALL}}$ respectively. Although the numbers are close, the difference lies in the number of functions explored. VIZSMITH explores 50% less visualization functions than VIZSMITH$_{\text{ALL}}$ while still getting slightly better accuracy as it only searches over reusable functions which we hypothesized to be more useful during synthesis than their non-reusable counterparts. Hence, we demonstrate the utility of reusability analysis to improve end-to-end synthesis performance.

VIII. LIMITATIONS AND THREATS TO VALIDITY

A. Real-World Usage

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

B. Code Licensing and Security

VIZSMITH's database is populated using code written by data scientists and machine learning practitioners that is publicly available on the Internet. As such, code snippets returned by VIZSMITH may not be appropriate for use in certain contexts due to the license of the parent notebook containing the code snippet. This can be mitigated by passing an appropriately vetted corpus to VIZSMITH. Security may also be a concern since VIZSMITH executes every function in its database as part of its metamorphic testing phase. We could mitigate this by adding extra checks to filter out functions with excessive resource consumption, unauthorized file system access, or network requests.

C. Construct Validity

All three research questions involve manual analysis and thus have a subjective component. For RQ1, we classified the functions manually. To reduce the effect of subjectivity, we provided simple and easily reproducible criteria for arriving at this classification. For RQ3, we analyzed the generated visualizations manually because it is hard to automatically identify stylistic variations in a reliable manner. We precisely listed down the classes of stylistic variations that we ignore while comparing two visualizations. Judging reusability as per Def. 4 involves manual inspection of the code, the data, and the visualization. Thus, RQ2 has a higher risk of imprecision than RQ1 and RQ3. We mitigated this by having three reviewers independently judge reusability and taking the majority vote. We also assessed the misclassifications qualitatively, and came up with general characterizations of the failure cases.

IX. RELATED WORK

We compare VIZSMITH against existing visualization authoring systems along the dimensions of intended usage, the use of code templates, the kind of specifications used and the aspect of learning from data. We also compare and contrast applications of code mining and reuse in other domains.

A. Visualization Authoring Systems

1) Exploratory Data Analysis: To facilitate data exploration, systems such as Voyager [29] and Draco [30] accept *partial* visual specifications containing the columns to visualize as well as wildcards to generate a collection of visualizations capturing different views of the data in a target grammar such as Vega-Lite [31]. These visualizations are filtered and ranked based on either manually designed heuristics [29], [32] or using constraint solving [30]. These heuristics ensure conformance to best visualization design practices. These systems are very useful for quickly exploring data and gathering insights while VIZSMITH is mostly intended for searching for a *specific* visualization. Nevertheless, VIZSMITH can also benefit from incorporating the design heuristics to better rank its output visualizations.

2) Reusable Visualization Templates: Ivy [33] allows users to build a specific visualization by choosing from a catalogue of visualization templates, which are similar in spirit to the reusable functions mined by VIZSMITH. However these templates are manually derived while VIZSMITH uses a combination of program analysis and metamorphic testing to collect its reusable functions.

3) Visualization Specifications: Similar to VIZSMITH, a number of visualization authoring systems accepting natural-language specifications from the user have been developed [34]–[36]. These systems use a carefully designed grammar or automaton to parse natural language queries describing the desired visualization and keep track of the interaction context. This grants users fine-grained control over the produced visualization. However the use of such a fixed grammar limits the space of visualizations that can be generated. Due to its use of mining, VIZSMITH can target different kinds of visualizations such as word-clouds, visualizations using different APIs and richer data transformation code such as computing correlations and cross-tabulations on top of the sorting, filtering and aggregation functionalities offered by the above systems. Finally, although this work only explores a simple keyword-based search to match snippets with user queries, recent advances in natural-language processing (NLP) [37], [38] could be leveraged to improve the search.

Richer modes of specification have also been explored. Falx [17], [18] allows users to provide pieces of the target visualization and the system utilizes program synthesis to generate the required data transformation and visualization code. This form of specification captures a lot more information about the desired visualization and thus Falx can complement VIZSMITH in cases where a very specific visualization is desired and cannot be described accurately with keywords.

4) Learning from Data: Data2Vis [39] trains deep learning models on pairs of dataset and visualization specifications obtained from the Vega-Lite corpus [40] and recommends visualizations given a dataset at inference time. However, it does not grant control over the columns or fields that are visualized which would force the user to pick out the desired visualization from a large set. PlotCoder [41] is the closest to VIZSMITH in that it generates visualization code from natural language that contains information about columns to visualize. However it restricts the set of visualizations to a subset of matplotlib, and cannot generate transformation code like VIZSMITH.

B. Code Mining and Reuse

Code mining and reuse has been employed in a number of other applications. The EG system [42] uses static analysis across large code-bases to build a database of usage examples for APIs which can be queried. However the examples are not always executable and thus their output cannot be shown on the user's input. For visualization synthesis, it is essential for the user to see the output visualization on their data to select the one that meets their needs, thus such an approach would not work in our problem setting.

AutoType [43] and TDE [44] synthesize executable programs using mined code corpora for type-validation and string transformation respectively. Phoenix [45] and Getafix [46] induce repair patterns from mined static analysis repairs. All these domains offer precise specifications—positive and negative examples for a type, input/output pairs for transformations and a pass/fail from the static analyzer. This greatly simplifies the filtering of bad mined code as one can simply check them against the specification. In VIZSMITH's setting, the lack of such a precise target necessitates the use of techniques such as metamorphic testing to weed out bad visualization functions.

Aroma [47] takes a different approach to reusability. It accepts a partial code snippet as input and performs code completion by searching over a large indexed code corpus and intersecting the search results. This intersection step ensures the completed code only contains elements that are common across a sufficiently diverse set of snippets and thus reusable.

X. CONCLUSION

We presented VIZSMITH, a tool which accepts a dataset, columns to visualize and a text query from the user and synthesizes visualization code. First, in an offline phase VIZSMITH mines Kaggle notebooks to create a database of 7176 *reusable* Python functions. It uses a novel metamorphic testing approach to automatically assess reusability of functions. When presented with the user query, VIZSMITH efficiently searches this database to find relevant functions, execute them and return the generated visualizations. We evaluated VIZSMITH and found that it can suggest the right visualization for 56% of the benchmarks. We also found that using reusability analysis helps improve the quality of visualizations and reduces the search space by 50%. VIZSMITH is available publicly at https://github.com/rbavishi/vizsmith-demo.

XI. ACKNOWLEDGEMENTS

We thank Caroline Lemieux, Karan Bavishi, and all our anonymous reviewers for their invaluable feedback on this paper. This research is supported in part by a grant from Fujitsu Research of America, and NSF grants CCF-1900968, CCF-1908870, and CNS-1817122.

References

[1] E. Segel and J. Heer, "Narrative Visualization: Telling Stories with Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, p. 1139–1148, Nov. 2010. [Online]. Available: https://doi.org/10.1109/TVCG.2010.179

[2] 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.

[3] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. D. Team, "Jupyter Notebooks - a publishing format for reproducible computational workflows," in *ELPUB*, 2016.

[4] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. [Online]. Available: https://ggplot2.tidyverse.org

[5] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[6] M. Gharehyazie, B. Ray, and V. Filkov, "Some from Here, Some from There: Cross-Project Code Reuse in GitHub," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, p. 291–301. [Online]. Available: https://doi.org/10.1109/MSR.2017.15

[7] C. Sadowski, K. T. Stolee, and S. Elbaum, "How Developers Search for Code: A Case Study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 191–201. [Online]. Available: https://doi.org/10.1145/2786805.2786855

[8] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How Do Developers Utilize Source Code from Stack Overflow?" *Empirical Softw. Engg.*, vol. 24, no. 2, p. 637–673, Apr. 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9634-5

[9] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss, "Hunter: Next-Generation Code Reuse for Java," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1028–1032. [Online]. Available: https://doi.org/10.1145/2950290.2983934

[10] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An Infrastructure for Large-Scale Collection and Analysis of Open-Source Code," *Sci. Comput. Program.*, vol. 79, p. 241–259, Jan. 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2012.04.008

[11] S. P. Reiss, "Semantics-Based Code Search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 243–253. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070525

[12] S. Wang, D. Lo, and L. Jiang, "Active Code Search: Incorporating User Feedback to Improve Code Search Relevance," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 677–682. [Online]. Available: https://doi.org/10.1145/2642937.2642947

[13] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-Automating Small-Scale Source Code Reuse via Structural Correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, p. 214–225. [Online]. Available: https://doi.org/10.1145/1453101.1453130

[14] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What Do Developers Search for on the Web?" *Empirical Softw. Engg.*, vol. 22, no. 6, p. 3149–3185, Dec. 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9514-4

[15] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL'18)*, 2018.

[16] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Industry Track of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. ACM, 2019, pp. 964–974.

[17] C. Wang, Y. Feng, R. Bodik, I. Dillig, A. Cheung, and A. J. Ko, "Falx: Synthesis-Powered Visualization Authoring," *arXiv e-prints*, p. arXiv:2102.01024, Feb. 2021.

[18] C. Wang, Y. Feng, R. Bodik, A. Cheung, and I. Dillig, "Visualization by Example," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371117

[19] "The kaggle data-science platform." [Online]. Available: https://www.kaggle.com/

[20] "Voice call quality customer experience." [Online]. Available: https://data.gov.in/catalog/voice-call-quality-customer-experience

[21] "Stacked bar chart." [Online]. Available: https://matplotlib.org/stable/gallery/lines_bars_and_markers/bar_stacked.html

[22] "How can i normalize data and create a stacked bar chart?" [Online]. Available: https://stackoverflow.com/questions/57337796/how-can-i-normalize-data-and-create-a-stacked-bar-chart

[23] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," *SIGPLAN Not.*, vol. 25, no. 6, p. 246–256, Jun. 1990. [Online]. Available: https://doi.org/10.1145/93548.93576

[24] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, p. 183–200, Feb. 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[25] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-Guided Program Reduction," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[26] E. Weyuker, "On Testing Non-Testable Programs," *Computer Journal*, vol. 25, 11 1982.

[27] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *Technical Report HKUST-CS98-01,*, 1998.

[28] G. Amati, *BM25*. Boston, MA: Springer US, 2009, pp. 257–260. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_921

[29] 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. [Online]. Available: http://idl.cs.washington.edu/papers/voyager

[30] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer, "Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, p. 438–448, Jan. 2019. [Online]. Available: https://doi.org/10.1109/TVCG.2018.2865240

[31] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-Lite: A Grammar of Interactive Graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, p. 341–350, Jan. 2017. [Online]. Available: https://doi.org/10.1109/TVCG.2016.2599030

[32] J. Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Trans. Graph.*, vol. 5, no. 2, p. 110–141, Apr. 1986. [Online]. Available: https://doi.org/10.1145/22949.22950

[33] A. McNutt and R. Chugh, "Integrated Visualization Editing via Parameterized Declarative Templates," *arXiv e-prints*, p. arXiv:2101.07902, Jan. 2021.

[34] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. Karahalios, "Datatone: Managing ambiguity in natural language interfaces for data visualization," *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*, pp. 489–500, 2015. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84958249800{&}partnerID=40{&}md5=f0eb3ceb834a66e6d0eb6b59ffc57163

[35] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A Natural Language Interface for Visual Analysis," *Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST '16*, pp. 365–377, 2016. [Online]. Available: http://doi.acm.org/10.1145/2984511.2984588

[36] E. Hoque, V. Setlur, M. Tory, and I. Dykeman, "Applying Pragmatics Principles for Interaction with Visual Analytics," *IEEE Transactions on Visualization and Computer Graphics*, no. c, 2017. [Online]. Available: dx.doi.org/10.1109/TVCG.2017.2744684

[37] J. Devlin, M.-W. 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, Volume 1 (Long and Short Papers)*. Minneapolis,

Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[38] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[39] V. Dibia and Ç. Demiralp, "Data2Vis: Automatic Generation of Data Visualizations Using Sequence to Sequence Recurrent Neural Networks," *arXiv e-prints*, p. arXiv:1804.03126, Apr. 2018.

[40] J. Poco and J. Heer, "Reverse-engineering visualizations: Recovering visual encodings from chart images," *Computer Graphics Forum*, vol. 36, no. 3, pp. 353–363, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13193

[41] X. Chen, L. Gong, A. Cheung, and D. Song, "Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 2169–2181. [Online]. Available: https://doi.org/10.18653/v1/2021.acl-long.169

[42] C. Barnaby, K. Sen, T. Zhang, E. Glassman, and S. Chandra, "Exempla Gratis (E.G.): Code Examples for Free," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1353–1364. [Online]. Available: https://doi.org/10.1145/3368089.3417052

[43] C. Yan and Y. He, "Synthesizing Type-Detection Logic for Rich Semantic Data Types Using Open-Source Code," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 35–50. [Online]. Available: https://doi.org/10.1145/3183713.3196888

[44] Y. He, K. Ganjam, K. Lee, Y. Wang, V. Narasayya, S. Chaudhuri, X. Chu, and Y. Zheng, "Transform-Data-by-Example (TDE): Extensible Data Transformation in Excel," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1785–1788. [Online]. Available: https://doi.org/10.1145/3183713.3193539

[45] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 613–624. [Online]. Available: https://doi.org/10.1145/3338906.3338952

[46] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[47] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360578