Subtle Bugs Everywhere: Generating Documentation for Data Wrangling Code

Chenyang Yang
Peking University

Shurui Zhou
University of Toronto

Jin L.C. Guo
McGill University

Christian Kästner
Carnegie Mellon University

***Abstract*—Data scientists reportedly spend a significant amount of their time in their daily routines on data wrangling, i.e. cleaning data and extracting features. However, data wrangling code is often repetitive and error-prone to write. Moreover, it is easy to introduce subtle bugs when reusing and adopting existing code, which results in reduced model quality. To support data scientists with data wrangling, we present a technique to generate documentation for data wrangling code. We use (1) program synthesis techniques to automatically summarize data transformations and (2) test case selection techniques to purposefully select representative examples from the data based on execution information collected with tailored dynamic program analysis. We demonstrate that a JupyterLab extension with our technique can provide on-demand documentation for many cells in popular notebooks and find in a user study that users with our plugin are faster and more effective at finding realistic bugs in data wrangling code.**

***Index Terms*—computational notebook, data wrangling, code comprehension, code summarization**

I. Introduction

It has been reported that data scientists spend a significant amount of time and effort on data cleaning and feature engineering [1], the early stages in data science pipelines, collectively called *data wrangling* [2] in the literature. Typical data wrangling steps include removing irrelevant columns, converting types, filling missing values, and extracting and normalizing important features from raw data. Data wrangling code is often dense, repetitive, error-prone, and generally not well-supported in the commonly used computational-notebook environments.

Importantly, data wrangling code often contains subtle problems that may not be exposed until later stages, if at all. In our evaluation, we found dozens of instances of suspicious behavior, such as computations that use the wrong source, that are not persisted, or that inconsistently transform part of the data. Although they do not crash the program, they clearly violate the code's apparent intention (often specified in comments and markdown cells), thus we consider them as bugs. Unfortunately, as tests are very rare in data science code in notebooks [3], these bugs remain undetected even for many highly "upvoted" notebooks on popular data science sites like Kaggle.

In this work, we propose to automatically generate *concise summaries* for the data wrangling code and purposefully *select representative examples* to help users understand the impact of the code on their data. This form of *automated documentation* is useful for multiple scenarios that require code understanding:

- *Debugging:* Data wrangling code is often concise, sequencing multiple nontrivial data transformations, as in our example in Fig. 1a, but also usually not well tested [3]. Data scientists currently mostly rely on code reading and inserting print statements to look for potential problems.
- *Reuse:* Data scientists heavily reuse code through copying and editing code snippets, often within a notebook, from other notebooks, from tutorials, or from StackOverflow [4]. At the same time reusing data wrangling code can be challenging and error-prone [5], especially if the reused code needs to be adapted for the data scientist's own data.
- *Maintenance:* Data science code in notebooks is often not well-documented [3], [6], [7], yet data science code needs to be maintained and evolve with changes in data and models, especially when adopted in production settings [8]. To avoid mistakes in maintenance tasks and degrading model quality over time, understanding existing data wrangling code and assumptions it makes is essential.

Our work is inspired by past work on *code summarization* to automatically create summaries of code fragments that could serve as documentation for various tasks. However, while existing code summarization work [9] tries to characterize what a code fragment does generally (for all possible inputs), our approach summarizes what *effect* code has on specific input data in the form of a dataframe, highlighting representative changes to rows and columns of tabular data. Given the data-centric nature of data wrangling code, understanding the effect that data wrangling code has on the data often is the immediate concern for data scientists. To the best of our knowledge, this is a novel view on summarization, tailored for debugging, reuse, and maintenance tasks of data scientists.

Moreover, our approach generates the documentation *on demand* for the code and data at hand to help with program comprehension. This is achieved by instrumenting data science code to collect runtime data and select data-access paths and branches executed at runtime, using *program synthesis techniques* to generate short descriptive summaries, and using techniques inspired by *test-suite minimization* to select and organize examples. We integrate our tool, WrangleDoc, in JupyterLab, a commonly used notebook environment.

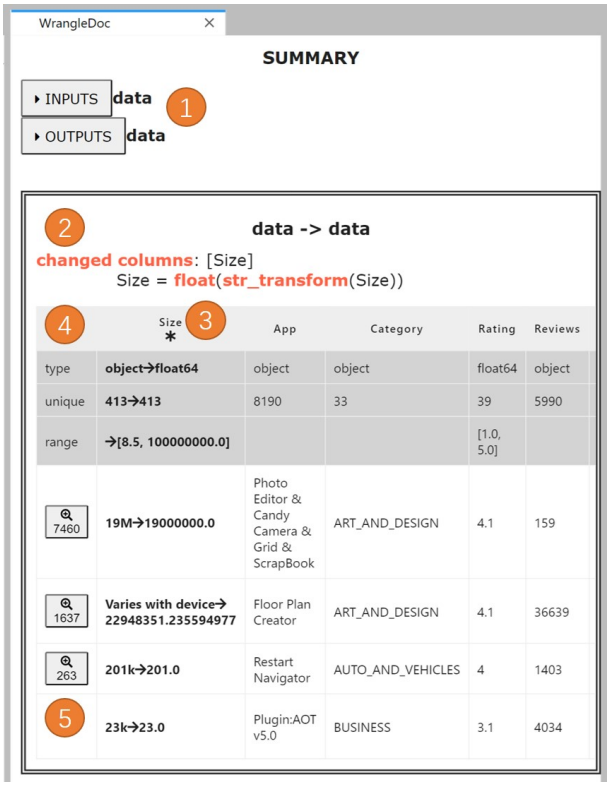
We evaluated our approach and tool in two ways. First, we conducted a case study with 100 Kaggle notebooks to evaluate

```
data = pd.read_csv('./data.csv')
# x = load some other data that's not relevant for the
    next cell
```

```
# first change 'Varies with device' to nan
def to_nan(item):
    if item == 'Varies with device':
        return np.nan
    else:
        return item

data['Size'] = data['Size'].map(to_nan)

# convert Size
num = data.Size.replace(r'[kM]+$', '', regex=True).
    astype(float)
factor = data.Size.str.extract(r'[\d\.]+([KM]+)', expand
    =False)
factor = factor.replace(['k','M'], [10**3, 10**6]).
    fillna(1)
data['Size'] = num*factor.astype(int)

# fill nan
data['Size'].fillna(data['Size'].mean(), inplace = True)
```

```
# some training code reading combined
targets = data['Target']
data.drop('Target', inplace=True, axis=1)

clf = RandomForestClassifier(n_estimators=50,
    max_features='sqrt')
clf = clf.fit(data, targets)
```

(a) Three notebook cells, loading tabular data, transforming the 'Size' column (converting k and M to numbers and replacing 'varies with device' by mean value), and learning a model from the data. While this code is fairly linear and relies heavily on common APIs, it encodes nontrivial transformations compactly, that are not always easy to understand.



(b) WRANGLEDOC Interface: documentation of the second cell; ①: data flow into or out of the cell, ②: concise summary of changes, ③: highlighting changed columns, ④: meta information (type, unique, range) for columns, ⑤: selected examples.

Fig. 1: Excerpt of real data wrangling code from a Kaggle competition and corresponding generated documentation with WRANGLEDOC. Due to case sensitivity in regular expressions, values with a 'k' are not transformed correctly, as easily visible in the generated summary.

correctness and runtime overhead and additionally explore the kind of documentation we can generate for common notebooks. Second, we conducted a human-subject study to evaluate whether WRANGLEDOC improves data scientists' efficiency in tasks to debug notebooks. Through the two studies, we provide evidence that our approach is both practical and effective for common data wrangling code.

Overall, we make the following contributions:

- An approach to summarize data transformations for data wrangling code, based on program synthesis.
- An approach to purposefully select rows to illustrate changes in data wrangling code, inspired by test suite minimization techniques.
- A prototype implementation as a JupyterLab plugin.
- Empirical evidence showing that our approach can accurately generate summaries for nontrivial, real-world data science code with acceptable overhead.
- A user study finding that our approach improves data scientists' efficiency when debugging data wrangling code.

We share the tool and our supplementary material on GitHub.[1]

[1]https://github.com/malusamayo/notebooks-analysis

II. DESIGN MOTIVATIONS

Many prior studies explored practices of data scientists in notebooks and challenges that they face. With the surging interest in machine learning, notebooks are a very popular tool for learning data science and for production data science projects [6], [7], [10], [11], used by data scientists with widely varying programming skills and software engineering background. Data science work is highly exploratory and iterative [11]–[13] with heavy use of copy-and-paste from other notebooks and online examples [14]. While researchers found wide range of challenges, including reproducibility [3], [15], [16], collaborative editing [17], [18], and reliability [5], we focus on challenges regarding comprehension and debugging.

Data wrangling code can be challenging to understand: Although it is typically linear and structured in short cells, data wrangling code can be dense and make nontrivial transformations with powerful APIs, as in our example (Fig. 1).

To better capture how data scientists approach understanding data wrangling code, we conducted a small informal experiment, in which we gave four volunteers with data science experience a notebook and two tasks that required program comprehension. Specifically, we asked them to modify the

notebook to accommodate changes to the input dataframe and to look for possible improvements of model performance, all while thinking aloud [19].

We observed two main strategies that our participants used to understand data wrangling code. On the one hand, they frequently reasoned *statically* about the code, inspecting the code line by line without running it. In this process, they often left the notebook to look up the API documentation and code examples as needed for the numerous functions in the used data science libraries, such as *extract* and *replace* and their various arguments in our example. On the other hand, they also reasoned *dynamically* by observing executions. Our participants frequently injected *print* statements at the beginning and the end of cells, or in a new cell, to inspect data samples (typically the first few rows) and manually compare them before and after the data wrangling steps. We saw that dynamic reasoning quickly became overwhelming and tedious with large amounts of data, especially if data triggering problematic behavior is not part of the first few rows. In our example (Fig. 1a), the first five rows of the 9360 rows contained sizes ending with the letter 'M' and containing the value of *'Varied with device'*, but not sizes ending in 'k', which makes the incorrect transformations of 'k' ending rows difficult to spot.

Existing tools are limited: Notebook environments are evolving and various new tools are proposed by practitioners and researchers [20]. For example, more recent notebook environments now provide code completion features and can show API documentation in tooltips; the IDE PyCharm and JupyterLab extensions integrate a traditional debugger—all standard features in IDEs. Several extensions, like *pandas profiling* [21], help inspect data stored in variables.

Yet tool support for understanding data wrangling code is still limited and does not well support the activities we observed. Classic tools like debuggers, if available at all, do not provide a good match for data-centric, linear, and often exploratory notebook code, where a single line can apply transformations to thousands of rows at once and actual computations are performed deep in libraries (often in native code). Tools for exploring data in variables are useful for understanding data at one point in time, but do not help in understanding complex transformations within a cell.

Data wrangling code is frequently buggy: Several researchers have pointed out code quality problems in notebooks [11], [22], [23]. Notebooks almost never include any testing code [3] and practitioners report testing as a common pain point [5]. The commonly used data wrangling APIs are large and can be easily misunderstood [24]. Due to the dynamic and error-forgiving nature of Python and the Pandas library design, buggy code often does not crash with an exception but continues to execute with wrong values, which could subsequently reduce model accuracy.

It is generally easy to introduce mistakes in data wrangling code, which became very obvious when we inspected examples of documentation generated with our tool on popular notebooks (some among the most upvoted notebooks on Kaggle). Without actively looking for bugs (it is not always clear what the code intends to do), we found many examples with subtle problems in data wrangling code.

API misuse

```
1  # attempting to remove na values from column, not table
2  df['Join_year'] = df.Joined.dropna().map(lambda x: x.
       split(',')[1].split(' ')[1])
3
4  # loc[] called twice, resulting in assignment
5  # to temporary column only
6  df.loc[idx_nan_age,'Age'].loc[idx_nan_age] = df['Title'
       ].loc[idx_nan_age].map(map_means)
7
8  # astype() is not an in-place operation
9  df["Weight"].astype(str).astype(int)
```

Typos

```
10 # reading from wrong table (should be df2)
11 df2['Reviews_count'] = df1['Reviews'].apply(int)
```

Data modelling problems

```
12 # converting money to numbers, e.g., '10k' -> 10000.0
13 # ignoring decimals, thus converting '3.4k' to 3.4000
14 df["Release Clause"]= df["Release Clause"].replace(regex
       =['k'], value='000')
15 df["Release Clause"]= df["Release Clause"].astype(str).
       astype(float)
```

Fig. 2: Examples of subtle bugs in data wrangling code, ranging from data cleaning stage (e.g., normalizing the column 'Reviews' to integers) to feature engineering stage (e.g., extracting new feature 'Join_year' from the 'Joined' column).

For example, there is a subtle bug in our example in Fig. 1a where the code tries to convert 'k' to 1000 and 'M' to 1,000,000 in download counts: A capitalized 'K' in Line 14 results in converting 'k' to 1 instead of 1000. The code executes without exception, but produces wrong results, e.g., 670.0 for '670.0k' rather than the intended 670000.0. The problem could have been found easily if one could observe example transformations with 'k'.

In Fig. 2, we illustrate three kinds of problems in data wrangling code that we found *repeatedly* across 100 popular notebooks in our evaluation (described later in Section V-A):

- **API misuse** is common where a function call looks plausible, but does not have the intended effect on the input data (e.g., *dropna* does not remove the entire row of a table if applied to a single column). This commonly results in computations that are not persisted and have no effect on the data used later.
- Simple **typos** in variable names, column names, and regular expressions are the source of many other problems, often leading to wrong computations.
- Finally, multiple problems relate to **incorrect modeling of data**, often stemming from wrong assumptions about the different kinds of data in the dataset, thus missing rare cases.

All the above problems can be difficult to locate without a clear and thorough understanding of the API specifications, how they are used in the data wrangling code, and the impact

on the specific instances from the input dataset.

III. Solution Overview

Before we describe the technical details of how we generate documentation, let us illustrate the kind of documentation we generate with WrangleDoc from a notebook user's perspective. In a nutshell, we summarize the changes a code fragment (typically a notebook cell) performs on dataframes and show them in a side panel through a JupyterLab extension, as illustrated in Fig. 1b for our running example. Our documentation includes the following pieces of information:

①: We identify the dataframes (tabular variables) that flow in and out of the code fragment to identify which important variables are read and written in the code fragment. To avoid information overload, we deliberately include only variables that are later used in the notebook again, but not temporary variables. In our running example, the dataframe *data* is changed for subsequent cells, whereas we omit temporary variables *num* and *factor* from our documentation.

②: We provide a concise summary of the transformations for each changed dataframe using a domain specific language (DSL) we designed. The summary describes which columns of the dataframe were added, removed, or changed, how columns were changed, and whether rows were removed. The summary intentionally uses somewhat generic function names like **str_transform** to indicate that strings were manipulated without describing the details of that transformation, which can be found in the code. These summaries provide a quick overview of what the code does, helping to ensure that the (static) understanding of APIs aligns with the observed execution. It is particularly effective at highlighting "data not written" bugs, where the summary would clearly indicate that no data was changed. For example, data scientists can easily spot all *API misuse* bugs in Fig. 2 when they encounter the unexpected summary of "no changes" for their transformation code. Similarly, the *typos* bug in Fig. 2 can also be surfaced as the summary "Review_count = **int**(**merge**(Reviews))" would show that different items are merged, whereas the code intends to convert strings to integers without merging.

③–④: We show sample data from the modified dataframes, specifically comparing a dataframe's values before and after the cell: The summary highlights which columns have been modified (③), highlights changes to column data and metadata, including types, cardinality, and range of values (④). This direct before-after comparison highlights the changes that would usually require manual comparison of two dataframes, hence reducing the manual efforts of comparing the output of print statements in dynamic debugging.

⑤: Finally, where classic print statements would simply show the first few rows of long dataframes, our documentation purposefully groups rows that take the same path at branching decisions in transformation code, showing one example each and highlighting the number of other rows that take the same path. Grouping rows by transformation decisions draws attention to paths that may not occur in the first few rows, making it easier to spot potential problems. For example, this

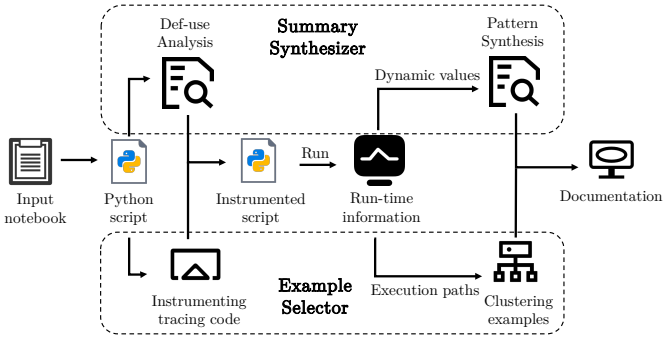


Fig. 3: Approach overview.

makes the bug in Fig. 1 that does not transform 'k' to 1000 obvious, even though it occurs only in 3 percent of all rows and not in any early ones. Our approach enables the data scientists to examine those rare examples and corner cases effectively.

As we will show, the above forms of documentation support effective static and dynamic reasoning, which is the foundation of various debugging, reuse, and maintenance tasks, and they help surface subtle bugs in data wrangling code.

IV. WrangleDoc: Synthesizing Summaries and Selecting Examples

WrangleDoc generates documentation on demand with two components: *Summary Synthesizer* and *Example Selector*. Both collect information by analyzing and instrumenting a notebook's code and observing it during its execution—see Fig. 3 for an overview. The *Summary Synthesizer* gathers static and run-time information about access to dataframes and columns and runtime values of dataframes before and after a cell to synthesize summary patterns (Fig. 1b, ①–④). The *Example Selector* traces branching decisions during data transformations to cluster rows in a dataframe that share the same execution paths (Fig. 1b, ⑤).

A. Summary synthesis

The goal of synthesizing summaries is to derive a concise description of how data is transformed by a fragment of data wrangling code, typically a notebook cell. To avoid distracting users with implementation details, which may use nontrivial API sequences, external libraries, and custom code, we synthesize summaries that describe the relationship between data *before* and *after* the code fragment. Through instrumentation, we collect data of all variables (with emphasis on tabular data in dataframes) before and after the target code, from which we synthesize summaries that explain the differences, such as added columns, removed rows, or changed values.

a) Synthesis approach: As in all summary generation, there is an obvious tradeoff between providing concise summaries (e.g., *'dataframe X was changed'*) and detailed summaries (e.g., *"column Y was added with values computed by removing all dashes from column Z, replacing 'K' at the end of the string by 1000, and then converting the result into a number"*). Summaries at either extreme are rarely useful: Too

```
<df> := createCol(<df>, COL, <col>)0
     | modifyCol(<df>, COL, <col>)0 // modify column
     | removeCol(<df>, COL)0
     | removeDuplicateRows(<df>, ROW+, COL*)0
     | removeNullRows(<df>, ROW+, COL*)0
     | removeRows(<df>, ROW+, COL*)1 // remove some rows
     | rearrangeCols(<df>, COL*, COL*)0 // change col. order
     | rearrangeRows(<df>, ROW*, ROW*)0 // change row order
     | concatRows(<df>, <df>)0 // concat dataframes by rows
     | DATAFRAME0
     | compute()15 // unspecified dataframe computation

<col> := fillna(<col>)2 // fill null values
     | merge(<col>)2 // merge items to reduce cardinality
     | category(<col>)2 // convert columns to category type
     | float(<col>)2 | str(<col>)2 | int(<col>)2
     | bool(<col>)2 | datetime64(<col>)2 // type conversion
     | encode(<col>)2 // encode columns in consecutive ints
     | one_hot_encoding(<col>)2 // encode columns in 0/1 ints
     | type_convert(<col>)3 // other type conversion
     | str_transform(<col>)3 // unspecified string transf.
     | num_transform(<col>)3 // unspecified numerical transf.
     | compute(<col_ref>*)15 // unspecified col. computation
     | <col_ref>*0

<col_ref> := DATAFRAME.COL0
```

Fig. 4: The DSL for data transformation. Cost per expression is indicated as subscript.

concise summaries do not convey much information, whereas too detailed summaries might just paraphrase the code and provide little benefit over reading the code directly.

Our summary synthesis aims to find a balance by describing data wrangling code as expressions formed from an extensible grammar of *transformation patterns*, which describe the impact of the transformations concisely and unambiguously. For example, pattern **fillna** describes that null values are filled, but not any details about how the new values are computed; pattern **str_transform** describes that a column with string values has been manipulated, but not how; pattern **one_hot_encoding** describes that a binary column was created for distinct values in a source column. We include generic catch-all summaries for transformations we cannot further explain, such as **compute** for generating data from unknown processes. While the pattern language is easily extensible, we start with the most used patterns shown in Fig. 4, which we derived by manually summarizing and grouping common transformation patterns in dozens of sampled notebooks from GitHub. Our DSL supports abstracting multiple transformation patterns in one combined expression. The operation of applying **float** to the 'Size' column in data after **str_transform** in the running exampling in Fig. 1a will be synthesized as *"**modifyCol**(data, Size, **float**(**str_transform**(data.Size)))*" (cf. Fig. 1b, ②).

During the summary synthesis, the synthesis engine searches for the expressions in the pattern language that match the input-output examples created by the target code. Because there could exist multiple matches for the same input-output pair, we design a cost function to select the best matched expression. Concretely, we assign a higher cost to more generic patterns (see Fig. 4) and consider the cost of an expression as the sum of the cost of all patterns used in that expression. This way, our synthesis engine favors concise but concrete patterns.

```
1  .decl apply(depth:number, pattern:Pattern)
2  .decl var(depth:number, type:ColType, nafilled:boolean,
       carddrop:boolean, onehot:boolean, encode:boolean)
3
4  var(depth + 1, type, true, carddrop, any, any) :-
5      apply(depth, "fillna"),
6      var(depth, type, false, carddrop, _, _)
```

Fig. 5: We describe propagation of knowledge about rules in terms of Datalog relations and illustrate the propagation rule for "fillna". In line 1-2, there are two Datalog facts: **apply** and **var**. We use **var** to track a column's attributes (e.g., whether its missing values are filled), and we use **apply** to represent application of a pattern. The rule in line 4-6 propagates **var** through a pattern "fillna". The attribute 'nafilled' is set to true, while 'onehot' and 'encode' are set to any, because any prior information about these is lost in the possible transformations.

While this kind of search is standard for program synthesis, the novelty of our approach is in encoding summary generation as a synthesis problem, not in the synthesis algorithm itself.

b) Synthesis implementation: For the synthesis, we follow a standard top-down enumerative search to explore expressions of a given grammar, as shown in Algorithm 1. The synthesis engine incrementally enumerates expressions allowed by the grammar by substituting non-terminals with new expressions in order of increasing costs; the synthesis prunes the search space for partial expressions (i.e., contain holes for some nonterminals) that cannot explain the input-output difference; it returns the expression that explains the input-output difference with the lowest cost as all expressions are maintained in a priority queue. The synthesis problem can be encoded in standard synthesis engines like Rosette [25], but for simplicity, we implemented our own.

To validate whether a (partial) expression can explain the input-output pair, we compute and check the result of an expression against the output. Aside from parts that can be easily checked concretely, such as the removal of columns and rows, many patterns express generic computations that can match many input-output examples (e.g., **fillna**). Here, we reject infeasible expressions similar to how static analyses identify problematic computations [26]: We track abstract facts across patterns and each pattern has a transfer function that can generate or kill facts. Throughout these patterns, we track each column's type (if known), whether it has missing values, its length, its cardinality, and whether it looks like a common encoding pattern. This way we can check that **fillna** actually filled missing values without having to know concrete filled values or having to worry about interactions with other patterns. In Fig. 5, we show an example rule for propagating facts when applying "fillna". Details of the other validation rules can be found in the supplementary material.

c) Data gathering and optimization: To collect the values of all variables before and after the target data wrangling code, we instrument the user's code to store values into files at runtime. To target the instrumentation, we use a simple static

Algorithm 1 Enumeration-based synthesis algorithm

Input:
e ▷ Input-Output example pair
C ▷ Cost function for transformation patterns
s ▷ A set of accessed columns

function SYNTHESIS(e, C, s)
  $q \leftarrow [\langle df \rangle]$ ▷ Priority queue of patterns, ordered by C
  $top \leftarrow$ "compute()"
  while q is not empty **do**
    $cur \leftarrow q.dequeue()$
    if $C(cur) > C(top)$ **then**
      break ▷ Prune because elements left have higher cost
    if cur is fully resolved **then**
      if $validate(e, cur)$ **then**
        $top \leftarrow cur$
    else
      $new\_patterns \leftarrow extend(cur, s)$
      for p in $new\_patterns$ **do**
        if $is\_feasible(e, p)$ **then**
          $q.add(p)$
  return top

def-use analysis (adopted from prior work [23]) to record only values of variables that flow into or out of the target code.

In addition, we collect the sequence of access paths for read and write access to columns of dataframes by dynamically intercepting such access, to limit the columns over which to search as part of the synthesized expressions. Furthermore, when synthesizing column expressions, we consider only source columns for which we observed read access before write access to the target column.

In our current implementation, we consider each cell as a separate segment for which we gather data and synthesize summaries, but it is also possible to consider each line as a segment if more granular documentation is desired. For example, users could interactively indicate what to consider as a segment, or a tool could automatically cluster lines or cells [27]–[29].

d) Presenting results: In the user interface, we present transformations for all changed dataframes. As exemplified in Fig. 1b, we translate the synthesized expressions into a structured format ②, separated by affected column. In addition, we highlight parts of the transformation patterns in a table of example data, showing but crossing out removed columns, highlighting changed columns ③, and a few patterns, such as type conversions, as part of the table metadata ④.

B. Example selection

As discussed, data scientists often inspect the data before and after a cell with temporary print statements, but they may miss issues that are not visible in the first few rows. Our insight is to group rows that are affected by the same transformation paths, highlighting examples from each path to reveal data that fit unusual processing conditions, which may be of particular interest in understanding and debugging tasks.

Inspired by *test-suite minimization* techniques that select a small number of test cases from a larger pool to trigger different execution paths [30], [31], we group rows in a dataframe by the branching decisions taken in the data wrangling code, thus identifying a minimum number of rows that trigger different execution paths. Conceptually, our strategy can be considered as inspecting path coverage within data wrangling code where each row is treated as a test input to the code.

If a function is applied to every row separately, we can treat each row as separate test input and track branching decisions within the supplied function. However, most transformations in data science code are applied to entire tables or vectors at once, through functions like *replace*, *fillna*, and *map*, possibly parallelized. Also many decisions in common functions like *split* and *replace* happen deep in libraries or native code. For simplicity and manageable overhead, we instrument common library functions on tables and vectors, each producing a vector of decision outcomes for every instrumented function that has internal branching decisions. For example, for *replace* we record whether the search term has been replaced and for *split* we record the number of splits. Only for user-defined functions supplied to *map* or *apply* functions we collect branching decisions with a tracer as a traditional test coverage tool does.

Using the vector that represents the branching decision of every decision point for each row, we *group* all rows that took the same path through all decisions. For our motivating example, we have branching decisions corresponding to the *if* statement in the function supplied to *map* and in the calls to *replace*, *extract*, and *fillna*. In this case, the input data takes three distinct paths, shown in Fig. 1b ⑤.

Presenting results: When displaying the examples for the target data wrangling code, we show only the first row from each group, while indicating how many additional rows there are in each group, and provide a button to show more examples (Fig. 1b, ⑤). In a tooltip, we also show the branching decisions taken in each group. This presentation highlights the most common transformations and also directs the user's attention to uncommon transformations, where the subtle bugs often occur in data science code.

C. Implementation

We implemented our synthesis and example selection engine as an extension to the popular JupyterLab environment, called WRANGLEDOC. The goal of this prototype implementation is to demonstrate feasibility and allow experimentation with a best-effort approach, though a fully-featured industry-ready implementation would likely need to extend this.

Specifically, our *DSL patterns* are derived transformations observed in a convenience sample of notebooks on GitHub. While all transformations can be explained with the generic "compute" pattern, we are likely missing more specific patterns that may be useful in some settings. For *example selection*, we instrumented 24 common library functions that we observed in the same sample, including *str.replace*, *pandas.Series.fillna*, and *pandas.DataFrame.apply*. We also instrumented if expression that may lead to different branches with the same line number. We do not claim that instrumentation is exhaustive in our prototype, but the implementation can be easily extended with support for more functions.

TABLE I: Characteristics of our subject notebooks, 25 per dataset, reporting averages for code length, number of code and text cells, and number of named and lambda functions.

Dataset	LoC	code cells	text cells	funct.	lambda
Titanic	516	54	49	5.2	2.2
Google Play	195	42	24	1.5	5.2
FIFA19	328	44	25	4.0	1.0
Airbnb	267	47	33	0.9	0.4
All	327	47	33	2.9	2.2

V. Empirical Study of Popular Notebooks

We evaluate our approach both regarding technical capabilities on popular notebooks (this section) and how it helps users debug notebooks in action (next section).

We start by exploring to what degree WrangleDoc can generate documentation in data wrangling cells of a set of notebooks and how accurate that documentation is: *How often are the synthesized summaries correct for data wrangling cells?* ***(RQ1)***

Next, we report statistics about patterns and branching characteristics observed to capture to what degree data wrangling code performs nontrivial transformations for which documentation is likely useful: *What are typical characteristics of explanations for data wrangling code?* ***(RQ2)***

Finally, we measure the overhead introduced by our instrumentation and the computational effort required to synthesize the documentation to capture to what degree WrangleDoc can be used in an interactive setting: *How much overhead does* WrangleDoc *introduce?* ***(RQ3)***

A. Notebook selection

To answer the research questions, we apply our approach to a set of notebooks that were not used during our tool's development. There are several challenges with assembling a corpus for our study: First, while millions of public notebooks are shared on sites like GitHub, they are often from class projects, of low quality, and challenging to reproduce, e.g., due to missing data or library dependencies [3]. In addition, many notebooks start with an already cleaned dataset and focus more on modeling than data cleaning and feature engineering. Hence we decided to curate a small but diverse corpus of reproducible popular notebooks with significant data wrangling code.

We decided to sample high-quality notebooks from popular Kaggle competitions that provide tabular data in a raw format. Kaggle is the largest social platform for data science competitions, where users can upload datasets and corresponding challenges and others can submit and rate solutions in the form of notebooks. Popular challenges often have thousands of submitted solutions. We select multiple solutions per competition, as they share the same setup. Specifically, we select four popular competitions: Titanic, Google Play Store, FIFA 19 and Airbnb [32]–[35]. From each competition, we choose the top 25 notebooks based on "most votes" after filtering non-Python notebooks, task-irrelevant notebooks (not solving the modeling task, typically tutorials for Python libraries), and notebooks we could not reproduce due to missing or outdated dependencies. We sampled each competition with careful consideration: the Titanic dataset is usually used for educational purposes; many solutions are written as educational notebooks for data science learners. Google Play and FIFA 19 are selected based on their popularity, representative of trending notebooks on Kaggle. Finally, we selected Airbnb as a challenge that uses multiple larger datasets, which provides a better approximation of production setting.

Notebooks in our corpus have many characteristics that are similar to those found in other large scale studies of publicly available notebooks [3], [6]: As shown in Tab. I, they are typically long and split into many cells, rarely abstract code into functions or lambda expressions, though they contain more text cells than most public notebooks.

B. Summary correctness (RQ1)

First, we analyze correctness of the synthesized summaries for the sampled notebooks. We are not aware of an automated procedure that could test correctness, other than how we validate patterns during the synthesis process in the first place, hence we *manually* judge whether the summary corresponds to the actual transformation in the cell. To gain confidence in the manual judgement, multiple authors independently analyze a subset of cells to establish inter-rater reliability.

Research design: We proceed in four steps. First, we identified all cells from the 100 sampled notebooks that create or modify any dataframes, by monitoring state changes of script execution. We found 1401 such cells writing to 1998 dataframes. Second, we synthesized documentation for all the 1401 cells and prepared a user interface to show that documentation, including tool tips explaining each involved pattern. Third, to establish a reliable judgement process, we created a rubric on how to evaluate correctness of the synthesized summary (shared in supplementary material). Using the rubric, four authors independently judged the correctness of 20 randomly selected dataframes and their evaluation achieved excellent inter-rater agreement (0.88 according to free-marginal kappa [36]). Finally, having established reliability of judgement, another 80 randomly sampled dataframes were judged by a single author. Analyzing 100 out of 1998 dataframes gives us a margin of error of less than 10 percent at a 95 % confidence level.

Results: WrangleDoc created correct summaries for 92 of the 100 inspected dataframes that were created or changed in sample code. In six out of the eight incorrect cases, WrangleDoc did not create a summary even though data was changed due to limitations of our current def-use analysis — it can not catch modifications to a dataframe through indirect references. The remaining two cases are operations not currently covered by our patterns, involving reindexing rows and performing a join operation, for which WrangleDoc synthesized a wrong pattern expression. The results provide confidence that WrangleDoc's summaries are indeed mostly correct.

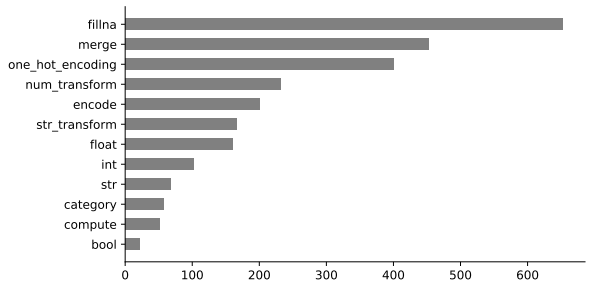


Fig. 6: Top 12 patterns observed in our dataset.

C. Wrangling characteristics (RQ2)

After establishing the accuracy of our generated summaries, we collect statistics on the summaries and examples generated for all 1998 modified or created dataframes in the cells of the 100 sampled notebooks, demonstrating that WRANGLEDOC provides summaries for nontrivial transformations.

Summary characteristics: Among the 1998 dataframes that are created or modified in cells, WRANGLEDOC does not create documentation for 971 dataframes, almost always because those variables are not used outside the cell (temporary variables, such as *num*, *factor* in Fig. 1a) and in a few cases because of limitations of the static analysis (see RQ1). For the remaining 1027 created or changed dataframes, WRANGLEDOC creates summaries: 39% of them involve modifying columns, 26% adding columns, 34% removing columns, 25% removing rows, and 14% rearranging columns or rows. None of them are summarized with a generic dataframe-level **compute**.

As part of these summaries, WRANGLEDOC generated expressions for 3754 created or modified columns. Of those column-level expressions, 79.5% consist of a single pattern and 19.1% are expressions with 2–3 patterns; only for 1.4% of columns we cannot provide a more descriptive pattern than a generic **compute**. The most common patterns in column-level summaries are **fillna**, **merge**, **one_hot_encoding**, **num_transform**, and **encode** (see Fig. 6 for details), which matches typical intuition about data wrangling code: Handling missing values is a common and important cleaning task, reducing variability in data by merging or grouping data is common for feature engineering, as are various (often numerical or binary) encodings.

Although the studied notebooks differ considerably in size, most of those notebooks contain a substantial amount of data wrangling code that uses many different patterns across different cells. More than half of the notebooks use over 5 patterns throughout their explanations.

The summaries expressed through our patterns are naturally very concise and often not immediately obvious from the notebook's code. For example, when inspecting 10 cells with **encode** pattern, we found that they follow radically different strategies to implement similar encodings, as exemplified in Fig. 7. Such level of diversity challenges simple template-based code summarization strategies, while our synthesis approach can identify more general patterns.

```
data['Initial'].replace(['Mr','Mrs','Miss','Master','
    Other'],[0,1,2,3,4],inplace=True)
```

```
title_mapping = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master":
    4, "Rare": 5}
dataset['Title'] = dataset['Title'].map(title_mapping)
```

```
combine["Deck"] = combine["Deck"].astype("category")
combine["Deck"].cat.categories = [0,1,2,3,4,5,6,7,8]
combine["Deck"] = combine["Deck"].astype("int")
```

```
le = preprocessing.LabelEncoder()
le = le.fit(df_combined[feature])
df_train[feature] = le.transform(df_train[feature])
```

Fig. 7: Widely different implementations can be summarized as **encode**.

Example selection: Our example selection technique is also used in the explanations of many cells. Out of the 1027 dataframes for which WRANGLEDOC generates documentation, 54% have at least one branching decision in their computation, resulting in grouping rows into at least two groups. We observed a total of 1240 executions of statements that introduce branching decisions, of which 96.3% are common API functions applied to entire columns (most commonly *fillna*, *replace*, and *loc*), and only 3.7% are from *if* expressions within user-defined functions. This confirms our observation that, instead of writing their own functions, data scientists heavily rely on common APIs for data wrangling code.

We also found a few cells with large numbers of branching decisions, resulting in identifying more than 20 groups, though most outliers represent long cells that contain multiple data wrangling steps (likely from copied code) that could reasonably be split into smaller more focused steps. For a few cases, the large number results from a large number of new one-hot-encoding columns, each of which executes a different path.

D. Analysis Overhead (RQ3)

To analyze the overhead of WRANGLEDOC, we measured wallclock time for executing the sampled notebooks with and without our tool and the time for static analysis and synthesizing summaries on a Linux machine with a 4-core Intel(R) Xeon(R) CPU (E5-2686 v4) and 16GB memory. The static def-use analysis is near instantaneous, and the synthesis is also usually very fast, on average 2 seconds per target dataframe, but the instrumentation does slow down execution by 309 percent on average (see Tab. II). In a typical usage scenario where one analyzes one cell at a time, most cells execute fairly quickly and the imposed overhead would add only a few seconds, which is acceptable for interactive use.

E. Threats to validity

Regarding external validity, while we carefully selected a sample of notebooks, readers should be careful in generalizing the results to other notebooks, such as lower-quality notebooks, educational notebooks, production notebooks, or notebooks using different libraries or datasets (e.g., images).

TABLE II: Tool overhead: all metrics are average of all notebooks in the dataset. Execution time refers to the time to run the entire notebook (wallclock time) without and with our tool's instrumentation.

Dataset	Original/Instr. Exec. Time (s)	Slowdown (%)
Titanic	33.1/60.5	182.7%
Google Play	25.3/59.1	233.5%
FIFA19	35.9/121.8	339.1%
Airbnb	90.3/328.3	363.5%
All	46.3/143.2	309.3%

Regarding internal validity, results must be interpreted within the limitations of the used research design. First, despite careful evaluation of inter-rater reliability we cannot entirely exclude bias and subjectivity that may arise from human judgment in our evaluation. Second, we analyze only the *correctness* of transformations, but not their quality: While every transformation can be expressed, worst case, with pattern **compute**, the synthesized pattern may not be the most specific and informative ones that could be expressed with our language or with patterns not in our language. We chose to evaluate correctness and completeness instead of quality because it is unclear how to define and measure quality reliably. Third, we do not evaluate whether we miss execution paths during example selection due to the difficulty of establishing firm ground truth.

VI. Controlled experiment on debugging notebooks

In addition to the technical evaluation in Section V, we conducted a human-subject controlled experiment to evaluate to what degree our JupyterLab extension actually supports data scientists in their work, asking: *How well does* WrangleDoc *improve data scientists' efficiency at finding data wrangling bugs?* ***(RQ4)***

Specifically, we asked participants to find four bugs in notebooks, with and without WrangleDoc. Our approach is designed for program comprehension, which is the essential activity in bug finding. By studying the users performing these tasks, we can understand how our approach can support program comprehension both qualitatively and quantitatively.

A. Experiment design

We design our user study as a conventional within-subject controlled experiment, in which each participant solves four tasks, two with our tool and two without. We use a Latin-square design [37] with four groups, varying (1) which tasks are conducted using our tool and (2) task order (see Fig. 8). Compared to a between-subject design, our design gives us more statistical power for the same number of participants, is less influenced by individual differences between participants, and controls learning effects.

Tasks: We prepare two notebooks with two data wrangling bugs each (shared in our supplementary material). Both notebooks are modified versions of notebooks analyzed in Sec. V, and all bugs are based on real bugs we found. Tasks 1 and 2 in the first notebook correspond to incorrect data modelling (missing cases) and API misuse; Tasks 3 and 4 in the second notebook correspond to a typo in a regular expression and API misuse. Tasks 1 and 3 have similar symptoms (incorrect transformations for a subset of the data) and roughly have the same level of difficulty. Similarly, Tasks 2 and 4 have similar symptoms (code does not have a persistent effect) and roughly have the same level of difficulty. Participants are instructed to look for a bug in a specific subset of cells (distinct for each task) and report the location of the bug or show an example of incorrect data wrangling results. Participants can modify and run notebooks code to obtain any dynamic information and are free to search for information online.

Group 1: Task 1, 2 with tool then Task 3, 4 without tool	**Group 2:** Task 1, 2 without tool then Task 3, 4 with tool
Group 3: Task 3, 4 with tool then Task 1, 2 without tool	**Group 4:** Task 3, 4 without tool then Task 1, 2 with tool

Fig. 8: Latin square design varying treatment and task order.

We tested the difficulty of the tasks and the clarity of instructions in a pilot study with five participants, replacing one bug where participants questioned intended behavior and refining the wording of our instructions.

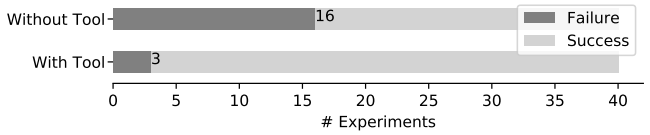
Participants: For our study, we recruited 20 participants. We estimated the number of needed participants conservatively based on the expected large effect size from pilot experiments.

We recruited students with data-science experience, who are common target users for notebook environments and likely share traits with early-career data scientists. We recruited through university-wide social media by advertising for a one-hour study for students with data-science experience at Peking University, compensating participants for their time.

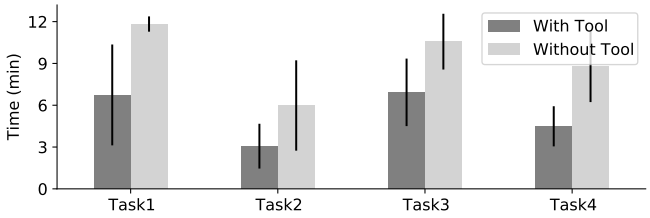
The 20 participants (15 male, 5 female; 18-24 years old) all have Python programming experience (from half a year to seven years; 18 with more than 2 years of experience). Over half of participants self-rated their familiarity with Jupyter notebook as 'familiar' or 'very familiar'. Most participants have a computer science background or study data science, while four learn or use data science for their domains (information systems, economics, geography, and physics). We randomly assigned participants to the four experimental groups. The four groups are comparable in terms of Python experience and Jupyter familiarity.

Procedure: After participants filled out a background survey to gather demographic information, we invited participants for a one-hour experiment session in the lab, one participant at a time. We provided a laptop with the notebooks prepared in a Chrome web browser.

We started the session by introducing the WrangleDoc extension with a tutorial. Through a dedicated notebook, we described the tool's features in text cells between code cells on which they could directly try it for about 10 minutes. After



(a) Number of bugs found and not found.



(b) Average completion time and standard variance per task.

Fig. 9: User study results.

the tutorial, we gave them a warm-up bug hunting task, to make sure they understand how to use WRANGLEDOC.

We then asked participants to work on the four tasks. Depending on their assigned experimental group, participants started either with tasks 1 and 2 or tasks 3 and 4 and either used WRANGLEDOC for the first two or the last two tasks. We enforced a time limit of 12 minutes per bug. When participants suggested an incorrect solution, we rejected it and asked them to continue searching until they either identified the bug or exhausted the time limit. Participants only continued to the next task once they finished or timed out on the previous one.

After completing all four tasks, we conducted an optional interview, where we asked participants about their experience with understanding data wrangling code and using our tool.

Analysis: We analyze both whether participants find the bugs within the time limit and how long it takes them if they do. We analyze completion times with an ANOVA analysis, testing to what degree our tool, the tasks, the order (tool first or tool last), and the participants' self-reported experience explain variance in completion times. We use the McFadden's pseudo-R^2 measure to evaluate the goodness-of-fit of our model. In Tab. 9, we report the exponentiated coefficient, standard error, p-value, and effect size (see column "LR Chisq" for the absolute amounts of deviance explained).

For tasks where participants did not find the bug within the time limit, we report the maximum time of 12 minutes, but our results are also robust to analyzing competition time variance only among tasks that were finished within the time limit.

B. Results

Overall, our experiment demonstrates that WRANGLEDOC could significantly improve participants' performance, as it helps the understanding process in general as well as expose unusual cases in a clear form. First, we find that participants are *much* more likely to find the bugs if they use our tool (37 out of 40) than if not (24 out of 40), as shown in Fig. 9b. Second, we find that participants complete the tasks on average 44% faster when using our tool (stat. sign., $p < 0.001$), as shown in Fig. 9b and Tab. III. Lastly, we find that all other factors (task order, Python experience, notebook familiarity) have little effect on how fast participants complete the tasks (see Tab. III).

TABLE III: User study result model ($R^2 = 33\%$).

	Coeffs (Errors)	LR Chisq
(Intercept)	$37013.17\ (1.55)^{***}$	
Interv.: Used WRANGLEDOC?	$0.02\ (0.68)^{***}$	34.53^{***}
Order: Tool in first notebook?	$1.47\ (0.68)$	0.32
Years of Python experience	$1.07\ (0.25)$	0.07
Jupyter experience (1–5)	$0.83\ (0.34)$	0.32
Task number	$0.69\ (0.30)$	1.48
$^{***}p < 0.001$		N = 80

Beyond the quantitative results, we observed that participants use similar strategies to understand notebook when they do the tasks without WRANGLEDOC as observed in Sec. II. We found that participants learned WRANGLEDOC quickly. In our post-experiment interviews, participants universally expressed that the tool was helpful, but differed in how WRANGLEDOC helped them: Some commented that synthesized summary helps their overall code understanding, e.g. *"Summary is very straightforward. Before I read the code, I just looked at the summary to get a rough idea of what the code does,"* but others emphasized the usefulness of table visualization and examples, e.g., *"Example rows are the most helpful part. It saves me lots of time of manually checking all possible cases for the data,"* and *"Visualization helps a lot. Everything I need to know (for a given cell) is placed on the left and in bold, and data changes are shown directly in one table. [...] When I was doing tasks without the tool, I have to scroll and compare data before and after, and this is really annoying."*

C. Threats to validity

Readers should be careful when generalizing our results beyond the studied participant population and tasks. The participants in our user study are not professional data scientists, but their background is comparable with early-carrier data scientists. While the tasks mirror many practical debugging and reuse settings that require understanding existing code, the bug finding experience for the participants in our study might be different from practical settings where data scientists are working with their own code, with different data, and without the artificial, time constrained setting of an experiment.

VII. RELATED WORK

Studies on computational notebooks: Researchers have conducted empirical studies to understand the practices and challenges of using computational notebook [3], [5]–[7], [11], [15], [16], [38]. For example, Pimentel et al. [3] studied the *reproducibility* issues of over one million notebooks and found that only 24% of the notebooks could be executed without exceptions, and only 4% produced the same results. Koenzen et al. [14] found that *code duplication* is common in notebooks and presented the needs for supporting code reuse in notebook

interfaces. Chattopadhyay et al. [5] studied the pain points for data scientists when working with notebooks, including large challenges due to poor tool support and scalability challenges. Wang et al. [17], [18] studied the challenges for collaborative editing using computational notebooks.

While some praise notebooks as a literate programming environment that provides opportunities for integrating documentation with code to publish code with a narrative and rationale [10], [39]–[42], large-scale empirical studies of notebooks have found that most public notebooks contain rather sparse documentation [3], [6], [7]. Such studies also highlight how most notebooks use only a small set of common libraries [3], [6], which makes analysis strategies like ours feasible that require static or dynamic analyses for commonly used functions.

Tooling support for computational notebooks: A large number of tools has been proposed and developed to help data scientists on various tasks, including fine-grained version control [11], [43], code cleaning through slicing [23], tracking provenance information [44]–[46], synthesizing data wrangling code from examples [47], facilitating collaborative exploratory programming using notebooks [18], and assuring reproducibility [48], among many others. Lau et al. [20] summarized the design space of notebooks covering different stages of a computational workflow and provided an excellent overview of tools for notebooks in academia and practice.

With regard to finding bugs, researchers have proposed the use of static analysis tools on data science code [22], [49] to look for certain patterns of mistakes. For several kinds of bugs we found, e.g., API misuse and not persisted changes, specialized analyzers could likely be developed.

Regarding documentation, JupyterLab can show API documentation in a tooltip on demand. Closest to our work, Wang et al. [50] developed three strategies for documentation generation in notebooks: neural-network-based automated code summarization, linking to external API documentation based on keywords in a cell, and prompting users to manually write documentation. Their work is intended to support users in writing markdown cells when sharing the notebook, rather than providing on-demand summaries of data transformations for program comprehension as in our approach; they do not rely on runtime information and do not consider the data processed.

Other related fields: Gulzar et al. [51]–[53] explored strategies to *test and debug big data applications* in Apache Sparks, which often have a similar flavor to the data wrangling code in notebooks. They analyze the executions of the running analysis, using row-based provenance tracking and symbolic execution to identify and generate inputs that crash the data wrangling code or violate user-provided test cases. In contrast, our summaries help surface wrong behavior with existing data that did not result in crashes or fail tests.

Outside of notebooks, various *code summarization* techniques have been developed to provide short summaries of code fragments automatically [9], for example for generating descriptive summaries for methods and classes, commit messages, or even "extreme summarization" as method names; they use various forms of information retrieval [54]–[56], code structure analysis [57], machine learning [50], [58]–[61], and natural language processing [62]. They usually either consider the code as individual tokens, a sequence of tokens, or use static analysis of the code to provide more effective inference of code properties. Since they do not analyze concrete executions dynamically, none of these methods can summarize the code's impact on specific data. We, on the other hand, specifically use code synthesis and test suite minimization techniques based on runtime information. Our tool can generate abstract patterns and examples derived from observing executions which is more accurate and relevant to the data wrangling task, but specific to a given dataset.

Program synthesis is a well-established field to generate programs from input-output samples [63], which has also been suggested to synthesize data science code from provided before-and-after samples of dataframes [47]. In our work, we use synthesis in an unusual way: We execute code to dynamically collect before and after values and then apply synthesis to those values but synthesize only general patterns for documentation rather than concrete code. Similarly, we apply *test suite minimization*, which has been broadly studied for removing redundancy from large test suites [30], [31], in an unusual way to select input rows for data science code. In both cases, we build on existing techniques but use them in novel contexts. We are not aware of other work that uses synthesis or test suite minimization in a similar context.

VIII. Conclusion

We propose to adapt program synthesis to generate summaries for data transformations performed in notebooks and use test suite minimization techniques to group rows by shared execution paths through data wrangling code. Both forms of documentation are based on observing a notebook's execution at runtime with concrete data, implemented in a JupyterLab extension WrangleDoc. Our evaluation found that our approach can generate correct documentation with acceptable runtime overhead and that the documentation helps data scientists find subtle bugs in data wrangling code.

Acknowledgment

We thank Anjiang Wei and Yiqian Wu for their feedback on this work and Jingjing Liang for her help with the user study. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2021-03538, the NSF award 1813598, and Fonds de recherche du Québec (FRQNT) 2021-NC-284820.

References

[1] G. Press, "Cleaning big data: Most time-consuming, least enjoyable data science task, survey says," *Forbes*, Mar 2016, https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/.

[2] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, "Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011, pp. 65–74.

[3] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter notebooks," in *Proc. Conf. Mining Software Repositories (MSR)*. IEEE Computer Society, 2019, pp. 507–517.

[4] A. Koenzen, N. A. Ernst, and M. Storey, "Code duplication and reuse in Jupyter notebooks," *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*, pp. 1–9, 2020.

[5] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *Proc. Conf. Human Factors in Computing Systems (CHI)*. ACM Press, 2020, p. 1–12.

[6] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino *et al.*, "Data science through the looking glass and what we found there," *arXiv preprint arXiv:1912.09536*, 2019.

[7] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proc. Conf. Human Factors in Computing Systems (CHI)*, 2018, pp. 1–12.

[8] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proc. Int'l Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, 2019, pp. 291–300.

[9] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," *arXiv preprint arXiv:1909.04352*, 2019.

[10] J. M. Perkel, "Why Jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

[11] 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 *Proc. Conf. Human Factors in Computing Systems (CHI)*, 2018, pp. 1–11.

[12] 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.

[13] K. Patel, J. Fogarty, J. A. Landay, and B. Harrison, "Investigating statistical machine learning as a tool for software development," in *Proc. Conf. Human Factors in Computing Systems (CHI)*, 2008, pp. 667–676.

[14] A. P. Koenzen, N. A. Ernst, and M.-A. D. Storey, "Code duplication and reuse in Jupyter notebooks," in *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE Computer Society, 2020, pp. 1–9.

[15] M. Bussonnier, J. Forde, J. Freeman, B. Granger, T. Head, C. Holdgraf, K. Kelley, G. Nalvarte, A. Osheroff *et al.*, "Binder 2.0-reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in Science Conference*, vol. 113, 2018, p. 120.

[16] J. Wang, K. Tzu-Yang, L. Li, and A. Zeller, "Assessing and restoring reproducibility of Jupyter notebooks," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, 2020, pp. 138–149.

[17] A. Y. Wang, A. Mittal, C. Brooks, and S. Oney, "How data scientists use computational notebooks for real-time collaboration," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–30, 2019.

[18] A. Y. Wang, Z. Wu, C. Brooks, and S. Oney, "Callisto: Capturing the" why" by connecting conversations with computational narratives," in *Proc. Conf. Human Factors in Computing Systems (CHI)*, 2020, pp. 1–13.

[19] K. Holtzblatt and H. Beyer, *Contextual design: defining customer-centered systems*. Morgan Kaufmann, 1997.

[20] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo, "The design space of computational notebooks: An analysis of 60 systems in academia and industry," in *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE Computer Society, 2020, pp. 1–11.

[21] "pandas-profiling," 2021. [Online]. Available: https://github.com/pandas-profiling/pandas-profiling

[22] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: On the need of analyzing Jupyter notebooks," in *Proc. Int'l Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM Press, 2020, p. 53–56.

[23] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Proc. Conf. Human Factors in Computing Systems (CHI)*, 2019, pp. 1–12.

[24] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, "What do developers ask about ml libraries? a large-scale study using stack overflow," *arXiv preprint arXiv:1906.11940*, 2019.

[25] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM Press, 2014, p. 530–541.

[26] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.

[27] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proc. Int'l Conf. Software Engineering (ICSE)*, vol. 1. IEEE Computer Society, 2015, pp. 134–144.

[28] S. Zhou, S. Stanciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2018, pp. 105–116.

[29] G. Zhang, M. A. Merrill, Y. Liu, J. Heer, and T. Althoff, "Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis," 2020.

[30] H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 419–429.

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[32] "Titanic - Machine Learning from Disasters." [Online]. Available: https://www.kaggle.com/c/titanic/code?competitionId=3136

[33] L. Gupta, "Google Play Store Apps," Feb 2019. [Online]. Available: https://www.kaggle.com/lava18/google-play-store-apps

[34] K. Gadiya, "FIFA 19 complete player dataset," Dec 2018. [Online]. Available: https://www.kaggle.com/karangadiya/fifa19/

[35] Dgomonov, "New York City Airbnb Open Data," Aug 2019. [Online]. Available: https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data

[36] J. J. Randolph, "Free-marginal multirater kappa (multirater k [free]): An alternative to fleiss' fixed-marginal multirater kappa." *Online submission*, 2005.

[37] G. E. P. Box, *Statistics for experimenters: design, innovation, and discovery*. Wiley-Blackwell, 2009.

[38] K. Subramanian, I. Zubarev, S. Völker, and J. Borchers, "Supporting data workers to perform exploratory programming," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–6.

[39] J. Somers, "The scientific paper is obsolete," *The Atlantic*, vol. 4, 2018.

[40] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez *et al.*, "Ten simple rules for writing and sharing computational analyses in Jupyter notebooks," 2019.

[41] J. Singer, "Notes on notebooks: Is Jupyter the bringer of jollity?" in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2020, pp. 180–186.

[42] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Proceedings of the 20th International Conference on Electronic Publishing*, vol. 2016, 2016.

[43] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE Computer Society, 2018, pp. 147–155.

[44] J. Hu, J. Joung, M. Jacobs, K. Z. Gajos, and M. I. Seltzer, "Improving data scientist efficiency with provenance," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2020, pp. 1086–1097.

[45] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noworkflow: A tool for collecting, analyzing, and managing provenance from python scripts," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, 2017.

[46] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, and Y. Wu, "Vamsa: Tracking provenance in data science scripts," *CoRR*, vol. abs/2001.01861, 2020.

[47] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists," in *Proc. Conf. Human Factors in Computing Systems (CHI)*. ACM Press, 2020, p. 1–12.

[48] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," 2021.
[49] N. Hynes, D. Sculley, and M. Terry, "The data linter: Lightweight, automated sanity checking for ml data sets," in *NIPS MLSys Workshop*, 2017.
[50] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, "Themisto: Towards automated documentation generation in computational notebooks," *arXiv preprint arXiv:2102.12592*, 2021.
[51] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, "Bigdebug: Debugging primitives for interactive big data processing in spark," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2016, pp. 784–795.
[52] M. A. Gulzar, S. Mardani, M. Musuvathi, and M. Kim, "White-box testing of big data analytics with complex user-defined functions," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 290–301.
[53] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, 2020, pp. 722–733.
[54] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 35–44.
[55] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "Autofolding for source code summarization," *IEEE Trans. Softw. Eng. (TSE)*, vol. 43, no. 12, pp. 1095–1109, 2017.
[56] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class api summaries," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 120–130.
[57] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE Computer Society, 2013, pp. 23–32.
[58] N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li, and Z. Ren, "Source code fragment summarization with small-scale crowdsourcing based features," *Frontiers of Computer Science*, vol. 10, no. 3, pp. 504–517, 2016.
[59] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE Computer Society, 2018, pp. 200–20 010.
[60] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.
[61] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proc. Int'l Conf. Program Comprehension (ICPC)*, 2020, pp. 184–195.
[62] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 2017, pp. 205–216.
[63] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.