**wranglesearch: Mining Data Wrangling Functions from Python Programs**

José Pablo Cambronero  
Massachusetts Institute of Technology  
Cambridge, MA, U.S.A.  
jcamsan@mit.edu

Raul Castro Fernandez  
The University of Chicago  
Chicago, IL, U.S.A.  
raulcf@uchicago.edu

Martin C. Rinard  
Massachusetts Institute of Technology  
Cambridge, MA, U.S.A.  
rinard@csail.mit.edu

**ABSTRACT**

Analysts spend a substantial amount of time wrangling (i.e., preparing) data for their analyses. We present wranglesearch, a system that automatically extracts reusable data wrangling functions from a corpus of existing Python programs written to analyze a particular dataset. A new analyst can query wranglesearch’s function database to obtain wrangling functions that they can integrate into their own analyses, leveraging the wrangling efforts of prior analysts.

**1 INTRODUCTION**

Data analysts invest large amounts of effort and time in writing dataset-specific wrangling operations such as normalization, subsetting, imputing, and type casting [16, 24, 30]. Wrangling operations are often implemented as ad-hoc Python snippets or notebooks handcrafted for specific datasets, developed iteratively, and stored and accessed separately from the dataset that they analyze [15, 22]. In tandem, widely used datasets, such as profit/loss sheets at investment banks or inventory ledgers at retailers, often play a central role in the operations of the organization that maintains them, with multiple analysts working with the same dataset over its lifetime and independently developing a substantial amount of data wrangling code.

**Mining Data Wrangling Functions.** We present wranglesearch, a new system for automatically extracting data preparation code from Python programs. wranglesearch enables new analysts to come to an existing dataset, quickly find, and integrate previously written data wrangling operations into their analyses. wranglesearch is therefore designed to mine wrangling functions for datasets where people have already written data analysis code, exploiting the availability of this source code to automatically extract reusable data preparation steps.

In addition to reducing the amount of effort required to analyze the dataset, wranglesearch can also help new analysts better understand the overall structure of the dataset, as relevant characteristics (e.g., key categorical values) are typically imbedded in wrangling code, promoting a more effective analysis of the dataset.

wranglesearch begins with a collection of existing scripts that analyze the same dataset, consisting of one or more relational tables. In an offline phase, wranglesearch executes an instrumented version of each script and collects a dynamic execution trace. The traces collected are used to build a dynamic data dependency graph for each program, from which wranglesearch can identify data preparation statements. wranglesearch then lifts these code snippets into modular function definitions, which are stored in a function database. In an online phase, developers can query this dataset-specific collection of functions by providing column names or function names as an input query. wranglesearch identifies possible results and ranks these based on a heuristic score, inducing diversity to avoid returning overly similar functions to the user.

**Results.** We carried out an online survey-based study to evaluate the effectiveness of wranglesearch’s mined functions. In the survey, we presented participants with three different data preparation tasks, along with five of wranglesearch’s mined functions. We found 90%–100% of participants found at least one mined function relevant to their data preparation task, and 76%–86% of participants at least somewhat agreed that access to wranglesearch would help in the data preparation tasks.

To evaluate wranglesearch’s generalizability, we applied wranglesearch to three datasets from different domains. wranglesearch successfully mined between 60 and 290 wrangling functions, of which 51% to 97% could be successfully executed without any modifications. The mined functions perform wrangling operations such as encoding numeric features based on threshold values, identifying sentinel missing string values and replacing them with a semantic NULL, and joining multiple tables with mismatched column names and subsetting the result. Additionally, we show that wranglesearch’s offline mining approach is practical, leading to a median overhead ranging between 61% and 77% when extracting wrangling functions compared to normal (i.e., uninstrumented) program execution.

**Example Usage.** Picture a new data analyst working with a widely used loan issuance dataset. Loan issuance data is critical to the lending activities of many major financial institutions, but despite this importance, reporting and structuring practices for loan records vary across institutions and data providers. This analyst has not worked with this specific dataset before but has been tasked with identifying the sum total of original issuance amounts associated with delinquent loans. Though they are unfamiliar with this dataset, they know that their group has deployed wranglesearch. In an offline phase, wranglesearch previously executed their colleagues’ programs written to analyze this same dataset, extracted statements that perform data preparation, and stored them as functions in a function database. The analyst can query this database to find relevant data preparation steps. To query wranglesearch, the analyst leverages wranglesearch’s lPython [27] “magic” command directly from their Jupyter [23] notebook.

```python
%load_ext wranglesearch.magic  
%query "loan_status" 2
```

1. All mined functions, including not executables ones, are accessible at https://github.com/josepablomca/wranglesearch/tree/master/analysis-results/executability-results.
2. delinquent loans are all loans not up to date on their corresponding payments.
The first command loads wranglesearch’s querying function, tquery into the IPython environment. The second command corresponds to a wranglesearch query, and is structured as tquery <query-terms > <number-results>, where query-terms is a collection of one or more terms that the user believes may overlap with wrangling operations of interest, and number-results is the maximum number of wrangling operations that the user would like to inspect. In this case, the analyst wants to wrangle the dataset column loan_status and so uses this as their single query term. They are also interested in inspecting at most two wrangling functions. wranglesearch searches for candidate wrangling functions in its database and computes a heuristic score for each function based on the number of overlapping terms in query-terms and each function’s definition, while accounting for function definition length. Next, if the number of candidate functions exceeds the user’s number-results, wranglesearch induces diversity in the final result set to avoid returning functions that are very similar to each other. To induce diversity, wranglesearch clusters candidate results based on a vector representation of their source definition and takes a limited number of candidates from each cluster [13]. wranglesearch can now return the top two candidate functions, which includes the function shown below, and embeds their source code definitions directly in the analyst’s notebook.

```python
def f1(df):
    # core cleaning code
    import pandas as pd
    # df = pd.read_csv('..\input\loan.csv', low_memory=False)
    df = df.loc[(df['loan_status'] != 'Current')]
    return df
```

The analyst can now use f1 to subset loans down to the population of interest for their analysis. As the analyst encounters additional wrangling steps in their work, they can run new wranglesearch queries and embed other function definitions, facilitating code discovery and re-use.

**Related Work.** The software engineering, programming languages, and human-computer interaction communities have developed techniques for searching, mining, and refactoring code. This body of work includes code search based on: natural language queries [9, 21], vector representations of code [4, 5, 21], and structural abstractions of code [7, 25]. Code repository mining has been used to identify recurring and re-usable patterns of code, e.g., idioms [2, 3]. These solutions are increasingly deployed [29] to automate development tasks, removing the need to manually search through forums such as StackOverflow [11] or large code repositories. However, existing programs do not link the resulting code fragments with the data they are intended to consume, which is needed to solve the data wrangling reusability problem. While some data analysis systems have linked code and data [15, 22], these have mostly focused on an individual analyst’s code, rather than serving as a source for re-usable code across different individuals working (potentially independently) with the same dataset.

The data management community has also explored code mining for data analysis. AutoType [35] automatically mines GitHub repositories to identify re-usable code fragments for semantic type validation. TDE [14] synthesizes input transformation programs by using input/output examples to search over a large collection of mined transformations and compose results. Transform-by-Pattern [20] replaces the concrete input/output examples used in TDE with input/output patterns, a generalization for settings where concrete examples are harder to produce. VAMSA [26] mines Python scripts to statically extract the sources and attributes used to train a specific model found in source code.

wranglesearch takes a complementary approach to this body of existing work. In particular, wranglesearch’s code searches are specific to a particular dataset. The resulting wrangling functions were mined from a collection of programs written to analyze the same dataset. To mine these functions, wranglesearch uses dynamic analysis to identify relevant data preparation statements in existing programs, modularize them into standalone functions, and store them in a function database. To search for wrangling functions, wranglesearch users do not provide input/output examples, but rather use term-based queries. The resulting wrangling functions’ definitions can then be integrated into a data analyst’s computational notebook automatically.

**Why the Scalable Data Science Track?** A major challenge to scalability in data science is the need to manually prepare datasets. Solutions that facilitate this time consuming task can help scale data analyses and increase analyst productivity. We believe that existing source code provides a rich source of metadata that can help with problems ranging from data wrangling re-usability (the focus of this paper), to discovery, cleaning, visualization, and modeling.

**Contributions.** This paper presents the following main contributions:

- Techniques to mine, organize, and retrieve wrangling functions from existing programs
- An implementation of these techniques in a system we name wranglesearch
- A survey-based evaluation of the effectiveness of our mined wrangling functions, as well as experiments on the generalizability and practicality of using wranglesearch

### 2 WRANGLESEARCH

Figure 1 presents an overview of wranglesearch, separated into an offline and online phase. During an offline phase, wranglesearch takes a collection of Python scripts along with their tabular input data, executes the scripts, mines the resulting execution traces, and lifts extracted code snippets into functions that are stored in a function database. This function database captures the relationship of the mined functions to columns in the original dataset and other functions. During an online phase, wranglesearch serves queries using the function database, returning executable code snippets that perform data wrangling and are related to the input query.

Before discussing the key definitions for wrangling functions, our approach to mining them from execution traces, and the querying of our function database, we motivate our decision to specialize our tool to Python scripts.

**Why Python?** wranglesearch is designed specifically to mine code from Python scripts. Python remains one of the most popular tools for data analysis. Psallidas et al.’s [28] analysis of 5.1 million computational notebooks on Github (as of July 2019) found that 92%
Figure 1: wranglesearch’s two-phase design: an offline phase is used to mine wrangling code snippets and an online phase serves queries by returning relevant functions mined.

were written in Python. Similarly, Xin et al [34] surveyed users of automated machine learning systems [19] and found that the Python libraries Pandas and NumPy were the most popular data wrangling tools. Additionally, Python’s reference CPython implementation provides built-in tracing functionality (through sys.settrace) that can be used to easily implement the necessary dynamic analysis. A similar implementation targeting other popular languages may be possible. For example, R has popular data wrangling libraries such as dplyr [32] and data.table [10], and Goel and Vitek [12] recently developed and released a dynamic analysis pipeline for R scripts.

2.1 Defining Wrangling Code

First we provide three key definitions that wranglesearch uses to narrow the scope of code snippets it mines.

**Definition 1 (wrangling statement).** A wrangling statement is defined as an assignment statement of the form \( t.s = e \) or \( \{ s' \} = e \), where \( t \) is a program variable of type `Table`\(^4\), \( s \) is an identifier, \( s' \) is a string literal, and \( e \) is an expression in the target Python program.

A wrangling statement is focused on identifying the creation of new columns or overwriting of existing column values. These statements capture typical wrangling operations such as datatype casting, value binning, and imputation.

**Definition 2 (wrangling trace).** Let \( \mathcal{P} \) be a sequencing operator. Let \( p_1 \) be the \( i \)-th program statement in a sequence, where \( p_i \) may depend on previous program statements or on a disk read. Let \( w \) be a wrangling statement. Then a wrangling trace is defined as a sequence \( \langle p_1, ..., p_n, w \rangle \) corresponding to a linearization of a data dependency graph rooted at \( w \).

A wrangling trace includes all the transitive dependencies of a wrangling statement. Ordered execution of these dependencies (including disk reads of input data) results in the outputs defined by the original source code.

**Definition 3 (wrangling function).** A wrangling function is a Python function of type `Table`\(^4\) → `Table`, which takes as input at least one table and returns a single table. The body of the wrangling function correspond to a subset of statements in a wrangling trace.

A single wrangling trace may (implicitly) produce multiple outputs of interest. For example, a trace that first subsets a table and then defines a new column has two intermediate tables that may be useful to an analyst. As a result, wranglesearch generates one or more wrangling functions per wrangling trace, making each intermediate table a possible return value.

If we call a wrangling function with new parameters, we obtain new wrangled outputs. This means that each wrangling function can be used by an analyst to prepare their dataset, where the new version of the dataset has been derived using existing code written for their dataset.

2.2 Mining Wrangling Code

wranglesearch uses dynamic analysis [6] – the process of executing a program and monitoring its state and operations – to identify code that satisfies our wrangling definitions. The online phase in Figure 1 presents an overview of this analysis pipeline. First, wranglesearch instruments the corpus of Python scripts and executes them on their input dataset. wranglesearch collects a program trace, from which it constructs a dynamic data dependency graph for statements in the original program. wranglesearch mines this graph to identify wrangling traces that capture the steps associated with wrangling statements. wranglesearch then extracts one or more wrangling functions from each such trace. These wrangling functions are then stored in a function database. We now discuss each step in detail.

2.2.1 Program Instrumentation and Execution. wranglesearch’s rewrits input scripts to facilitate tracing. In particular, wranglesearch lifts nested expressions, adds stub functions that record runtime information such as memory locations and type information, and adds book-keeping code. To avoid impacting the semantics of the original program, wranglesearch does not rewrite certain Python constructs, specifically: lambdas, comprehensions, generator expressions, yield expressions, and f-strings.

To reduce the size of traces collected, wranglesearch only traces code written or directly invoked by the user. This eliminates long internal call chains, which do not reflect meaningful information about the input dataset. wranglesearch also reduces overhead by tracing execution only for a bounded (and pre-defined) number of iterations for each loop body. wranglesearch’s instrumentation, which relies on CPython’s built-in tracing support (sys.settrace), logs events at the line level. wranglesearch does not record lines corresponding to condition expression in control-flow statements. This decision simplified our instrumentation, and is based on the observation that data analysis code is often based on sequential composition of operators, with complex control-flow modularized into those operators, an observation supported by recent large-scale analysis of data analysis programs [28].

After executing the programs, wranglesearch uses the recorded execution traces to build a data dependency graph. Each node in the graph corresponds to a single line executed in the rewritten source code. A directed edge is placed from node \( N_i \) to node \( N_j \), if the statement at \( N_j \) uses data that is defined or modified in the statement at \( N_i \). Each node is annotated with additional information such as the type of any objects used and any calls made.

Algorithm 1 details how wranglesearch extracts a wrangling trace from the dynamic data dependency graph. Line 3 traverses the graph to identify assignment nodes that constitute a wrangling statement (Definition 1). These nodes form the end points of our
wrangling traces. To extract the trace, wranglesearch performs a backwards traversal through the data dependency graph using each wrangling statement as a starting point (lines 5–7), a technique known as program slicing [1]. We term the slice obtained a wrangling trace (Definition 2). This trace may be missing statements as a result of our limited program instrumentation. In particular, wranglesearch does not rely on an external knowledgebase detailing the effects of third-party libraries. For example, given a third-party function \( f \), a table \( t \), and a column name \( c \), wranglesearch does not know if the call \( f(t, \text{col}="c") \) potentially modifies column \( c \). Consider a library that re-uses the original array allocated for \( t \ c \), the only way to determine if a modification had taken place would be to perform some form of comparison over the column values, such as a hash, or have external knowledge about the effects of \( f \). wranglesearch eschews this and instead extends the wrangling trace to also include earlier program statements that assign or use a column name that appears in our trace, even if memory addresses do not indicate a direct link in our dependency graph.

**Algorithm 1** Extracting wrangling traces from a data dependency graph

**INPUT:** \( g \) is a data dependency graph; \( \text{IsWranglingStmt} \) is a predicate that returns true if a statement satisfies Definition 1; \( \text{REACHABLE} \) returns the set of nodes reachable in a graph when we reverse edges and start a depth-first-search from a given node; \( \text{SUBGRAPH} \) projects a graph down to a subset of nodes.

**OUTPUT:** A collection of wrangling traces

1. function \( \text{EXTRACTWRANGLINGTRACES}(g) \)
2. \> Set of wrangling statements
3. \( \text{seeds} \leftarrow \{ \text{node} | \text{node} \in g, \text{IsWranglingStmt}(\text{node.stmt}) \} \)
4. \> Set of sets of reachable (dependency) nodes
5. \( W_{\text{nodes}} \leftarrow \{ \text{REACHABLE}(g, s) | s \in \text{seeds} \} \)
6. \> Projection down to produce a set of wrangling traces
7. return \( \{ \text{SUBGRAPH}(g, \text{nodes}) | \text{nodes} \in W_{\text{nodes}} \} \)
8. end function

2.2.2 Extracting Functions. Algorithm 2 details how wranglesearch lifts each wrangling trace to one or more wrangling functions. wranglesearch may lift more than one wrangling function from a single trace to cover the possibility of multiple intermediate tables of interest to the user. By definition, a wrangling function (Definition 3) must have at least one table input parameter and return a single table. To satisfy this definition, wranglesearch removes any statements that execute a disk read from the wrangling traces and collects any free variables of type Table that result from this rewrite (line 2). These free variables constitute the input parameters of the wrangling function. To produce a single return value, we identify all variables of type Table that are modified or defined within the wrangling trace (line 3). For each such statement, wranglesearch creates a copy of the wrangling function with a new return value, and defines the body as the trace statements necessary for the given return value (lines 5–9).

**Algorithm 2** Lifting a wrangling trace to a wrangling function

**INPUT:** \( t \) a wrangling trace; \( \text{FREE} \) returns free variables in a statement after removing disk reads; \( \text{FINALMOD} \) returns true if a variable is modified or created within the trace and this is its last occurrence in the trace; \( \text{REACHABLE} \) returns the set of nodes reachable in a graph when we reverse edges and start a depth-first-search from a given node; \( \text{SUBGRAPH} \) projects a graph down to a subset of nodes.

**OUTPUT:** A set of wrangling functions

1. function \( \text{LIFTTOWRANGLINGFUNCTION}(t) \)
2. \( \text{inputs} \leftarrow \{ \text{node} | \text{node} \in t, \text{FREE}(t, \text{stmt}) \land \text{Type}(\text{node}) = \text{TABLE} \} \)
3. \( \text{returns} \leftarrow \{ \text{node} | \text{node} \in t, \text{node} \in \text{FINALMOD}(t, \text{stmt}) \land \text{Type}(\text{node}) = \text{TABLE} \} \)
4. \( \text{funs} \leftarrow 0 \)
5. for \( r \in \text{returns} \) do
6. \( \text{uses} = \text{inputs}, r \)
7. \> Dependencies for this return statement and prune dead code
8. \( \text{fbody} \leftarrow (\text{stmt})(\text{node} \in \text{SUBGRAPH}(g_r, \text{REACHABLE}(g_r, r))) \)
9. \( \text{funs} \leftarrow \text{funs} \cup f \)
10. end for
11. return funs
12. end function

2.3 Querying a Function Database

We expect users to retrieve wrangling functions based on dataset column names and third-party functions and integrate the resulting functions into their own code. For example, a user may want to identify wrangling functions that use column “date”, and then use these functions in their notebook. To enable this interaction, we implemented an IPython “magic” command for wranglesearch and a generalized query routine that takes column names and function names as query terms. This command retrieves the top results and inline their source code definitions in the user’s notebook. To support a more granular interaction, where for example the user may want to distinguish between functions that use a column versus define a column, we employ typed relations between wrangling functions, columns, and third-party functions. We now describe in detail the storage, querying, and ranking of wrangling functions.

2.3.1 Storage. The use of typed relations makes a graph data model a natural storage model for wranglesearch’s wrangling functions. wranglesearch’s graph model defines three node types and four relation types that can characterize mined snippets.

Nodes of type Column represent columns in the dataset used by the mined scripts. Nodes of type WranglingFunction represent wrangling functions mined by wranglesearch. Nodes of type Function represent all other functions called within the wrangling functions themselves (e.g. third-party library functions).

Let \( f \) be a wrangling function. Let \( \text{USES}(f) \), \( \text{DEFINES}(f) \), and \( \text{CALLS}(f) \) be the set of columns read, columns assigned to, and other functions called by \( f \) during execution in the original source code, respectively. Let \( \text{WRANGLESFOR}(f) \) be the set of function calls that are reachable in the original source code by traversing the data dependency graph starting from the value returned by \( f \), i.e. these are calls that depend on \( f \)’s return value. wranglesearch adds a corresponding typed edge between the wrangling function \( f \) and each element in the result of the sets described. We implement these relations using the popular graph database neo4j [31], but they could have also been implemented using a traditional relational database.

2.3.2 Querying. The online phase in Figure 1 provides a high-level overview of querying. wranglesearch first retrieves all possible results based on existing relations in the function database. wranglesearch then uses a heuristic to rank results with respect to the
input query. Given that multiple wrangling functions may be similar to each other, and the user may have requested a small result set, wranglesearch may diversify the set of results before returning to the user. We now describe these steps in detail.

### Fetching candidate results
wranglesearch uses typed relations in its function database to fetch all candidate wrangling functions. For each term in the input query, wranglesearch branches to different typed relations depending on whether the term is a function object or a column name. If the term is a function object, wranglesearch extracts the function’s qualified name (as library path, if any, and function name) and queries using relations CALLS and WRANGLES FOR relations. If the term is a string, wranglesearch assumes this is a column name, and queries using relations USES and DEFINES.

#### Ranking results
Algorithm 3 presents the details of wranglesearch heuristic ranking of candidate results. First, wranglesearch counts the number of lines of code in the body of the wrangling function, as well as the number of tokens that overlap with the input query (lines 3–7). The lines of code and the overlap are then normalized. To normalize lines of code, we divide the minimum lines of code in the set of candidate results by the lines of code in each corresponding result. Similarly, we normalize term overlap by dividing the maximum term overlap by each function’s term overlap. The two normalized terms are then combined in a convex sum producing a heuristic score for relevance (lines 8–14). Intuitively, a wrangling function that is the shortest and has the highest term overlap will score the highest in the set. wranglesearch removes any wrangling functions with a zero score (line 16), and then sorts the remaining candidates in descending order of heuristic score (line 17). Heuristic ranking based on term overlap has been shown to work in other code search systems [29].

**Algorithm 3** Ranking query results using a heuristic shortness and term overlap heuristic score

**INPUT:** `results` is an unordered set of wrangling functions; `Q` is the query used to retrieve this set of functions; a user-determined weight `α` to compute a convex sum of heuristic score terms (defaults to 0.5).

**OUTPUT:** Collection of wrangling functions sorted in descending order of their heuristic score

1. `function Rank(results, Q, α)`
   2. `M_loc ← {}; M_overlap ← {}` /* Maps for LoC and token overlap
   3. for `f ∈ results` do
   4.     `M_loc[f] ← LinesOfCode(f.body)`
   5.     `tokens ← Tokenize(f.body)`
   6.     `M_overlap[f] ← Count(Q∩tokens)`
   7.   end for
   8.   `▷ Normalise and compute heuristic score`
   9.   `l_{min} ← Min(M_loc.Values), o_{max} ← Max(M_overlap.Values)`
   10.  `M_{score} ← {}`
   11.  for `f ∈ results` do
   12.      `score ← \frac{l_{min} + α * M_{overlap}[f]}{o_{max} + (1-α)}`
   13.      `M_{score}[f] ← score`
   14.   end for
   15.   `▷ Zero score results are pruned out`
   16.   `results ← \{f | f ∈ results \land M_{score}[f] > 0\}`
   17.   `results ← SortBy(results, M_{score})`
   18. end function

#### Diversifying results
After computing heuristic scores, wranglesearch (like other information retrieval systems [36]) may need to diversify the set of results, if these exceed the limit set by the user. Algorithm 4 presents the details of wranglesearch’s diversification. If the set of resulting functions does not exceed the number of results implied by the product of the requested number of clusters and entries per cluster, wranglesearch can just return the current collection of functions (lines 2–4). If diversification is needed, wranglesearch relies on clustering wrangling functions based on their source code (lines 6–8). First wranglesearch computes the weighted vector representation of each function as the TF-IDF transform of tokens in the function definition body. Then wranglesearch computes the correlation matrix over the TF-IDF representation, and clamps values to zero. Given this matrix, wranglesearch then creates `K` clusters using spectral clustering, and sorts the `n` most diverse functions from each of the `K` clusters (lines 10–15).

**Algorithm 4** When there are more query results than the target number, wranglesearch uses a clustering-based approach to diversify the final set of results.

**INPUT:** `results` is a collection of wrangling functions in descending order of their heuristic score; a user-determined number of candidate clusters `K` (defaults to 5); a user-determined number of results per cluster `n` (defaults to 3).

**OUTPUT:** Sequence of at most `K ∗ n` wrangling functions sorted in descending order of their heuristic score

1. `function Diversify(results, K, n)`
2.   if `Count(results) ≤ K ∗ n` then
3.       return results /* No need to diversify
4.   end if
5.   `▷ Spectral clustering based on TF-IDF of function bodies`
6.   `results_corr ← Corr(TFIDF(results))`
7.   `results_corr ← Max(results_corr, 0)` /* Element-wise clamping
8.   `clusters ← SpectralCluster(K, results_corr)`
9.   `Counter ← 1; results_pruned ← {}`
10.  for `f ∈ results` do
11.     if `Counter[clusters[f]] < n` then
12.        `results_pruned ← results_pruned ∪ f`
13.        `Counter[clusters[f]] += 1`
14.     end if
15.   end for
16.   return results_pruned
17. end function

### 3 Evaluation
To evaluate wranglesearch we focus on two key research questions (RQ). Can wranglesearch successfully mine wrangling functions (RQ1)? Is it practical to apply wranglesearch to mine wrangling functions from collections of Python scripts (RQ2)?

#### RQ1: Can wranglesearch successfully mine wrangling functions?
To address RQ1 we designed a user study, which we carried out using an online survey.5 The survey evaluates the relevance, ranking, and utility of wrangling functions for three data preparation tasks.

5https://mit.co1.qualtrics.com/jfe/form/SV_6X4UtWnVvR5P4k45
3.1.1 Methodology. We now present our survey design.

**Participants.** The ideal participant for our survey is a software developer or data analyst with experience wrangling datasets using Python and the Pandas library in particular. To recruit a sufficient number of such participants, we hosted our survey on the online participant recruitment website Prolific. 5 We used Prolific’s filters to approximate our ideal participant. We required the participants to have computer programming experience and a university degree (bachelors or higher), the latter as proxy for experience with data analysis. To ensure participants had sufficient Python and Pandas knowledge we incorporated a validation question in our survey. We excluded those who failed to answer correctly.

We received 83 survey responses, of which 33 successfully answered our validation question. Of these 33 participants, we identified 21 that completed their survey without inconsistencies in their responses. We analyze and present the results associated with these 21 participants. Most participants had at least one year of programming experience (7 of the 21 had 5 or more years of experience) and one year of data analysis experience. Most participants had experience with at least one data analysis tool, including Python, Matlab, and SQL.

**Dataset and Tasks.** The participant is asked to picture themselves as an employee at a large data analytics company, where they perform data wrangling tasks. The participant is given the description of a loan issuance dataset, which contains 75 different columns (such as loan amount) for 800,000 loans. The survey then presents three different data preparation tasks, each accompanied by a short description of the kind of analyses that might require such a preparation step. Table 1 presents the three tasks. Task 1 consists of identifying the subset of loans in the dataset that are not current. Task 2 consists of rounding the interest rate column to the nearest integer. Task 3 consists of computing the month and year associated with the issuance date for each loan. Participants are then presented with five wrangling functions for each preparation task.

**Survey Questions.** After a participant has read through a data preparation task, they are presented with a set of wrangling functions mined by wranglesearch. After reviewing the functions, the participant is asked a series of questions (Table 2). First we ask participants if the data preparation task is reflective of those they perform in their own data analysis. We term this a task relevance question and use it to validate that our survey tasks are realistic and match real user needs. We then ask participants to mark the functions that they believe are relevant to the data preparation task described. We term this a function relevance question and use it to evaluate wranglesearch retrieval at a coarse level. Next we ask the participant to select the function that they consider to be most relevant to the task. We term this a function ranking question, designed to evaluate granular ranking of functions. Finally, we ask participants if having access to these snippets would make completing the data preparation task easier. We term this a function utility question, designed to determine if wranglesearch is helpful.

**Within-Subjects Design.** The survey is designed to have two arms, corresponding to two different versions of each set of wrangling functions, enabling a within-subjects analysis. For each set of wrangling questions, the ws arm presents the top five wrangling functions as ranked by wranglesearch. The ws-r random arm presents a set of five wrangling functions sampled uniformly at random from the functions mined by wranglesearch and presented in a fixed random order. Each participant is exposed to the two arms for a given data preparation task in sequence, before moving on to the next data preparation task. To mitigate the possible impact of task ordering, and learning effects for arm ordering, we randomize the order of tasks and the order of arms within each task.

**Statistical Analysis.** We use Wilcoxon Signed Rank Test (WSRT) [33] over participant’s paired observations to determine statistical significance. We chose a non-parametric test as our data varies between ordinal and cardinal. We set a significance level of 0.05 and perform Bonferroni adjustment [8] for repeated testing.

3.1.2 Results. We now present the results of our survey, organized based on the specific question presented.

**Task relevance.** Figure 2 presents the distribution of answers to our task relevance question. We confirmed that our data preparation tasks were reflective of real data analyses. Between 66.7% (tasks 1 and 2) and 71.4% (task 3) of participants at least somewhat agreed that the tasks presented were reflective of their own data preparation steps.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Task Description</th>
<th>wranglesearch Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identify non-current loans based on loan_status column</td>
<td>Query(&quot;loan_status&quot;)</td>
</tr>
<tr>
<td>2</td>
<td>Round interest rate (int_rate) column to nearest integer</td>
<td>Query(&quot;int_rate&quot;, pandas.DataFrame.astype)</td>
</tr>
<tr>
<td>3</td>
<td>Compute issuance month/year based on issuance date (issue_d) column</td>
<td>Query(&quot;issue_month&quot;, pandas.to_datetime)</td>
</tr>
</tbody>
</table>

Table 1: Data preparation tasks for the online survey, as well as wranglesearch queries used to retrieve the top five wrangling functions presented to participants in one arm of the study.

<table>
<thead>
<tr>
<th>Question Name</th>
<th>Question</th>
<th>Answer Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task relevance</td>
<td>The task is reflective of tasks you might perform during your own data analyses.</td>
<td>7-point Likert</td>
</tr>
<tr>
<td>Function relevance</td>
<td>Which fragment snippets do you consider relevant to the task? Leave blank if none.</td>
<td>Multiple choice</td>
</tr>
<tr>
<td>Function ranking</td>
<td>Which snippet, if any, do you consider to be most relevant to the task?</td>
<td>Single choice</td>
</tr>
<tr>
<td>Function utility</td>
<td>Access to these snippets makes completing the task easier.</td>
<td>7-point Likert</td>
</tr>
</tbody>
</table>

Table 2: Questions presented in the online survey. Questions prefixed with “Function” are presented for each set of wrangling functions, while the “Task relevance” question is presented once per data preparation task. We assign question names for clarity in our subsequent analyses.
Function relevance. We analyzed the number of relevant wrangling functions produced by \textsc{wranglesearch}. Figure 3 shows the number of functions marked as relevant by participants for each task. We find that all participants found at least one function to be relevant, in both \textsc{ws} and \textsc{ws-random} arms, for task 1. For tasks 2 and 3, 100% and 90% of participants, respectively, found at least one relevant wrangling function when retrieval was performed using \textsc{wranglesearch}’s ranking procedure (\textsc{ws} arm). In contrast, only 57% and 62% of participants found at least one relevant function for tasks 2 and 3, respectively, when randomized retrieval was performed (\textsc{ws-random} arm). We find that the difference in number of relevant code snippets between survey arms is statistically significant for tasks 2 and 3. For these two tasks, \textsc{wranglesearch}’s retrieval method provides a clear advantage over randomized retrieval as it retrieves a larger number of relevant functions. For task 1, two of \textsc{ws-random}’s functions include the predicate "\text{'loan_status'" != \text{'Current'}) and one function includes a type casting of the column 'loan_status', which may explain why users found a comparable distribution of relevant functions in both arms.

Function ranking. We analyzed the effectiveness of the ranking procedure at a granular level. Figure 4 shows the distribution of the rank of the snippet participants regarded to be most relevant to each task. We find that our ranking produces a statistically significant improvement in task 2, with an average \textsc{ws} rank of 0.86 compared to an average \textsc{ws-random} rank of 3.33, but not in the remaining tasks. So while \textsc{wranglesearch}’s ranking procedure appears useful for retrieving relevant functions in the top 5, it does not seem to provide granular enough ranking.

Function utility. We evaluated the extent to which \textsc{wranglesearch} would help an analyst complete the data preparation task at hand. We find that our ranking produces a statistically significant difference in rank distribution between the arms in task 2. Functions are ordered from F0 to F4. A paired WSRT shows a statistically significant difference between the distribution of responses between arms for task 3.

Discussion. Wrangling functions mined by \textsc{wranglesearch}, presented in both the \textsc{ws} and \textsc{ws-random} arms, are judged by participants as helpful for performing data preparation tasks. We find that the ranking used by \textsc{wranglesearch} retrieves more relevant wrangling functions than a \textsc{ws-random} approach using randomized retrieval. However, the simplicity of \textsc{wranglesearch}’s ranking makes it less effective at a total ordering of results, and thus in practice the user may wish to retrieve the top set of \textsc{wranglesearch} results and then perform manual inspection prior to integrating into their workflow.

3.2 RQ2: Is it practical to apply \textsc{wranglesearch}?

We evaluate \textsc{wranglesearch}’s ability to mine functions from different datasets, the extent to which these functions can be executed, and the associated mining overhead.

Generalizing and Executability. We collected three datasets from the data science website Kaggle, along with the Python programs associated with each dataset. \texttt{loan\_data\_2018}, corresponds to a loan issuance dataset, which we used originally for our survey.
Table 3: wranglesearch can be applied to different datasets and script corpora, mining between 60 and 290 wrangling functions, depending on the dataset, of which between 54 and 165 are executable without modifications.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Available Scripts</th>
<th>Executable Scripts</th>
<th>Mined Functions</th>
<th>Executable Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_data_2018</td>
<td>91</td>
<td>58</td>
<td>290</td>
<td>147 (51%)</td>
</tr>
<tr>
<td>loan_data_2021</td>
<td>91</td>
<td>38</td>
<td>68</td>
<td>54 (79%)</td>
</tr>
<tr>
<td>house_sales</td>
<td>119</td>
<td>52</td>
<td>60</td>
<td>58 (97%)</td>
</tr>
<tr>
<td>university_rankings</td>
<td>86</td>
<td>69</td>
<td>180</td>
<td>165 (92%)</td>
</tr>
</tbody>
</table>

Table 3 shows the number of scripts executed in the spring of 2018. As a result, scripts that executed for loan_data_2018 may not execute for loan_data_2021.

Discussion. While wranglesearch was not developed with the goal of minimizing mining overhead, the median overhead ranges between 61% and 77%. More improvements may be readily available through focused rewrites of our implementation. A challenge to executing real code is running it in the correct environment. As libraries evolve, older code may be increasingly hard to execute. While wranglesearch mines between 68 and 180 functions for programs executed with 2021 packages, existing techniques could improve this further. wranglesearch could help mitigate this issue by attempting to execute a script with varying versions of a library, an approach similar to that of DockerizeMe [18]. Similarly, increasing the program corpus size can lead to further improvements in the number of functions mined.

4 CONCLUSION

We introduced and evaluated wranglesearch, a system that automatically extracts data wrangling functions from a collection of Python scripts written to analyze the same dataset. These functions are stored in a function database that can be queried, enabling discovery and re-use of wrangling code.

REFERENCES


Ignacio Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. Data wrangling: The challenging journey from the wild to the lake. In CIDR, 2015.


