# DataRinse: Semantic Transforms for Data preparation based on Code Mining

Ibrahim Abdelaziz
IBM Research
Yorktown Heights, New York
ibrahim.abdelaziz1@ibm.com

Julian Dolby
IBM Research
Yorktown Heights, New York
dolby@us.ibm.com

Udayan Khurana
IBM Research
Yorktown Heights, New York
ukhurana@us.ibm.com

Horst Samulowitz
IBM Research
Yorktown Heights, New York
samulowitz@us.ibm.com

Kavitha Srinivas
IBM Research
Yorktown Heights, New York
kavitha.srinivas@ibm.com

## ABSTRACT

Data preparation is a crucial first step to any data analysis problem. This task is largely manual, performed by a person familiar with the data domain. *DataRinse* is a system designed to extract relevant transforms from large scale static analysis of repositories of code. Our motivation is that in any large enterprise, multiple personas such as data engineers and data scientists work on similar datasets. However, sharing or re-using that code is not obvious and difficult to execute. In this paper, we demonstrate DataRinse to handle data preparation, such that the system recommends code designed to help with the preparation of a column for data analysis more generally. We show that DataRinse does not simply shard expressions observed in code but also uses analysis to group expressions applied to the same field such that related transforms appear coherently to a user. It is a human-in-the-loop system where the users select relevant code snippets produced by DataRinse to apply on their dataset.

**PVLDB Reference Format:**
Ibrahim Abdelaziz, Julian Dolby, Udayan Khurana, Horst Samulowitz, and Kavitha Srinivas. DataRinse: Semantic Transforms for Data preparation based on Code Mining. PVLDB, 16(12): 4090 - 4093, 2023.
doi:10.14778/3611540.3611628

## 1 INTRODUCTION

Enterprises often have many data analysts performing modeling and exploration on the same or similar types of datasets. One of the most time consuming parts of any data analysis step is the preparation of data, such as encoding categorical values, handling ordinals, imputation of missing values, data aggregations, amongst others. This step often requires a significant amount of domain knowledge, and is largely carried out manually by a data engineer, data scientist, analyst or a domain expert. The knowledge, once applied, is often buried in code, and this code tends to be stored in enterprise repositories. Listing 1 shows an example of such a transform, where the data scientist is trying to bin `Age` so that it can be combined with the passenger class attribute to make a combined inference about the wealth and age of the passenger, which likely impacted survival on the Titanic. This snippet of code is adapted from a much larger script that performs multiple operations on the titanic dataset[1]. A key goal of DataRinse is to allow a data scientist re-use of code relevant to the dataset without them having to peruse all the code that manipulates the same dataset.

In DataRinse, we perform this by applying static code analysis on Python scripts. We adopt the path of static analysis compared to dynamic analysis because very often the datasets associated with code in repositories are not available readily making dynamic analysis impractical. It works by finding the transformations to fields of a dataframe, which are then extracted, and converted into an intermediate representation, and this representation is traversed to generate code to help the user transform their dataframe. A key goal of DataRinse is to behave as a recommendation system rather than automating data preparation in one shot. Code generation for data preparation is particularly challenging because: (a) the order in which specific transformations are performed is important; (b) code generation has to be sensitive to the 'groups' of transformations that constitute a single operation conceptually. We illustrate both points with a running example. Listing 1 shows a code snippet from an actual script for preparing the titanic dataset for exploratory analysis. Note that when code is generated, it needs to be aware that a number of transforms are being performed as a 'group', so all the contiguous `Age` transforms bin `Age` into buckets. Furthermore, the transformation of `Age * Pclass` is dependent on that binning operation being performed first; that is, there is a dependency between the first function and the second.

DataRinse performs both grouping of conceptually related operations into functions and encodes dependencies between several functions to offer users functions that can be easily incorporated into their own notebooks. We use control flow from static analysis to perform grouping, and we use control and data flow to understand dependencies between the functions that are generated in DataRinse. Furthermore, to make this consumable to the end user, we display a drop down box in the notebook, so that they can choose which field they want to view cleansing functions for. For each function that is extracted, we also show the URL of the


Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611628

[1] https://github.com/davified/clean-code-ml/blob/master/notebooks/titanic-original.ipynb

script that was used to generate that function. Listing 4 shows the output of DataRinse of a code snippet for the Age column, where a single function captures all binning changes to Age, and a second separate function shows the creation of a new column using it. The user has a number of such functions per column that can be readily applied to their dataset if they deem it applicable, and the ordering of transformations is encapsulated in the code generated such that a call to transform Age is performed before the use of it in expressions such as `Age * Pclass`, as shown. All the generated functions are dynamically evaluated on the dataframe to ensure that any functions that are not compile-able are filtered out. Similarly, the generated functions that do not result in a change in the values of the dataframe are also eliminated.

We note that ChatBots such as ChatGPT[2] can appear to provide similar functionality for such tasks. There are however key differences between those systems and DataRinse. While ChatGPT generates some useful data operations, it has the following characteristics that are different from DataRinse: (a) It has non-determinism in its generation, it can sometimes find cleansing operations, but produces different results on each run; a well-known problem for ChatBots (see Listings 2, 3 for two different code snippets produced for the same dataset (b) It does not follow basic data operation rules such as being consistent across train and test, and it is not comprehensive - `Fare` is handled only in one snippet. (c) There is no attribution to the source that these are drawn from, (d) It is not focused - the snippets were drawn from a much larger script, where the system produced imports, read CSV files, etc. (e) It can only work off its training dataset - DataRinse can for instance run over any scripts in any repository at any desired frequency (e.g., on every commit or daily) requiring negligible computational resources; ChatBots have to be retrained on enterprise specific code.

## 2 SYSTEM OVERVIEW

Figure 1 describes the overall DataRinse system for data preparation. Given a new dataset, the system queries code repositories such as GitHub with the table's metadata, specifically the table and column names. One current limitation of DataRinse is that it will only select scripts that match column names and table names - this is a limitation we plan to address in future work through more flexible column-to-concept mappers [7]. The analysis framework is an open source project that we created[3] which performs inter-procedural static analysis to create control flow and data flow graphs for Python scripts [1]. The scope of the analysis is the entire notebook, so dataflow and control flow can be used to order the cleansing operations. We perform interprocedural analysis to trace both data and control flow, as illustrated in Figure 2. The chalky lines for `read_df` and `prep_df` denote function calls, and dataflow follows them, e.g `"titantic_train.csv"` flows to `fl_name`. After analysis, the code generation component for cleansing starts by isolating reads, writes and updates to pandas dataframe. It uses dataflow information to determine which expressions are relevant, e.g. dataflow connects the read of a pandas dataframe at the top of Figure 2 to the update of `dataset` at the bottom.

[2]https://openai.com/blog/chatgpt
[3]https://github.com/wala/graph4code/tree/master/semForms

```python
import pandas as pd

def read_df(fl_name):
  return pd.read_csv(fl_name)

def prep_df(combine):
    for dataset in combine:
        dataset.loc[ dataset['Age'] <= 16, 'Age'] = 0
        dataset.loc[(dataset['Age'] > 16) &
        ↪  (dataset['Age'] <= 32), 'Age'] = 1
        dataset.loc[(dataset['Age'] > 32) &
        ↪  (dataset['Age'] <= 48), 'Age'] = 2
        dataset.loc[(dataset['Age'] > 48) &
        ↪  (dataset['Age'] <= 64), 'Age'] = 3

def main():
    train_df = read_df("titanic_train.csv")
    test_df =  read_df("titanic_test.csv")
    combine = [train_df,test_df]
    prep_df(combine)
    for dataset in combine:
        dataset['Age*Class'] = dataset.Age *
        ↪  dataset.Pclass
```

**Listing 1: Example data preparation code, handling the titanic dataset**

```python
train_df = train_df.drop(['PassengerId', 'Name',
↪  'Ticket', 'Cabin'], axis=1)
train_df['Age'].fillna( train_df['Age'].median(),
↪  inplace=True)
train_df['Embarked'].fillna(
↪  train_df['Embarked'].mode()[0], inplace=True)
train_df['FamilySize'] = train_df['SibSp'] +
↪  train_df['Parch'] + 1
train_df['IsAlone'] = 1
train_df['IsAlone'].loc[train_df['FamilySize'] > 1]
↪  = 0
train_df['Sex'] = train_df['Sex'].map({'female': 0,
↪  'male': 1}).astype(int)
```

**Listing 2: Code snippets from ChatGPT, when prompted to prepare the titanic dataset for analysis, first attempt**

The code generation component assembles related code into multiple functions, organized by the fields that they change, as well as records any dependencies in the application of the functions by encoding calls to relevant functions. In Pandas, calls on a dataframe can sometimes mutate the dataframe (e.g. `drop()`), and sometimes not (e.g. `isnull()`). Because there is no way of knowing if the values of a dataframe are actually modified by a call, the filtering component dynamically evaluates the functions returned by the code generation module in the client notebook to filter out functions that do not transform the data frame. Extracted functions across

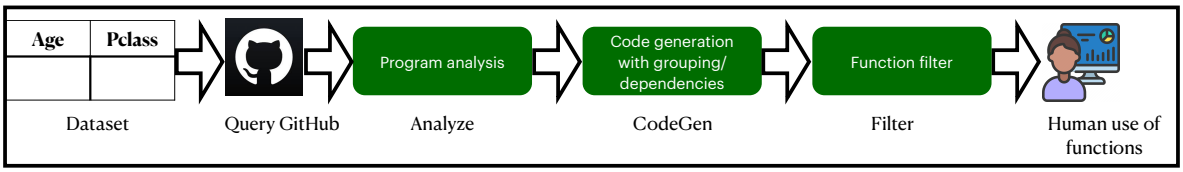


**Figure 1: Overview of DataRinse**

```
train_df.drop(['PassengerId', 'Name', 'Ticket',
  ↪  'Cabin'], axis=1, inplace=True)
test_df.drop(['Name', 'Ticket', 'Cabin'], axis=1,
  ↪  inplace=True)
train_df['Age'].fillna( train_df['Age'].median(),
  ↪  inplace=True)
test_df['Age'].fillna(test_df['Age'].median(),
  ↪  inplace=True)
test_df['Fare'].fillna( test_df['Fare'].median(),
  ↪  inplace=True)
```

**Listing 3: Code snippets from ChatGPT, when prompted to prepare the titanic dataset for analysis, second attempt**

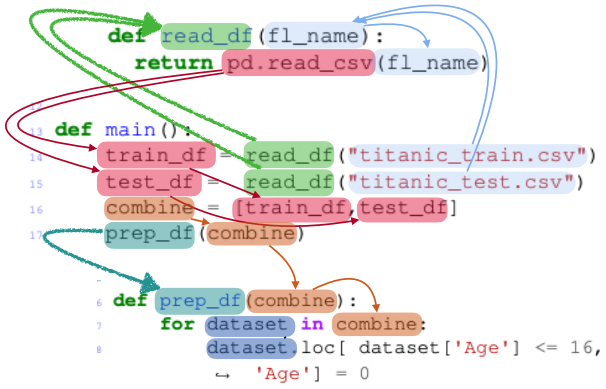


**Figure 2: Analysis on listing 1**

multiple scripts are then grouped by field, and each function for a given field encapsulates the extraction from a single script. The user can choose which functions they would like to look at, and explore what they would like to add to their own code through an interactive interface.

## 3 DEMONSTRATION

We show in Listing 4 an example of how DataRinse recommends data transforms is used for the example code shown in 1, which is a simplified version of the original code in[4], for the titanic dataset[5]. The UI is structured as a Jupyter notebook where the user loads

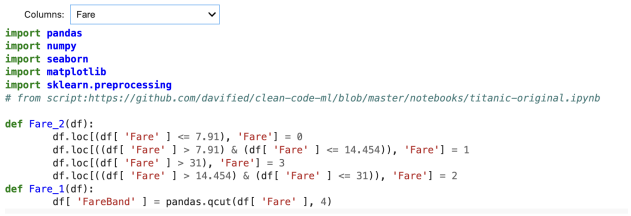


**Figure 3: Display of expressions by columns**

a dataset. The dataset name and columns are fed to GitHub to retrieve scripts that may be cleansing the same dataset. The scripts are analyzed, and code generation is performed for each field in that is manipulated in the scripts. Any dependencies in manipulations are captured by the code generation as well, as shown in Listing 4. A second example is shown in Listing 5 for the UCI heart disease prediction dataset[6]. Notice here that the order of operations in the original script[7] are maintained, which involves setting missing values in the resting ECG results to `normal` before using a `LabelEncoder` to encode all the values.

In general, there may be too many data manipulation operations across scripts. Since the generated code relies on an IR, obvious duplicates are eliminated at creation time across scripts. Scripts are also ordered by the ones that have the most number of functions across fields, because those are the most comprehensive. The code suggestions are organized by columns using a dropdown as shown in Figure 3. At the conference, we will allow the audience to select an available dataset, and explore the code segments provided by the system. We will also work with the audience to use a dataset of their choice.

## 4 EXPERIMENTS

To assess the generality of DataRinse as a tool, we applied it to 1,589 Python notebooks from GitHub that were converted into Python scripts for analysis. Table 1 shows some statistics of running DataRinse over these scripts. Approximately 11% (182) of the 1,589 scripts had failures because the code could not be parsed or was not actually Python code; so that left us with 1407 scripts. Analysis ran successfully on most of those scripts (1,406); but after removal of duplicate scripts, we had 1,142 scripts. Of the 1,142 scripts that were successfully analyzed, 55% yielded at least one cleansing function. The total number of CSV files mentioned in these scripts from which cleansing functions were extracted was 2,262, but we note that this is likely an overestimate since many scripts load train and test splits

[4]https://github.com/davified/clean-code-ml/blob/master/notebooks/titanic-original.ipynb
[5]https://www.kaggle.com/competitions/titanic/data

[6]https://www.kaggle.com/datasets/redwankarimsony/heart-disease-data
[7]https://www.kaggle.com/code/achintyak/decision-tree

```
def Age_01(df):
    df.loc[(df[ 'Age' ] <= 16), 'Age'] = 0
    df.loc[((df[ 'Age' ] > 16) and (df[ 'Age' ] <=
    ↪  32)), 'Age'] = 1
    df.loc[((df[ 'Age' ] > 32) and (df[ 'Age' ] <=
    ↪  48)), 'Age'] = 2
    df.loc[((df[ 'Age' ] > 48) and (df[ 'Age' ] <=
    ↪  64)), 'Age'] = 3

def all_df(df):
    Age_01(df)
    df['Age*Class'] = df['Age'] * df['Pclass']
```

**Listing 4: Partial Output of DataRinse for the script shown in the running example**

```
from sklearn.preprocessing import LabelEncoder

def restecg_0(df):
        df[ 'restecg' ] = df[ 'restecg'
        ↪  ].fillna('normal')

def restecg_1(df):
        df[ 'restecg_n' ] =
        ↪  LabelEncoder().fit_transform(df[
        ↪  'restecg' ])
```

**Listing 5: Partial Output of DataRinse the UCI heart dataset**

**Table 1: Statistics of running DataRinse on 1,589 GitHub scripts**

| Category | Count |
|---|---|
| Total number of scripts | 1,589 |
| Number of scripts that passed analysis | 1,407 |
| Number of scripts that failed due to front end errors | 182 |
| Number of distinct scripts that passed analysis | 1,406 |
| Number of scripts with extracted functions | 630 |
| Number of CSV files mentioned in scripts | 2,262 |
| Number of cleansing functions extracted (pre-filtering) | 4,688 |

for the same CSV file. The total number of cleansing functions extracted was 4,660, prior to any filtering. Filtering requires access to the datasets being manipulated in the script, so we do not have the actual post-filtering results yet. This will be a focus for future work.

## 5 RELATED WORK

Suggestions for cleansing have mostly been based on dynamic analysis based approaches; e.g. Auto-suggest [10] uses heuristics to handle hard issues such as missing data files and packages by searching to actually run notebooks. Auto-suggest depends on the availability of the CSV file in the public domain, and on dynamic analysis which is difficult to perform on a large number of scripts efficiently mostly because software versions change, and most notebooks are not shipped with environments. Vizsmith [2] similarly uses 1260 Kaggle notebooks and their associated CSV files to mine 7K visualization functions using dynamic analysis, as does wranglesearch [3]. Our work can mine functions statically, without the need for executing a notebook and is hence much more scalable.

Other systems follow variants of programming by example (PBE) to create cleansing functions [4, 6, 8, 10]. Transform Data By Example (TDE) indexes over 50K functions from GitHub, and uses it to perform data transformations given user examples [5]. A more recent work examines whether large language models such as GPT-3 can solve problems from [5], given prompts that include a couple of examples [9]. DataRinse does not require any sort of user specified examples - in fact, for the sorts of problems considered in this work, it is unclear that the user would even know what the input and output pairs should be - if they did, they would know how to code it. We note that most of the PBE work has targeted spreadsheet users where users frequently perform the same operation over cells of a column, transforming or extracting bits of strings.

## 6 CONCLUSION

In this work, we introduced DataRinse, an approach for mining transforms from existing repositories to re-use prior work performed by other data scientists on similar datasets. Our demo highlights how the system when given an input dataset searches GitHub for relevant scripts and generates code for cleansing functions which are recommended to the user.

## REFERENCES


[1] Ibrahim Abdelaziz, Julian Dolby, James P McCusker, and Kavitha Srinivas. 2021. A Toolkit for Generating Code Knowledge Graphs. *The Eleventh International Conference on Knowledge Capture (K-CAP)* (2021).

[2] Rohan Bavishi, Shadaj Laddad, Hiroaki Yoshida, Mukul R Prasad, and Koushik Sen. 2021. VizSmith: Automated Visualization Synthesis by Mining Data-Science Notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 129–141.

[3] José Pablo Cambronero, Raul Castro Fernandez, and Martin C Rinard. 2022. wranglesearch: Mining Data Wrangling Functions from Python Programs. https://www.josecambronero.com/publication/wranglesearch/wranglesearch/. [Online; accessed 31-May-2022].

[4] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1165–1177.

[5] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

[6] Zhongjun Jin, Michael R Anderson, Michael Cafarella, and HV Jagadish. 2017. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 683–698.

[7] Udayan Khurana and Sainyam Galhotra. 2021. Semantic Concept Annotation for Tabular Data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 844–853.

[8] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[9] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *arXiv preprint arXiv:2205.09911* (2022).

[10] Cong Yan and Yeye He. 2020. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1539–1554.