DOI: 10.1111/cgf.14529
Eurographics Conference on Visualization (EuroVis) 2022
R. Borgo, G. E. Marai, and T. Schreck
(Guest Editors)

Volume 41 (2022), Number 3

Leveraging Analysis History for Improved In Situ Visualization Recommendation

Will Epperson[1], Doris Jung-Lin Lee[2], Leijie Wang[3], Kunal Agarwal[2], Aditya G. Parameswaran[2], Dominik Moritz[1], Adam Perer[1]

[1]Carnegie Mellon University, Pittsburgh, PA, USA
[2]UC Berkeley, Berkeley, CA, USA
[3]Tsinghua University, Beijing, China

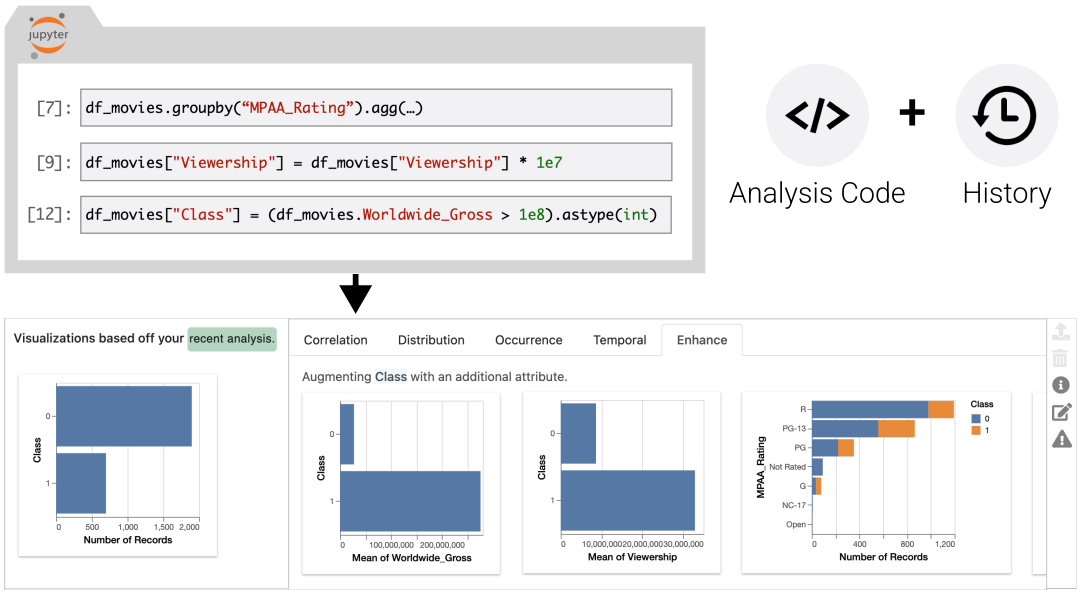

Figure 1: *Solas tracks the history of a user's analysis to provide improved in situ visualization recommendations. Above, a user has most recently created the* `Class` *column that is visualized on the left side of the interface. Recently executed Pandas commands interacted with* `Worldwide_Gross`, `Viewership`, *and* `MPAA_Rating`*; therefore,* `Class` *is shown relative to these columns.*

Abstract

Existing visualization recommendation systems commonly rely on a single snapshot of a dataset to suggest visualizations to users. However, exploratory data analysis involves a series of related interactions with a dataset over time rather than one-off analytical steps. We present Solas, a tool that tracks the history of a user's data analysis, models their interest in each column, and uses this information to provide visualization recommendations, all within the user's native analytical environment. Recommending with analysis history improves visualizations in three primary ways: task-specific visualizations use the provenance of data to provide sensible encodings for common analysis functions, aggregated history is used to rank visualizations by our model of a user's interest in each column, and column data types are inferred based on applied operations. We present a usage scenario and a user evaluation demonstrating how leveraging analysis history improves in situ visualization recommendations on real-world analysis tasks.

CCS Concepts

• ***Human-centered computing*** → ***Visualization; Visualization systems and tools;***

1 Introduction

During exploratory data analysis, analysts iteratively explore different methods for cleaning, aggregating, and filtering to make sense of their data [KHM17]. Throughout this exploration, recommended data visualizations can help users understand their analysis and determine next steps by automatically visualizing interesting relationships. However, most existing visualization recommendation systems provide recommendations on a *single* data snapshot that fails to capture the dynamic nature of analysis.

Recognizing this limitation, recent visualization recommendation libraries have started providing dynamically chosen recommendations **in situ** during the iterative process of data analysis [80820, PP, sd, Fbd, ada, LTA*22]. For example, the Lux library generates visualization recommendations when users display Pandas dataframes in a computational notebook. This in situ, dynamic approach to visualization recommendation has seen considerable adoption, and helps analysts identify valuable next steps in analysis [LTA*22]. However, existing libraries use a single snapshot of the dataset and/or the last data analysis or transformation step issued by the user, as opposed to the rich **history** of the user's exploration across steps. This history captures not just implicit user interest, but also provides cues into the underlying data semantics.

In this paper, we show how analysis history can improve recommendation. With analysis history, we can better understand both the *provenance* of data as users apply iterative transformations and which parts of the data users are likely interested in visualizing. For example, if a new column is created, a natural next step may be to visualize the distribution of this column. If an analyst has also recently explored other columns, we can suggest visualizations with these sets of columns to facilitate comparison. Over time, we build a model of a user's interest in each column of their data. Each time a user interacts with their data we update this model to reflect their current interests and use this information to improve visualization recommendations. By tracking history, we can also preserve data that is no longer in the current dataframe for visualization. For instance, when a filter is applied to data, we plot the unfiltered distribution alongside the filtered distribution to add context. Systems without the provenance provided by analysis history have no knowledge of how to combine these dataset iterations. Additionally, the operations a user applies to each column provide a hint about the high-level measurement types of the columns. For example, when a multiplication or division operation is applied, a column can be inferred as quantitative. As we track analysis history, we use these operational signals to do better type inference and visualize columns according to this inferred type.

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

We address the aforementioned challenges by integrating history tracking into a visualization recommendation tool, *Solas*. Our tool demonstrates how the extra information from analysis history leads to more insightful and better visualization recommendations. We provide a user evaluation of our task-specific recommendations that demonstrates that Python users find our provided visualizations useful for understanding the returned data from analysis functions. In summary, our contributions are as follows:

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

2 Background and Related Work

Two areas of interest are relevant to this work—interacting with analysis history and prior work on visualization recommendation.

2.1 Interacting with Analysis History

Provenance is often used to describe the history of how data and analysis evolve over time. Data provenance is used to understand analysis history, adapt to user preferences, and suggest next steps in a variety of analysis settings [XOW*20]. Graphical histories offer an approach for exploring the analysis history logged from a user's UI interactions with the visualization tool Tableau [HMSA08]. B2 logs an analyst's interactions with data as code snippets in a notebook so they can more easily recreate an analysis [WHS20]. Analysis history is also useful in experimentation and versioning so that analysts can track multiple versions of an analysis, switch between versions, and interact with these histories during exploratory programming [KHM17, KM18, KRA*18, WDBD21]. *Solas* offers a unique method for tracking analysis history through the code executed and suggesting visualizations from this history.

Aggregated past analyses can also be used to suggest potential next steps. Data scraped from Jupyter notebooks on GitHub has been used to suggest possible function parameters during analysis or to suggest next steps based on a user's current exploration path [YH20, RCK*21]; similar approaches have been used to recommend related SQL queries during analysis [AGG*15]. Data scraped from public repositories only represents a single snapshot of analysis and thus does not contain the full history of the user's exploration; *Solas* captures more detailed data interactions.

2.2 Visualization Recommendation

Visualization recommendation typically has two goals: (1) helping analysts follow best practices by creating visualizations that are both expressive and effective, and (2) removing the tedium of crafting visualizations to make the exploration process faster and more robust [Hee19]. These goals manifest in systems that recommend

a combination of *design* and *data* variations [WMA*16a]. Design variation shows data in a variety of visual encoding to a user to allow them to select the best way to visualize their data; data variation shows different combinations and subsets of the data to help users find interesting trends or patterns.

2.2.1 Recommending Design Variation

One of the early systems to focus on visualizing *design* variation was Mackinlay's APT system [Mac86]. APT recommended visualizations that satisfied the competing criterion of *expressiveness* (conveying the truth) and *effectiveness* (is the truth readily perceived). Later work also focused on presenting a variety of design encodings to an analyst that satisfy constraints on design best practices [MWN*19], or based on user-specified interest [MHS07]. Most similar to our work is Behavior-Driven Visualization Recommendation (BDVR) which matches a user's patterns of analysis to suggest alternative visual encodings [GW09]. Our work is distinct in several ways, namely that we track a user's analysis history through their code rather than direct manipulation and thus impose fewer restrictions on user inputs. By situating *Solas* in the Jupyter ecosystem, we provide visualizations when the alternative is no visualization at all, whereas all exploration in BDVR is visual.

2.2.2 Recommending Data Variation

Another complementary approach to visualization recommendation focuses on recommending data variation to the user. Foresight allows users to explore by selecting a guidepost metric of particular interest (such as high correlation) and then view charts with similar statistics [DHPP17]. In Zenvisage, users specify a query by sketching the general chart pattern they are looking for, such as a sharply increasing linear curve, and are presented with charts that loosely match this pattern [SKL*16].

The Voyager and Voyager 2 recommendation systems are driven by the maxim to "show data variation not design variation" [WMA*16b, WQM*17]. In these systems, a user specifies an attribute of interest, and the system shows visualizations of this attribute with one other attribute (possibly a wildcard) ranked by perceptual effectiveness scores. Furthermore, the CompassQL recommendation engine underlying these systems supports the partial specifications of visualizations that can fill in reasonable defaults according to best practices [WMA*16a]. Similarly, SeeDB allows a user to specify a base data query and the system finds interesting visualizations by comparing statistics between charts such as the skew or correlation of the data [VRM*15]. Despite their focus on data exploration, data variation systems notably all focus on a single iteration of a dataset as input. However, during their exploration, analysts are transforming, adding, and deleting data. By taking into account the provenance of data, *Solas* uses the history of analysis to provide improved recommendations.

Solas is an extension of the popular Lux library that recommends visualizations for Python Pandas dataframes [LTA*22]. Lux allows users to center recommendations around a particular attribute or subset of the data through a manually specified *intent*. However, in initial studies of the *Lux* system, users seldom used the intent specifications and found the in-place, immediate recommendations that *Lux* provides to be most helpful [LTA*22]. By tracking analysis history, *Solas* is able to automatically infer user intent and recommend appropriate visualizations. History tracking, task-specific visualizations afforded by history, and operational type inference are all unique to *Solas*. *Solas* groups recommended visualizations into the same semantic tabs as Lux such as Correlation, Distribution or Occurrence but sorts the charts in each of these tabs by the model of user column interest. The history tracking capabilities of *Solas* are not tied to Lux and can also be applied to other systems.

3 Tracking Analysis History

Our system design brings analysis tracking to users' native data analysis environments so they can use their normal data exploration tool stack. *Solas* tracks history for the popular Python data manipulation library Pandas and presents visualizations directly within Jupyter notebooks. Pandas is the most popular data manipulation library in Python, with over 300 million downloads as of 2021 [Pan]. Likewise, computational notebooks in Jupyter have become the tool of choice for data science in Python [Per18]. Due to their widespread adoption, *Solas* focuses on analysis history tracking and visualization in this ecosystem. Users can explore their data using Pandas and *Solas* automatically creates visualizations based on their analysis history.

3.1 Logging Python Pandas Function Calls

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

For each operation, we collect four pieces of information: the dataframe this operation occurred on, the data columns in the operation, the type of operation, and the time (in terms of execution count) when this operation occurred. In Jupyter, code is organized into cells that can be executed in arbitrary order. The output of the chunk of code in a cell is shown immediately below the cell. Whenever a cell is executed, the execution count increases by one and thus we use execution count as a time ordering of analysis commands. When `df["Medal"].value_counts()` is run, *Solas* logs that this operation occurred on the dataframe `df`, referenced the `Medal` column, was a value counts operation, and occurred at a certain execution count during the analysis. We discuss how we use this information for improved visualizations in Section 4.

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

Beyond a single operation, analysis history represents a *sequence* of operations over time. Many analysis steps return new dataframes or change an existing dataframe. Figure 2 demonstrates how a

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

Table 1: *Common analysis tasks that have accompanying task-specific visualizations in Solas. When an operation is performed, it is added to the history of that dataframe.*

filtering operation on `df_movies` at time step 11 returns a new dataframe that is assigned to the variable `filt_df`. By tracking history, we know that `df_movies` is the parent and `filt_df` the child. When an operation returns a new dataframe, this new object inherits the history from its parent. However, subsequent operations only affect either the parent or child, but not both. For example, the column assignment at execution count 12 in Figure 2 only affects the interest model of `df_movies` and the mean calculation in time step 14 only affects `filt_df`. By tracking this data provenance, we maintain references to data that would have been lost otherwise and can create unique visualizations that use the data before and after an operation is applied such as showing the background distribution for filtered data.

3.1.1 *Solas* Tracks Common Pandas Analysis API Calls

To ensure coverage of commonly used Pandas functions, we scanned the API documentation of Pandas and identified common analysis functions that might be applied to a dataframe or series. We additionally observed over 10 hours of online Pandas tutorials and analysis demonstration videos that showed how people use the API for real-world analysis tasks. Overall, our tracked analysis functions cover the most common *analysis* functions from previous investigations of Pandas API usage [PMX*20]. These operations range from simple variable selections to complex filters, aggregations, and statistical functions. Operations for which we provide a task-specific visualization are presented in Table 1. We also track history for additional functions like `df.head()` or `df.tail()`. However, since these functions interact with all columns in a dataframe, they do not provide us with additional information to model user interest in specific columns. We do not track history for table joins or operations that span multiple data tables since we focus our recommendations on visualizing one dataframe (and its history) at a time. Furthermore, joins result in a single dataframe object that can be visualized.

3.2 Modeling Column Interest

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

To accomplish this time-weighting in *Solas*, we use Jupyter's execution count as a time index for each history item. Every time a user executes a code cell, this execution count increases by one. Each operation begins with an initial weight w_0, that corresponds to how much we value this operation in our history. Most operations begin with $w_0 = 1$, with the exception of two operations. Column references begin with $w_0 = 0.5$ and column assignments with $w_0 = 2$. We found that column references are extremely common and happen in almost every single piece of analysis code. Therefore, we begin column references with a weight of $w_0 = 0.5$ to reflect this weaker signal. Likewise, column assignments are rarer and thus should be strongly valued. This is similar to the logic of TF-IDF from natural language processing, where more common words across documents are less interesting [Jon72]. Since column references happen more frequently, they provide us with less signal about a user's interest.

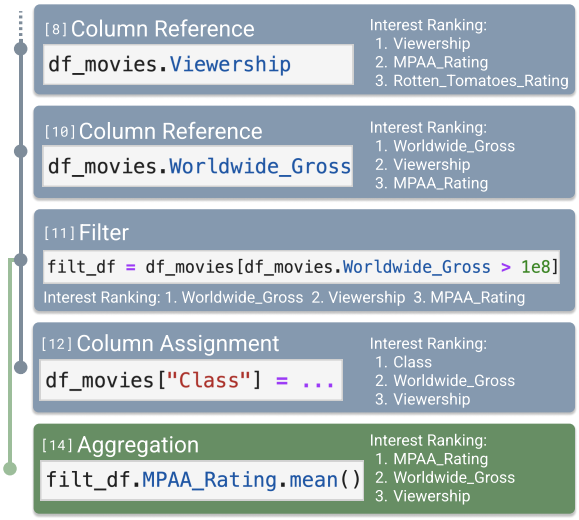

Figure 2: *The interest rankings demonstrate the model of column interest at different steps during the analysis in Section 5. When `Viewership` is referenced in time step 8, it has more interest than other columns from earlier in the analysis. After the filter in `[11]`, `filt_df` inherits the history from `df_movies` and further commands affect the models of each dataframe independently.*

To calculate our model of user interest in each column at a timestep t, we begin by iterating through the items in the dataframe's history in reverse order and decay the weights according to an expo-

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

Solas uses the model of column interest to visualize columns in the most recent operation relative to columns of interest in the enhance tab and to sort other recommendation tabs. The column interest ranking shown at each time step in Figure 2 demonstrates how this history aggregation works in practice. In time step 14 (the green box), the column interest model for `filt_df` ranks `MPAA_Rating` most highly since it was referenced most recently, and the interest in `Worldwide_Gross` has been decayed.

4 Visualization Recommendations from Analysis History

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

4.1 Improved Task Visualization

The most recent operation a user has applied to their data gives us the strongest signal about their current interest. As described in Section 3.2, the last operation gets the most weight in our model of a user's column interest. We also provide task-specific visualizations catered to the most recent operation in the *Solas* UI. To encode a single operation, we provide visualizations that, in our opinion, best communicate the task that the operation performs. Each of the functions in Table 1 is encoded in a specific way that best presents the task this function aims to accomplish. We chose the task-specific encodings to reflect both common practice (e.g. heatmaps for correlation matrices) as well as encodings that follow best practices such as those synthesized in prior work [MWN*19].

Our task-specific visualizations fall into two broad categories: those that detect pre-aggregated data and those that use historical

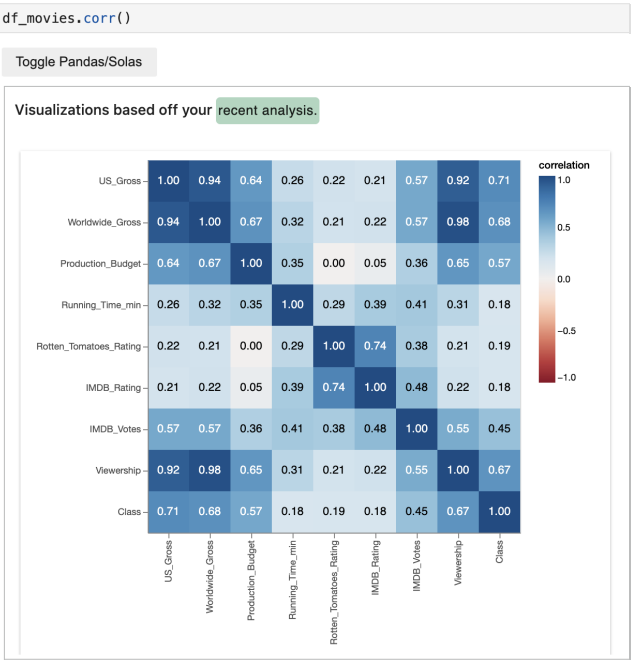

Figure 3: *By using analysis history, Solas better understands the semantics of data. It knows the values returned from `df.corr()` represent a correlation matrix and visualizes this data as a heatmap to highlight columns with high or low correlation.*

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

4.1.1 Detecting Pre-aggregated Data

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

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

Correlation. Calls to `df.corr()` return a correlation matrix. *Solas* plots this data as a heatmap over correlations to make it easier to spot columns with low or high correlation. Figure 3 shows a Correlation matrix visualization for the Movies dataset discussed in Section 5.

Null counts. There are several ways to check how many nulls are in a column in Pandas including `isna`, `isnull`, and `notnull`. Each of these functions returns a Boolean dataframe representing

if a value is null. We visualize this data as stacked bar charts showing how many nulls are in each column to help analysts identify columns with many (or few) null values.

4.1.2 Encoding Historical Data

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

Describe. The describe function returns statistical summaries of each quantitative column in a dataframe such as the count, mean, std, and quartiles. Since the goal of this function is to get an overview of the column, we visualize the data in a boxplot to communicate the distribution of the columns. The power of *Solas* becomes evident here as the data returned by `df.describe()` does not contain enough information to visualize a boxplot. Instead, *Solas* retrieves the parent dataframe (`df`) in order to plot outliers in the data needed for the boxplot. Furthermore, the data returned from `df.describe()`, like many analysis operations, is pre-aggregated. Visualizing these aggregates as raw values by treating them as quantitative values results in nonsensical visualizations. Yet without the history captured by *Solas*, a recommendation system would be unaware this data is pre-aggregated. Figure 5A shows our boxplot visualization of this function. Without analysis history, visualizing outliers is not possible.

Filters. During data analysis, filtering is extremely common. In Pandas, users can accomplish the same filtering task with the following three commands: `df[df.Age > 30]`, `df.loc[df.Age > 30]`, `df.query("Age > 30")`. To better understand a filter, it can be useful to plot how the returned data compares to the original distributions. Filtering on one column can sometimes have unexpected effects on the distribution of other columns. In *Solas*, we support this comparison between the filtered and original distribution by visualizing these two distributions as overlapping bars or histograms for each column. Users can toggle the background distribution on or off to support focus on only the filtered data or to compare to the background. Additionally, we sort the returned charts to prioritize distributions that shift the most after the filter is applied by calculating the earth mover's distance between the two distributions in the same approach as SeeDB [VRM*15]. Sorting in this way allows users to compare which distributions shift the most because of the filter. Figure 5D shows an example filter visualization for `df_movies[df_movies.Worldwide_Gross > 1e8]`. The first visualization shows how many points remain in the data after the filter is performed. The following visualizations show that the distribution of `Worldwide_Gross` shifted the most after filtering, followed by `US_Gross`, `Production_Budget`, and so on.

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

Operation	Inferred level
$=, \neq$	Nominal
$<, \leq, >, \geq$	Ordinal
$+, -$	Interval
$*, /, //, \%, **$	Ratio

Aggregation	Inferred level
min, max, median	Ordinal
mean, sum	Interval
prod, std, var, sem, skew	Ratio

Table 2: *When column operations and aggregations are applied to the data, the measurement level type is updated if the new level is more restrictive.*

For any groupby aggregation, we also update the x-axis name to include the aggregation so that users know how their data was aggregated in the plot. Figure 5C shows an example groupby where `Rotten_Tomatoes_Rating` is aggregated by its mean. *Solas* shows error bars for these groups, allowing users to understand the standard deviation in addition to the mean, without having to write any extra code.

4.2 Using Column Interest Model

Solas uses the model of column interest described in Section 3.2 in two ways. First, high interest columns are compared in the enhance tab. As the most recent operation has the highest interest, this tab shows visualizations comparing columns in the most recent operation to other recently interacted columns. Second, the recommendations in the other tabs are sorted according to this interest model. Particularly as datasets grow wide, there are many possible visualizations that can be shown to visualize univariate and bivariate distributions. Therefore ranking visualizations becomes increasingly important to show users visualizations that correspond to columns they care about. We group our recommended visualizations into the same task groups as Lux, including tabs for Correlation showing scatterplots and Distribution showing histograms. However, we sort the visualizations in each of these tabs by the column ordering provided by our model. *Solas* thereby shows visualizations most relevant to a user's recent interactions at the front, reducing the time needed to scroll to find relevant charts.

Figure 1 demonstrates this sorting after several analysis steps. Since our analyst has most recently interacted with the Class variable it is shown on the left-hand side of *Solas*'s UI. Next, *Solas* shows `Class` relative to `Worldwide_Gross`, `MPAA_Rating`, and `Running_Time_min` since these columns were also interacted with during the analysis in decreasing order of inferred interest. Other tabs such as Correlation, Distribution, Occurrence, and Temporal use this same ordering to present the most relevant visualizations to the user first.

4.3 Operational Type Inference

The last way that we use history to improve recommendation is by using the operations that an analyst applies to each column to do better measurement type inference. Measurement types refer to the

meaning of a column such as nominal, ordinal, interval, and ratio variables as opposed to data types such as int or float.

Type inference in *Solas* happens in two stages. First, we infer types with traditional methods based on dataset statistics and data types. Next, we update these default measurement type inferences based on the operations a user applies to each column. With better types for each column, we are able to visualize data with more appropriate encodings. We infer types for levels of measurement based on the operations supported by each level. Nominal variables only support equality, ordinal variables also support comparison, interval variables support addition and subtraction, and ratio variables also support multiplication and division [Ste46]. Each "higher" level supports all operations below. When we see an operation applied to a column, we know that the column must be *at least* of that level. Table 2 shows Python operators and their corresponding level of measurement that we use for type inference. For simplicity in *Solas*, we visualize variables as either nominal or quantitative and therefore group nominal and ordinal inferences into nominal and interval and ratio into quantitative.

Figure 4 demonstrates how we can update the type of a variable from interactions and how this affects recommended visualizations. In this example, the `Viewership` column is inferred as nominal by default since it has a low cardinality. However, once a multiplication operation is applied to the column, we learn this column must be quantitative in order to support multiplication. This type update changes the univariate visualization of `Viewership` from a bar chart to a binned histogram, and changes how `Viewership` is visualized in bivariate distributions as well. These type updates only go up the levels. Therefore if a column is inferred to be quantitative by default and we execute `df.col == 45`, we will not change the type to nominal since quantitative columns also support equality.

In addition to mathematical operators, we also learn type information from the aggregation functions presented in Table 2. The levels in each of these aggregation functions correspond to the operations required to carry out that functionality. For instance, `median` only requires greater than or less than comparisons so lets us learn ordinal information, whereas calculating the product (with `prod`) of a column requires multiplication and thus tells us this column should be Ratio typed.

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

4.4 Interacting with History

In *Solas*, all of the history tracking and recommendation happens under the hood. However, we support interactions for users to browse the history of operations that occurred on a dataframe (or its ancestors) to better understand the past operations and visualize them. When a user clicks on a previous step in the analysis, we show them the visualizations for this specific task. Additionally, users can delete history items if they do not want them to influence their recommendations.

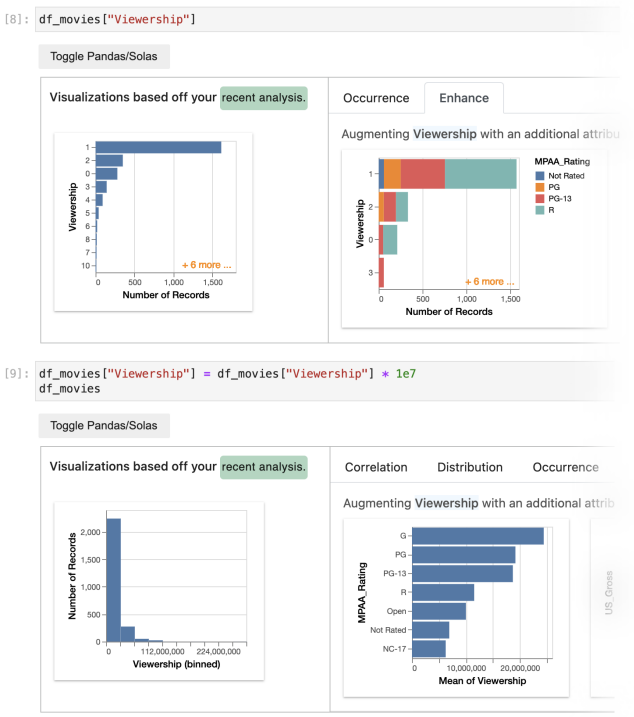

Figure 4: *Viewership initially represents the count of viewers in 10 millions. Since it has low cardinality, it is visualized as a nominal variable. However, when we re-scale the column by multiplying by 10 million, Solas infers that Viewership must be a quantitative column that supports multiplication and visualizes accordingly.*

5 Usage Scenario

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

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

5.1 Supporting Analysis with Task-Specific Visualizations

To get an overview of her data, she calls `df_movies.describe()`. There are initially eight quantitative columns in the dataset, and the `describe` function returns summary statistics for each column. To visualize this data, *Solas* plots each of these eight columns as a boxplot (Figure 5A). Looking at the plots, she notices many high-range outliers on the `US_DVD_Sales`, `Worldwide_Gross`, and `US_Gross` columns. By

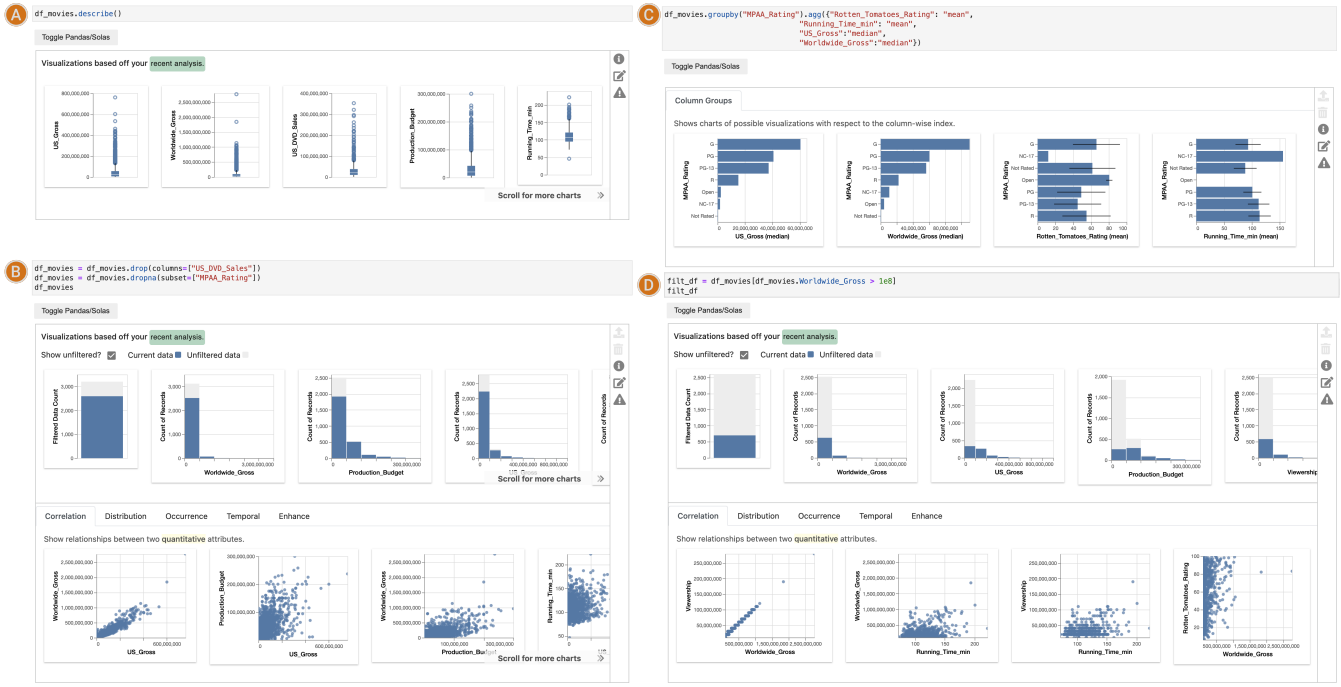

Figure 5: *Solas tracks history throughout an analysis to provide improved visualizations. We show four snapshots from an example analysis that demonstrate the recommendations from Solas. (A) When an analyst calls* `describe`*, we visualize the returned data as a boxplot by using information that is no longer present in the returned data to plot outliers. (B) After cleaning their data by dropping columns and nulls, Solas shows how the distribution of other columns change. (C) For groupbys and aggregations, we use history information to add better x-axis labels and include error bars when plotting the mean. (D) When filtering, Solas shows the background distribution of each column from the parent data relative to the filtered data. Users can toggle the background distribution on and off.*

using data from the parent dataframe, `df`, to plot these outliers, *Solas* is able to provide a visualization that best caters to the overview task of `describe`.

Next, our analyst checks if she needs to clean any columns. She calls `df_movies.isna()` and looks at the bar charts to see which columns have nulls in them. Most columns have no or very few nulls; however, the `US_DVD_Sales` column is almost all null. She decides to drop this column. Additionally, she filters to only keep non-null rows for `MPAA_Rating`, as she is potentially interested in including this column in her model. `MPAA_Rating` corresponds to movie ratings like PG-13 or R. *Solas* visualizes the returned data as a filter so our analyst can inspect how the `dropna` operation affects the distribution of other columns (Figure 5B).

Next, our analyst looks to explore how metrics in the dataset differ across `MPAA_Rating` to see if this column will be helpful for modeling later. First, she calls `df_movies["MPAA_Rating"].value_counts()` to understand the distribution. *Solas* knows the data returned is pre-aggregated and plots the results in a bar chart. Most movies in this dataset are rated R, followed by PG-13. She then groups by `MPAA_Rating` and aggregates several other columns. Since her earlier exploration revealed the skewed distribution of the `US_Gross` and `Worldwide_Gross` columns, she aggregates them by their median across `MPAA_Rating`. She also calculates the mean of `Rotten_Tomatoes_Rating` and `Running_Time_min`. When visualizing these results, *Solas* automatically includes error bars for the mean calculations to give more context (Figure 5C). Our analyst notices that the `Rotten_Tomatoes_Rating` has similar standard deviations across ratings, except for the Open category, which has a much smaller standard deviation on the visualization.

5.2 Improving Visualizations with Type Inference Updates

To continue transforming her dataset, the analyst inspects the `Viewership` column (Figure 4 Top). Since this column has low cardinality, *Solas* initially infers the type to be nominal. However, the analyst knows that this column represents the viewership in units of 10 million, so she multiplies the column by 10 million to get the raw viewership count. After this operation, *Solas* has evidence that `Viewership` is a quantitative column and updates the type and visualizations accordingly (Figure 4 Bottom). This operation-based type update would not be possible without tracking and reasoning about analysis history.

5.3 Surfacing Visualizations with Column Interest Model

Our analyst turns her analysis towards the `Worldwide_Gross` column, since she will be using this column for predictions. Her earlier analysis suggested this column is right skewed, and has a

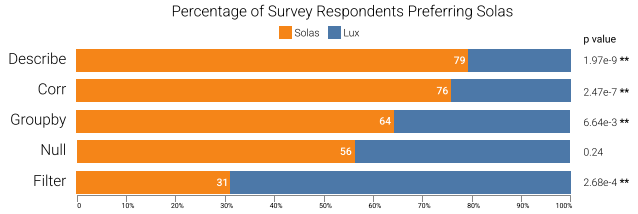

Figure 6: *Survey participants (N=87) significantly preferred Solas's encodings for* `describe`*,* `corr`*, and* `groupby`*. For* `isNull`*, they found either encoding equally acceptable. For* `filter`*, participants significantly preferred the Lux encoding and are given the option to toggle the background distribution on and off in Solas. P values marked with (**) are below 0.05 and considered significant.*

large number of high-value outliers as shown in the boxplot for `describe`. Still, she wants to recheck the distribution again to confirm. She simply references this column in a cell by typing `df_movies.Worldwide_Gross` and *Solas* plots both the distribution of this column as well as `Worldwide_Gross` relative to other columns in the dataset. *Solas* sorts these visualizations so that columns she has recently interacted with appear first such `Viewership` or `MPAA_Rating` (as reflected by the ranking at execution count `[10]` in Figure 2).

After looking at the histogram, she creates a predictive model for high-grossing movies. She applies a filter to `Worldwide_Gross`, and *Solas* shows how this filtered data compares to the unfiltered set (Figure 5D). She can toggle the background distribution on and off to inspect more closely the returned data with or without this additional context. Once again, this background context would be impossible without *Solas*'s history of her analysis. She iterates on her filter and decides on a value of $100M for her threshold since the *Solas* filter plots revealed that about a quarter of movies earn more than this much. Our analyst then creates a new binary variable called `Class` for whether or not a movie makes more than $100M. She will be using this variable as the prediction target for a binary classifier. She visualizes her data once again by displaying the dataframe with a call to `df_movies` (Figure 1). Since she has most recently created the `Class` variable, this action is highlighted in *Solas* and a bar chart for `Class` is shown. Furthermore, by looking at the enhance tab, she can see `Class` relative to other variables in the dataset. These recommendations are sorted by variables she has interacted with recently so `Class` vs `Worldwide_Gross` is shown first followed by `Class` vs `Viewership` and so on.

Finally, our analyst calls `df.corr()` to see how the other columns in her data are correlated with her new `Class` column (Figure 3). She notices several features have a strong correlation with the `Class` such as `US_Gross`. In contrast, others like `IMDB_Rating` have a weaker correlation, so they likely provide less predictive value. With this, our analyst is happy with her data exploration and is ready to begin modeling. By using *Solas*, she was able to spend less time thinking about how to visualize her data and more time focusing on the insights of her analysis.

6 Evaluation

To evaluate how well *Solas* suggests visualizations that users find helpful on real-world tasks, we ran a survey to assess if users preferred the task-specific encodings provided by *Solas* versus the default encodings shown in Lux. We use Lux as an example of visualizations that will be presented by a state-of-the-art recommendation tool that does *not* use history. Our evaluation demonstrates the value of incorporating analysis history into visualization recommendation as *Solas* suggests preferred encodings for several tasks. We chose to evaluate the task-specific encodings since they were most easily assessed by crowd workers and do not require an entire analysis context to be useful; we demonstrate the utility of our model of column interest in Section 5.

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

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

Participants' preferences are summarized in Figure 6. We conducted t-tests to assess if the fraction of responses was significantly different than 0.5 (which would indicate no preference). For `describe`, `corr`, and `groupby` participants significantly preferred the *Solas* encodings. For `describe`, participants preferred how *Solas*'s boxplot matched the descriptive statistics: "[Solas] actually shows a distribution to the descriptive statistics, so we can see if there's any skew/outliers/etc." (P30). Interestingly, for `describe` Lux re-aggregates the data and presents a misleading histogram that assumes the data is a normal quantitative column. However, 21% of participants still preferred the histogram since it "Just seems easier to analyze and see" (P26). This underscores the importance of communicating data with appropriate encodings since users will interpret the chart even if it is *visualizing irrelevant data.*

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

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

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

7 Discussion & Limitations

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

Our paper is subject to several limitations. As a system, *Solas* currently only tracks history for Pandas dataframes and works in Jupyter (or similar) notebooks. However, we believe the ideas of using analysis history tracking to augment visualization recommendation are applicable beyond Python and Pandas programming. Our evaluation focuses on how crowd workers successfully understand *Solas*'s task-specific visualizations. However, further studies might explore how systems augmented with history tracking like *Solas* help analysts explore their data on more in-depth analysis tasks.

7.1 Preventing Erroneous Findings

In building and evaluating *Solas*, we noticed trends around how users interact with their data and analysis histories. During our evaluation, some users still preferred the poor encoding of the aggregated data even though the chart communicated false findings such as re-aggregating pre-aggregated data into a histogram. By ensuring proper task-specific visualizations, *Solas* can help make sure that data analysts engage with their data truthfully and are not led astray by poor encoding. This finding echoes similar research from the XAI community about how users trust interpretability visualizations of a machine learning model even if the results are false [KNJ*20].

7.2 Next Step Recommendation

Once we have detailed information about an analyst's steps during their data exploration, we can use this information beyond visualization recommendation. By aggregating across multiple analyses, we can begin to recommend potential next steps during recommendation with accompanying visualizations. Existing work in this area typically mines Jupyter notebooks found on Github to understand how users go about their analysis [RCK*21, YH20], however, notebooks found on Github are often incomplete, or do not run [RTH18]. Furthermore, they do not represent the full breadth of analysis since only one snapshot is uploaded that may not contain previous analysis paths that have been deleted from the notebook. By using *Solas*, we can track detailed information about the full breadth of a user's analysis and use this data to provide improved next step recommendations.

7.3 Capturing Interest Across Multiple Analyses

In addition to next step recommendations, we can use aggregated analysis histories to better understand how analysts typically interact with a particular data source. Many teams interact with (versions of) a remotely stored data source. Each of these analyses can be tracked through *Solas* to build a model of how users interact with that data *across* analyses. When a user begins a new analysis, we can help them bootstrap their exploration by demonstrating how people typically explore or interact with that data source. We could even develop analysis templates based on common practices for single data sources or within a domain.

8 Conclusion

We present *Solas*, a visualization recommendation tool that uses analysis history to improve recommendations. By understanding the context and provenance of analysis, history-based recommendations provide improvements including task-specific visualizations, sorting visualizations by column interest, and operation-based type inference. In our user evaluation, participants engaged with the task-specific visualizations and found them helpful for understanding their data. Our use case demonstrates the utility of our column interest model. Finally, we discuss how history-based recommendations might be used in other contexts beyond notebook programming.

9 Acknowledgements

We would like to thank Marti Hearst, Venkat Sivaraman, and Alex Cabrera, along with our anonymous reviewers for their feedback on this work. This work was supported by a grant from Apple, Inc. Any views, opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as reflecting the views, policies or position, either expressed or implied, of Apple Inc.

References

[80820] 8080 LABS: bamboolib. https://bamboolib.8080labs.com/, 2020. 2

[ada] ADAMEROSE: PandasGUI. https://github.com/adamerose/pandasgui. 2

[AGG*15] ALIGON J., GALLINUCCI E., GOLFARELLI M., MARCEL P., RIZZI S.: A collaborative filtering approach for recommending olap sessions. Decis. Support Syst. 69, C (jan 2015). doi:10.1016/j.dss.2014.11.003. 2

[DHPP17] DEMIRALP C., HAAS P. J., PARTHASARATHY S., PEDAPATI T.: Foresight: Recommending visual insights. Proc. VLDB Endow. 10, 12 (aug 2017). doi:10.14778/3137765.3137813. 3

[Fbd] FBDESIGNPRO: sweetviz. https://github.com/fbdesignpro/sweetviz. 2

[GW09] GOTZ D., WEN Z.: Behavior-driven visualization recommendation. In Proceedings of the 14th International Conference on Intelligent User Interfaces (New York, NY, USA, 2009), IUI '09, Association for Computing Machinery. doi:10.1145/1502650.1502695. 3

[Hee19] HEER J.: Agency plus automation: Designing artificial intelligence into interactive systems. Proceedings of the National Academy of Sciences 116 (2019). doi:10.1073/pnas.1807184115. 2

[HMSA08] HEER J., MACKINLAY J., STOLTE C., AGRAWALA M.: Graphical histories for visualization: Supporting analysis, communication, and evaluation. IEEE Transactions on Visualization and Computer Graphics 14, 6 (2008). doi:10.1109/TVCG.2008.137. 2

[Jon72] JONES K. S.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation 28 (1972). doi:10.1108/eb026526. 4

[KHM17] KERY M. B., HORVATH A., MYERS B. A.: Variolite: Supporting exploratory programming by data scientists. Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (2017). doi:10.1145/3025453.3025626. 2

[KM18] KERY M. B., MYERS B. A.: Interactions for untangling messy history in a computational notebook. In 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2018), IEEE. doi:10.1109/VLHCC.2018.8506576. 2

[KNJ*20] KAUR H., NORI H., JENKINS S., CARUANA R., WALLACH H., WORTMAN VAUGHAN J.: Interpreting interpretability: Understanding data scientists' use of interpretability tools for machine learning. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (2020), CHI '20, Association for Computing Machinery. doi:10.1145/3313831.3376219. 10

[KRA*18] KERY M. B., RADENSKY M., ARYA M., JOHN B. E., MYERS B. A.: The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. Association for Computing Machinery, New York, NY, USA, 2018. doi:10.1145/3173574.3173748. 2

[LTA*22] LEE D. J. L., TANG D., AGARWAL K., BOONMARK T., CHEN C., KANG J., MUKHOPADHYAY U., SONG J., YONG M., HEARST M. A., PARAMESWARAN A. G.: Lux: Always-on visualization recommendations for exploratory data science. VLDB (2022). doi:10.14778/3494124.3494151. 2, 3

[Mac86] MACKINLAY J.: Automating the design of graphical presentations of relational information. ACM Trans. Graph. 5, 2 (apr 1986). doi:10.1145/22949.22950. 3

[MHS07] MACKINLAY J., HANRAHAN P., STOLTE C.: Show me: Automatic presentation for visual analysis. IEEE Transactions on Visualization and Computer Graphics 13, 6 (2007). doi:10.1109/TVCG.2007.70594. 3

[MWN*19] MORITZ D., WANG C., NELSON G. L., LIN H., SMITH A. M., HOWE B., HEER J.: Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. IEEE Transactions on Visualization and Computer Graphics 25, 1 (2019). doi:10.1109/TVCG.2018.2865240. 3, 5

[Pan] PANDAS: Pandas: Python data analysis library. URL: https://pandas.pydata.org. 3

[Per18] PERKEL J. M.: Why jupyter is data scientists' computational notebook of choice, Oct 2018. URL: https://www.nature.com/articles/d41586-018-07196-1. 3

[PMX*20] PETERSOHN D., MACKE S., XIN D., MA W., LEE D., MO X., GONZALEZ J. E., HELLERSTEIN J. M., JOSEPH A. D., PARAMESWARAN A.: Towards scalable dataframe systems. Proc. VLDB Endow. 13, 12 (jul 2020). doi:10.14778/3407790.3407807. 4

[PP] PANDAS-PROFILING: pandas-profiling. https://github.com/pandas-profiling/pandas-profiling. 2

[RCK*21] RAGHUNANDAN D., CUI Z., KRISHNAN K., TIRFE S., SHI S., SHRESTHA T. D., BATTLE L., ELMQVIST N.: Lodestar: Supporting independent learning and rapid experimentation through data-driven analysis recommendations. Proceedings of the 2021 IEEE Conference on Visualization and Visual Analytics (2021). 2, 10

[RTH18] RULE A., TABARD A., HOLLAN J. D.: Exploration and Explanation in Computational Notebooks. Association for Computing Machinery, New York, NY, USA, 2018. doi:10.1145/3173574.3173606. 10

[sd] SFU DB: dataprep. https://github.com/sfu-db/dataprep. 2

[SKL*16] SIDDIQUI T., KIM A., LEE J., KARAHALIOS K., PARAMESWARAN A.: Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. Proc. VLDB Endow. 10, 4 (nov 2016). doi:10.14778/3025111.3025126. 3

[Ste46] STEVENS S. S.: On the theory of scales of measurement. Science 103, 2684 (1946). doi:10.1126/science.103.2684.677. 7

[VRM*15] VARTAK M., RAHMAN S., MADDEN S., PARAMESWARAN A., POLYZOTIS N.: Seedb: Efficient data-driven visualization recommendations to support visual analytics. Proc. VLDB Endow. 8, 13 (sep 2015). doi:10.14778/2831360.2831371. 3, 6

[WDBD21] WEINMAN N., DRUCKER S. M., BARIK T., DELINE R.: Fork it: Supporting stateful alternatives in computational notebooks. Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (2021). doi:10.1145/3411764.3445527. 2

[WHS20] WU Y., HELLERSTEIN J. M., SATYANARAYAN A.: B2: Bridging Code and Interactive Visualization in Computational Notebooks. Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3379337.3415851. 2

[WMA*16a] WONGSUPHASAWAT K., MORITZ D., ANAND A., MACKINLAY J., HOWE B., HEER J.: Towards a general-purpose query language for visualization recommendation. In ACM SIGMOD Human-in-the-Loop Data Analysis (HILDA) (2016). doi:10.1145/2939502.2939506. 3

[WMA*16b] WONGSUPHASAWAT K., MORITZ D., ANAND A., MACKINLAY J., HOWE B., HEER J.: Voyager: Exploratory analysis via faceted browsing of visualization recommendations. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis) (2016). doi:10.1109/TVCG.2015.2467191. 3

[WQM*17] WONGSUPHASAWAT K., QU Z., MORITZ D., CHANG R., OUK F., ANAND A., MACKINLAY J., HOWE B., HEER J.: Voyager 2: Augmenting visual analysis with partial view specifications. In ACM Human Factors in Computing Systems (CHI) (2017). doi:10.1145/3025453.3025768. 3

[XOW*20] XU K., OTTLEY A., WALCHSHOFER C., STREIT M., CHANG R., WENSKOVITCH J. E.: Survey on the analysis of user interactions and visualization provenance. Computer Graphics Forum 39 (2020). doi:10.1111/cgf.14035. 2

[YH20] YAN C., HE Y.: Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020), pp. 1539–1554. doi:10.1145/3318464.3389738. 2, 10