Designing Human-Centered Interactive Visual Interfaces for Feature Engineering and Data Labeling

Alan Luo, Dylan Cashman, Remco Chang
Visual Analytics Lab at Tufts, Department of Computer Science
Tufts University, Medford, MA

Abstract

The goal of this research is to create interactive interfaces to help non-experts construct inquiries that help them explore their data. Our interface focuses on the task of data selection. In the context of relational algebra, we can understand our task as constructing a SQL WHERE clause. Existing systems for query construction focus on building around the SQL syntax, and thus cannot be used by non-experts. They do not offer useful interfaces for exploring the query space, since they are essentially alternative views of the query space. They offer no simplification or restriction of the complex space, allowing for non-experts to construct ambiguous or impossible queries. This research focused on building an interactive interface that aids non-experts by restricting the query space and by proving responsive feedback. We used our interface in conjunction with a back-end provided by Feature Labs, a startup company that designs software to make data analysis easier for non-expert users. The interface was tailored towards data labeling on time series data.

Background

Data is any quantifiable information. Relational algebra is a system used to classify data. Under relational algebra, data is divided into three fundamental types. *Categorical* data is classified into discrete, unordered sets. Examples include country of origin, type of tree, etc. *Ordinal* data is classified into discrete, ordered sets. The most common example is the Likert Scale, which rates preferences from negative to positive. *Numerical* data is any number, and can be discrete or continuous. Examples include height, price, etc. Arbitrarily complex data can be constructed using these basic types using relational algebra. In a tabular dataset, rows are entries and columns are features. For the data in Figure 1, `SUM(Qty*Price)` is an example of a useful feature that can be calculated, representing the total invoice price.

Invoice #	Description	Qty	Time	Price	User ID	Country
368	YELLOW COAT	3	8:34	4.95	13047	UK
368	BLUE COAT	3	8:34	4.95	13047	UK
369	BATH BUILDING	3	8:35	5.95	13047	UK
370	ALARM CLOCK PINK	24	8:45	3.75	12583	FR
370	ALARM CLOCK RED	24	8:45	3.75	12583	FR
370	ALARM CLOCK GRN	12	8:45	3.75	12583	FR
370	STICKER SHEET	12	8:45	0.85	12583	FR

Figure 1. (Abridged) rows of two users from an OpenML retail dataset. Each row represents a purchased item. Thus, single orders by a single user can be split into many different rows. This makes the dataset large and dense, but makes the process of obtaining useful information, such as total purchase cost or time of order, difficult.

Different features serve different problems: invoice price might be a good predictor of revenue but a poor predictor of customer zip code. Selecting salient features is an important problem in machine learning. As data collection expands, datasets often become dense in data but sparse in useful information, as seen in Figure 1. Selecting useful and relevant features is a crucial to machine learning tasks, but is very difficult. Feature engineering aims to simplify the process of feature selection by comparing large numbers of user-defined and artificially-generated features.

For our research, we collaborated with Feature Labs. The Feature Labs group provided with a back-end system which is capable of feature synthesis from relational datasets, deriving predictive models from raw data automatically, and other tasks. We focused on building an interface that would interact with their feature engineering API layer, which was built on Python.

Motivation of Research

Selecting data for feature engineering can be understood as constructing a SQL WHERE clause. The WHERE syntax is unwieldy, with single clauses often spanning pages. Existing interfaces for SQL query construction do not simplify the task of construction, they only automate menial tasks of entering text. Figure 3 shows such an interface.

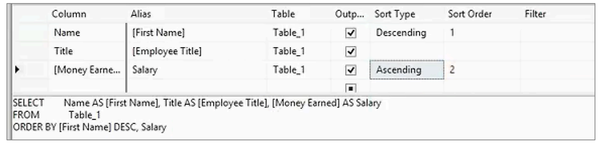

Figure 2. An example WHERE clause from Microsoft Query Builder. Notice that the table view (above) of the WHERE clause displays identical information to the raw query view (below).

Due to the one-to-one relationship between the table space and query space, users of these interfaces must already be SQL experts. This research aims to use the visualization space as a window to restrict the query space, making it possible for non-experts to create these queries. We focus on time series data, because it is common in real datasets. In particular, we target a query type supported by the predictive engineering interface in the API.

Predictive Engineering

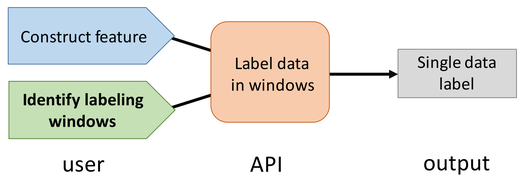

Figure 3. The predictive engineering process as given by the API. Using the Python API, the user constructs a feature and identifies labeling windows on the time series data. Then, the back-end system calculates labels for the data in each window. Finally, the labels are reduced to a single data label. Our system focuses on the step of identifying labeling windows, shown in bold.

Predictive engineering is a step that takes place before feature engineering. The objective of predictive engineering is to label data and prepare them for feature engineering. This process is shown in Figure 3. The output of this process is a Label Time for each data row. The Label Time has a label, which is the selected feature calculated for the data row over the identified windows, and a time, which is a Date object indicating the cutoff point after which data cannot be used in prediction. The time cutoff is used to ensure that training and testing data are not mixed.

We focused on simple time series windows, as given by the Label Maker API. We define a number of windows `number_windows` (or n), a window size `size_windows`, a window offset `offset_windows`, and a start date `start_date`. This gives regularly spaced data windows. A basic prediction problem will produce n Label Times per data row, with the time set to the start of each window. This basic structure is shown in Figure 4.

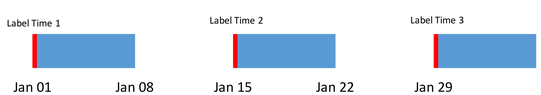

Figure 4. Sample prediction windows with `number_windows` = 2 weeks, `size_windows` = 1 week, and `offset_windows` = 2 weeks. For each window, a Label Time is appended to each data row where the label is the calculated feature and the time is the start of the window, marked in red.

Interactive Interface

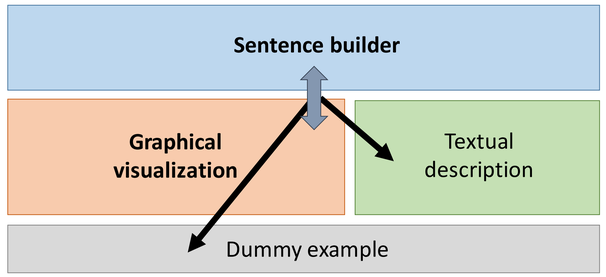

Figure 5. The basic layout of the interactive interface. The user can manipulate attributes through the bolded regions, the sentence builder and the graphical visualization. These two panels always reflect the same information, allowing for attributes like window size to be tuned back-and-forth from either one. When an attribute is modified, all four panels update in real time, though attributes cannot be tuned through the description and example panels, which serve purely for reinforcement of understanding.

The interface was motivated by the fundamental idea that a user can more effectively explore the query space when guided and restricted by certain attributes. Thus, the interface is built to help the user explore specific attributes. We focus on `number_windows`, `size_windows`, and `offset_windows`, the primary attributes used by the API. We designed the interface around the idea that multiple simultaneously updating views of the same information would allow the user to adjust a single attribute and observe multiple outlets of positive reinforcement, thus enhancing learning.

We employ many user interface techniques to achieve this reinforcement, including buttons, color changes, click-and-drag, and tooltips. Parts of the actual interface are displayed in Figure 6 and Figure 7.

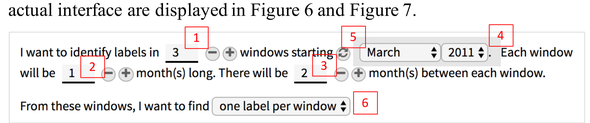

Figure 6. The sentence builder from the interactive interface. 1, 2, 3, 4: input fields allow the user to modify the four primary attributes used in predictive engineering. 5: the "cycle" button switches all time from absolute to relative time, a functionality also supported by the API. 6: he final input field determines whether 1 or n Label Times should be produced for each row.

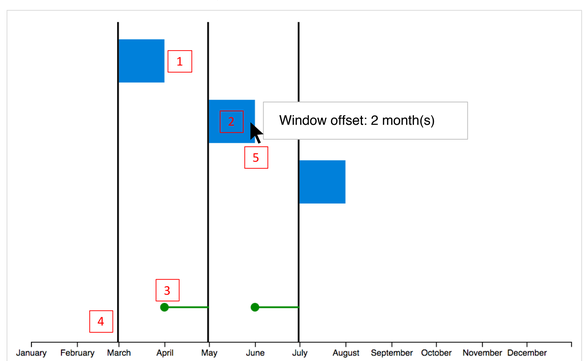

Figure 7. The graphical visualization from the interactive interface. 1: click-and-drag handle to change `size_windows`. 2: click-and-drag handle to change `offset_windows`. 3: visual indicator of space between data windows. This turns red if the windows overlap. 4: indicator of time from Label Time. 5: tooltip that appears when the mouse hovers over a draggable handle.

API Communication

The interface communicated with the Feature Labs API via JSON. On the client side, we used Javascript's built-in `JSON.stringify()`. On the server side, we used Python's default `json` module. We use JSON due to its universal interpretability and because its hierarchical nature would allow us to communicate relational aspects of the data. Our final prototype did not end up containing this relational information, but the framework remains in case such a feature is to be added in the future. Below is an example JSON as a Python string.

```
myObj = '{
    "number_windows":"3 weeks",
    "size_windows":"1 weeks",
    "offset_windows":"2 weeks",
    "estimate_type":"type_1",
    "start_date":"2/28/2011" }'
```

Variables are naively stored without any particular structure. In this example, `estimate_type` over n windows controls whether 1 or n labels are produced per query. On the server side, query windows are understood by default as being disjoint, and n labels are produced for n windows. To produce 1 label for n windows, we instead create one window and filter out the empty parts. This is illustrated in Figure X.

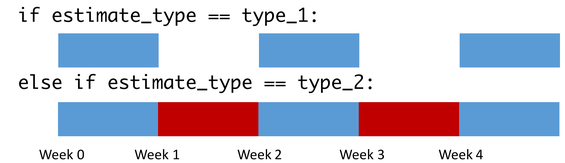

Figure 8. Type 1 and 2 queries on 3 week-long windows separated by a week each. Type 1 queries return 1 label per estimation window, while Type 2 queries return 1 label in total, and are constructed by taking a single estimation window and filtering out the empty space.

Conclusions

We constructed a system for providing interactive visual feedback to non-experts for constructing feature engineering problems. We used a multiple-panels approach to reinforce alternative understandings of one system. Cursory tests suggest that the interface, when compared to using the Python API, will both help users understand the task better and decrease the time spent constructing these problems.

Future Work

Though initial tests prove promising for improving understandability and performance, the system lacks a proper evaluation. In addition, further tuning of visual parameters is necessary to ensure effectiveness of the interface. Furthermore, the system does not fully express the depth of the API, and lacks a few key features, such as full support of relative event time. Future work will aim to iterate on the system, expand for additional features, and evaluate the system using a proper user study.

Acknowledgments

I would like to thank the Visual Analytics Lab at Tufts, along with Prof. Remco Chang and PhD candidate Dylan Cashman, for advising this research and providing constant guidance. Additionally, I would also like to acknowledge Dr. Hogue and my fellow students in the Science Research Program for their support.