HPDAV - Project 1

Giorgio Daneri[1]

University of Luxembourg.

Contributing authors: giorgio.daneri.001@student.uni.lu;

1 Introduction

This project aims at implementing two visualizations for bike sharing dataset, which have multiple features and are synchronized with each other. My second choice is a parallel coordinates plot, which captures in a clear and decisive way pattern that are present inside the data. The main components of the visualization are:

- **Scatter Plot:** it features a 2D brushing functionality to select data points with the cursor, as well as an hovering functionality to display the attribute values of data points.
- **Scatter Plot Control Bar:** allows the user to choose which variables to plot over the x and y axes, the attributes used to build the color scale and the size scale respectively. It preserves its interpretability even for large datasets like SeoulBikeData, offering a high degree of flexibility. Though countermeasures shall be used against cluttering, which makes patterns and clusters indistinguishable. Some comprehension difficulties may arise in plotting categorical variables along the axes, while color is best suited to represent them.
- **Parallel Coordinates Plot:** a simple chart with two vertical axes. I made this choice due to its effective representation of data patterns and correlations between different features of the dataset. It is a compact way of visualizing high-dimensional data without sacrificing too much readability. It is straightforward for the user and allows for easy interaction and customization through variable setting. 1D brushing is implemented for both axes, effectively becoming two-dimensional. When both brushes are used, the intersection of the selections is considered.
- **Parallel Coordinates Control Bar:** allows the user to choose which attributes to visualize on the two axes, and the possibility to invert the axes individually to better visualize patterns within the data. The user can also select the attribute used to build the color scale.

I chose to offer a wide range of possible configuration parameters, thus the user can select all the attributes of the dataset as visualization variables, either as dimensional, color or size variable. This grants more freedom in configuring the visualization, but may decrease its readability in specific cases, e.g. when selecting an ordinal variable for color, like "Date". The same goes for the variable used to define the size scale of the dots in the scatter plot, which may not convey significant information in some cases. The color variable should be mainly used for categorical or ordinal variables that have a limited range of possible values, thus making the colors clearly distinguishable at a glance. Also, separate control bars are useful to visualize different variables on the plots, making it easier to identify patterns and correlations. When it comes to the code structure, the two components implementing the plots consist of two javascript files each. The first one is the React component, which acts as a bridge between the Redux state and the D3 implementation. It is tasked with retrieving the state of all the variables needed inside the component, as well as dispatching actions when certain events occur, i.e. brushing on the plot. It uses React hooks to manage the state of life cycle of the component instance. More precisely, on mount it initializes the component and all its fields with default values, which are fetched from the corresponding Redux state. When unmounted, it clears the D3 component and all its elements. An important function of the react

component is to update the D3 instance by fetching the updated data from the state. Finally, it provides a container for the graphical instance, which is useful to apply styles through a CSS file.
The second file is the actual D3 implementation, hence the visualization of the data values that are passed to it with a precise logic. It handles low-level manipulations of the DOM elements, in order to render SVG elements, e.g. axes, dots, lines and layers for brushing and hovering. It also implements all the user interactions modes and the interactive nature of the visualization.

2 Scatter Plot

The React component of the scatter plot needs to retrieve the Redux state for the data to be rendered, all the attributes needed in the visualization and the data that has been brushed in the other plot. This is achieved thanks to the `useSelector` method of Redux, which allows access to a store state. A callback method is used to propagate the brushing interaction of the user to the other plot, by employing the `dispatch` function. In the D3 implementation, the `Create` function sets the height and width of the svg element, places it in the correct position, and initializes the axes of the plot. The `renderScatterplot` method is the central function of this component and handles all the updates following user interactions with the visualization, i.e. the modification of the attributes to be displayed. A special treatment needs to be applied to ordinal attributes, namely "Data", "Seasons", and categorical/binary ones, i.e. "Holiday" and "FunctioningDay". This is fundamental in order to still be able to include them in the visualization without breaking the plot. We need to define an ordering for all of them, so that axes ticks can be properly defined, as well as ordinal scales for color and size. For the first two attributes I applied lexicographic ordering, while for the last two I simply imposed that `No Holiday < Yes Holiday` and `No < Yes` respectively. The same rationale was used in the other chart. A problem manifested when choosing "Date" as one the axis variables, as the axis ticks were cluttered by all the dates, since automatic sampling did not work. I then manually sampled 1 every 10 values to be actually displayed on the axis ticks, while still displaying all the data points, thus improving readability. Then, in order to produce an color or size scale for these attributes, I selected the unique values by building a set, then converting it to an array and sorting it.

2.1 2D Brushing

When it comes to the 2D brushing functionality, I appended an ad hoc layer to the svg element in order to encapsulate this functionality and manage it easily. Inside the D3 brushing function, an event is triggered as soon as the interaction ends thanks to a proper hook, which notifies the React component and causes the update of the second plot based on the brushed data. This amounts to dramatically increasing the opacity of the corresponding lines, which makes them stand out. The selected data is stored in a variable, which is needed to preserve this information in case of re-rendering of the component, so that the user can modify the configuration variables while keeping the selection active. From a visualization perspective in this plot, selected items preserve the color based on the color scale, while the others turn gray.

2.2 Hovering

To implement the hovering functionality, I appended another layer on the svg element, selected all the points of class `dotG` and modified them through the join function. When entering, a tooltip is displayed in html format, showing the values of the point attributes. When the user brushes data on the other plot, this component is notified through an appropriate dispatch function. In the `isBrushed` helper function I check if the data points of the scatter plot match with the values brushed on the second chart based on their unique ids. Then the `updateDots` function applies the colormap only to the selected points, while the others are displayed as gray.
I had issues when trying to conciliate the hovering functionality with a smooth transition of the points. I had to append all the dots to an ad hoc layer for the hovering, otherwise any interaction outside of the scatterplot would cause a detach of the listener related to this functionality. For this reason, every time `renderScatterplot` is called, I remove this layer and create it again. This is inefficient, but it is the only way I found to make hovering working properly. Since all the points are created from scratch every time, the transition would move them from the upper-left corner to their updated position, rather than from their current position to the new one. This makes the transition meaningless, so I removed it altogether.

3 Scatter Plot Control Bar

Its purpose is to control which information is shown on the scatter plot. It consists of four drop down menus, for the axes, color and size variables. In order to fetch the items to be displayed, I used a selector to access the data stored in `state.dataSet` and the `useEffect` hook that listens for changes to the `data` slice and sets the column names. Then I obtain the default configuration from the Redux store by the means of `state.config`. All modification to the variables are dispatched once the user click on the "Generate" button. Upon the occurrence of this event, the `generateFromConfig` action is dispatched and causes an update of the state that is associated with the scatter plot component, thus causing its update and re-rendering.

4 Parallel Coordinates Plot

In the React component I employ an approach analogous to the scatter plot. First of all, the dataset and all the attributes for the configuration are fetched from the Redux store through the appropriate methods. Then the plot is rendered and updated by the means of two separate `useEffect` hooks, which also supply all the necessary variables for the axes, size and color scales. In the D3 implementation, most of the functionalities are enclosed in the `drawParallelCoordinates` method. As in the scatter plot, I impose an ordering on the categorical and ordinal variables with the same rationale, as well as building the color scale and the axes. If the user selected the checkbox to invert an axis, the range is inverted accordingly. This can be useful to better visualize patterns in some cases. The line generator uses a very low opacity in order to make the chart readable and allow the user to identify data clusters. This addresses one of the main problems of a parallel coordinates chart, i.e. cluttering due to the presence of too many data points, which is actually our case. Furthermore, the color scale is applied to the lines based on the selected attribute.

4.1 1D Brushing

Finally, the brushing implementation features two 1D vertical brushes, one for each axis, which are D3 `brushY` functions. The user can use both of them at once, so that the intersection is considered. A single function handles both of them, so we need to select the right one in order to filter the brushed data values. The opacity of selected lines is increased to make them clearly visible. After the extraction from the selection, the data is communicated to the Redux store through the `dispatch` function. I make sure that if no brushing is performed by the user, the Redux store is updated with an empty array, so that an eventual previous selection is reset also on the other plot.

5 Parallel Coordinates Plot Control Bar

The functionality of this component is analogous to the first control bar. The configurable variables are the same except for the line width, which I chose not to use due to its poor readability. As in the previous case, the updated state is dispatched and the Redux store updated only once the user clicks on the "Generate" button.

6 Redux Store

I used several data slices, which follow the logical partitioning of the program in the aforementioned components.

1. `brushedDataSlice` that contains the brushed data from the scatter plot
2. `brushedDataSliceSecond` that contains the brushed data from the parallel coordinates plot
3. `configSlice` that contains the variable settings of the first control bar
4. `configSliceRight` that contains the variable settings of the second control bar
5. `dataSetSlice` that stores the dataset as well as loading it at the start of the application. It defines an asynchrounos function using Redux toolkit `createAsynchThunk` used to fetch the dataset from the corresponding csv file. It is called once in the `App.js` component, so that it is available for later use by all other components.