Libra: An Interaction Model for Data Visualization (Supplemental Material)

Abstract—In this supplemental material, we first show the Unified Modeling Language (UML) diagram of the `Libra.js` in detail. Then, we show the classification of data services and show how to specify new components. Next, we show more details of `Libra.js` for creating expressive interactive visualizations. Finally, we perform an additional metric-based analysis for only the interaction-related code.

1 UML Diagram of `Libra.js`

Understanding the relationships between the components of `Libra.js` is key to grasping the robust functionality it provides. With a complex system like this, visual aids can offer clarity and context. To this end, this section begins with an exploration of the Unified Modeling Language (UML) diagram.

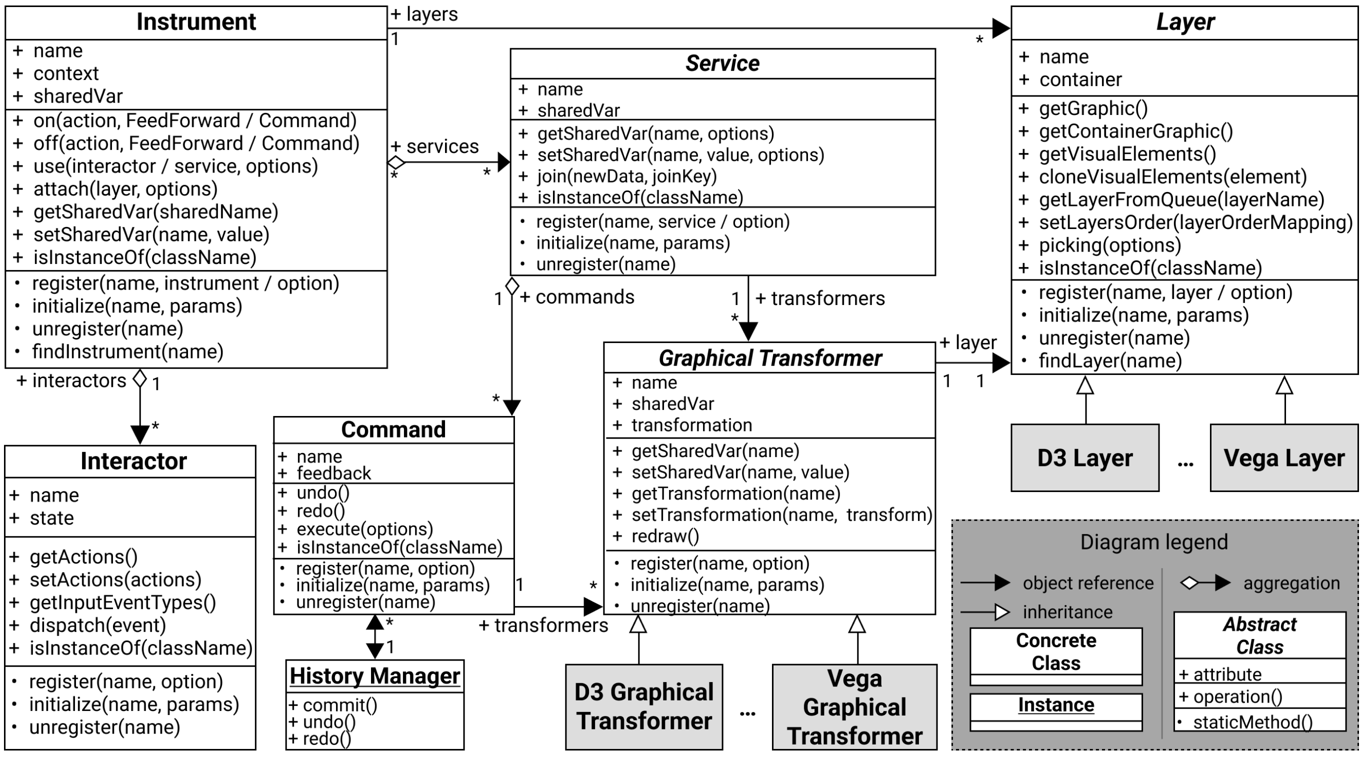


Fig. 1: The UML diagram of `Libra.js` with the diagram legend shown in the bottom right and the concrete classes in D3 or Vega with the gray background. Here we show the interfaces used in communication between classes and list some typical variables and methods in each class.

`Libra.js` is designed with a component-based architecture, reflecting a clear separation of concerns. This separation enables reusability and extensibility of its interaction components. As shown in Figure 1, at the heart of the system lies the *instrument*, a mediator responsible for translating user input into high-level actions, which are then orchestrated across various graphical layers. Each layer corresponds to a specific visualization or interaction task, including background, main, selection, and transient layers. The UML diagram highlights how *graphical transformers* serve as key components within each layer, converting data models into visual forms. These transformers ensure consistency and flexibility, supporting diverse use cases from simple selection to complex layout manipulations. Inter-layer communication is facilitated by shared services, such as the *selection service* and *layout service*, which decouple data manipulation logic from visual rendering. This modularity not only simplifies the implementation of interactions but also enables undo/redo functionalities and feedforward mechanisms. For instance, the selection service maintains state across data and visual spaces, updating both dynamically as users interact with the visualization.

2 Classification of Data Services in Libra.js

As we explore the multifaceted aspects of *data services* in our model, it becomes increasingly clear that their depth and range are vast, accommodating varied visualization and interaction needs. The main body of the paper has provided a snapshot of their functionality, focusing primarily on the selection service. However, to truly appreciate the breadth of their applications, it's essential to understand their different classifications and specialized roles. In this section, we delve into this, presenting a detailed exploration of other types of services such as *layout services* and *analysis services*. These services, each with their unique attributes and applications, offer expansive utility in visualization and interactive scenarios, providing additional depth to the interaction pipeline. Let's embark on this in-depth exploration, beginning with the fundamentals of the *layout service*, followed by a deep dive into the *analysis service*.

2.1 Layout Service

Most visualization toolkits provide a few layout algorithms allowing direct manipulation, such as force-directed graph layouts. Layout services allow the decoupling of the implementation of the actual layout and the direct manipulation to control it. A layout service computes a new layout with the interaction parameters, such as the new position of a set of items. Once a new layout is obtained, it updates the visual encoding through the shared variables. As shown in the excentric labeling example in the main paper, a label layout service is inserted for further manipulating the selected items passed from the selection service. Here, `Libra.js` does not use a selection layer but a layer for showing the arranged labels.

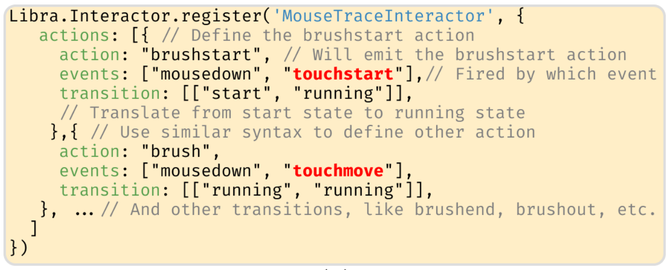
2.2 Analysis Service

Visual analytics applications use data analysis algorithms such as clustering, regression, and classification. Our model can provide such an analysis service for running statistics or machine learning (ML) algorithms on the data of interest and managing the results as one of our services; see the histogram service in the excentric labeling example. The underlying data model managed by these services can be visualized in a specific layer. For example, the histogram service visualizes the distribution of one quantitative attribute of the selected items that can be visualized on top of the main layer, as shown in the excentric labeling example.

3 Specifying Components in Libra.js

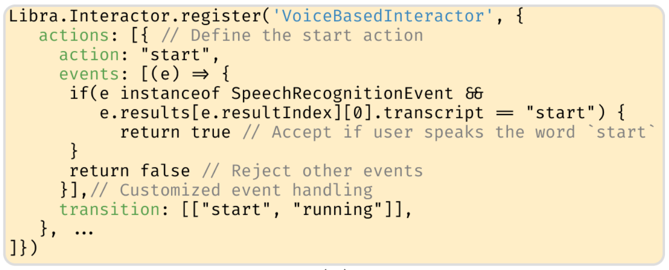
Customizing `Libra.js`'s components is a crucial aspect of its flexible design, allowing developers to tailor the interaction model to their specific needs. In this section, we will explore how to create new interactors, services, and graphical transformers in `Libra.js`.

3.1 Indicating a new Interactor



```
Libra.Interactor.register('MouseTraceInteractor', {
   actions: [{ // Define the brushstart action
     action: "brushstart", // Will emit the brushstart action
     events: ["mousedown", "touchstart"],// Fired by which event
     transition: [["start", "running"]],
     // Translate from start state to running state
    },{ // Use similar syntax to define other action
     action: "brush",
     events: ["mousedown", "touchmove"],
     transition: [["running", "running"]],
   }, ... // And other transitions, like brushend, brushout, etc.
  ]
})
```

(a)



```
Libra.Interactor.register('VoiceBasedInteractor', {
   actions: [{ // Define the start action
     action: "start",
     events: [(e) ⇒ {
      if(e instanceof SpeechRecognitionEvent &&
         e.results[e.resultIndex][0].transcript == "start") {
           return true // Accept if user speaks the word `start`
      }
      return false // Reject other events
     }],// Customized event handling
     transition: [["start", "running"]],
   }, ...
]})
```

(b)

Fig. 2: Specifying the interactor by overriding its transitions, where (a) encapsulates the low-level touch events directly and (b) offers customized functions.

Interactors in `Libra.js` are responsible for handling low-level user input events and translating them into high-level actions. To create a new interactor, **ID** needs to specify the input event types, the corresponding state transitions, and the associated actions. Figure 2 shows two examples of creating custom interactors in `Libra.js`.

In Figure 2a, we extend the mouse trace interactor to support touch events. This is achieved by adding new transitions that map touch events to the appropriate states and actions. For example, the `touchstart` event (newly added in red color) is mapped to the `start` state, and the `touchmove` and `touchend` events are mapped to the `drag` and `end` states, respectively.

Besides the traditional browser-encapsulated event type, we can also support the definition of other modalities, such as speech. Figure 2b demonstrates how to create a voice-based interactor. In this case, we override the default transitions and provide custom functions to handle voice input. Here we use the Web Speech API [5] to get the transcripts and recognize the user command.

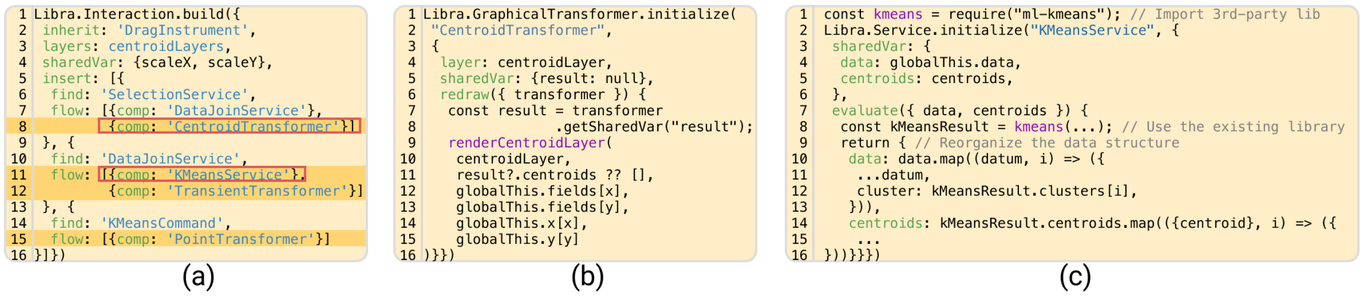
3.2 Developing the Services and Transformers

Figure 3 provides a comprehensive example of implementing a custom service and transformer for an interactive k-means clustering visualization. In Figure 3a, we define the overall structure of the interaction, specifying the layers, instruments, and services involved.

To create a custom transformer, developers can reuse their existing rendering code by invoking it within the transformer's `redraw` function, as shown in Figure 3b. This allows developers to leverage their existing visualization code while integrating it seamlessly with `Libra.js`'s interaction model.

For complex operations not provided by `Libra.js`, such as k-means clustering, developers can implement a custom service. In Figure 3c, we create a new `KMeansService` that performs the clustering algorithm based on the current centroids. The service exposes a `evaulate` function that takes the data points and the current centroids as input and returns the updated clustering result.

By creating custom interactors, services, and transformers, **ID** can extend `Libra.js` to support a wide range of interaction techniques and adapt them to their specific requirements. The modular and composable nature of `Libra.js`'s architecture enables developers to reuse and compose components, promoting code reusability and maintainability.



```
Libra.Interaction.build({
 inherit: 'DragInstrument',
 layers: centroidLayers,
 sharedVar: {scaleX, scaleY},
 insert: [{
  find: 'SelectionService',
  flow: [{comp: 'DataJoinService'},
          {comp: 'CentroidTransformer'}]
 }, {
  find: 'DataJoinService',
  flow: [{comp: 'KMeansService'},
          {comp: 'TransientTransformer'}]
 }, {
  find: 'KMeansCommand',
  flow: [{comp: 'PointTransformer'}]
}]})
```

(a)

```
Libra.GraphicalTransformer.initialize(
 "CentroidTransformer",
 {
  layer: centroidLayer,
  sharedVar: {result: null},
  redraw({ transformer }) {
   const result = transformer
                  .getSharedVar("result");
   renderCentroidLayer(
    centroidLayer,
    result?.centroids ?? [],
    globalThis.fields[x],
    globalThis.fields[y],
    globalThis.x[x],
    globalThis.y[y]
)}})
```

(b)

```
const kmeans = require("ml-kmeans"); // Import 3rd-party lib
Libra.Service.initialize("KMeansService", {
 sharedVar: {
  data: globalThis.data,
  centroids: centroids,
 },
 evaluate({ data, centroids }) {
  const kMeansResult = kmeans(...); // Use the existing library
  return { // Reorganize the data structure
   data: data.map((datum, i) => ({
    ...datum,
    cluster: kMeansResult.clusters[i],
   })),
   centroids: kMeansResult.centroids.map(({centroid}, i) => ({
    ...
})))}})
```

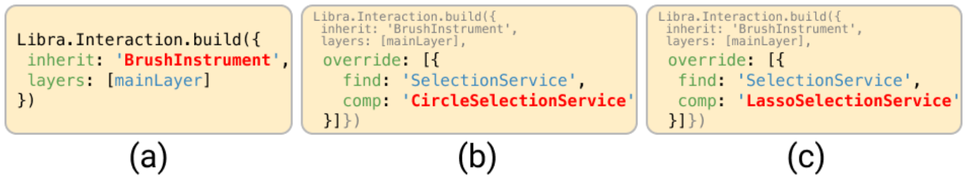
(c)

Fig. 3: Comprehensive depiction of `Libra.js`'s full code example, which shows the overall interaction structure (a), reusable transformer code (b), and an instance of algorithm integration, specifically k-means clustering (c).

4 Additional Expressiveness

4.1 Various Brush Techniques

In this section, we discuss the versatility of brush instruments in Libra by showcasing various examples of rectangular, circular, and lasso brushes (see Figure 4).



```
Libra.Interaction.build({
 inherit: 'BrushInstrument',
 layers: [mainLayer]
})
```

(a)

```
Libra.Interaction.build({
 inherit: 'BrushInstrument',
 layers: [mainLayer],
 override: [{
   find: 'SelectionService',
   comp: 'CircleSelectionService'
 }]})
```

(b)

```
Libra.Interaction.build({
 inherit: 'BrushInstrument',
 layers: [mainLayer],
 override: [{
  find: 'SelectionService',
  comp: 'LassoSelectionService'
 }]})
```

(c)

Fig. 4: Changing the selection service to generate various brush instruments: (a) a rectangular brush, (b) a circular brush, and (c) a lasso brush.

Rectangular Brush. A rectangular brush is the most common type of brush instrument, allowing users to select data points within a rectangular area. In this example, we start with the default brush instrument in Libra, which employs a rectangular shape. By creating a rectangle on the screen, users can interactively select and highlight data points that are intersected with the boundaries of the rectangle.

Circular Brush. To change the rectangular brush to a circular brush, we simply need to change the selection service to the circle selection service. The circular brush allows users to select data points within a circular area. By clicking and dragging the mouse, users can define the radius of the circle, and all data points within the circle will be selected and highlighted accordingly.

Lasso Brush. The lasso brush offers a more flexible and free-form selection experience. By replacing the selection service with the lasso selection service, users can draw an irregular, closed shape around the desired data points.

By simply changing the selection service, developers can offer various brush instruments, catering to different scenarios and user preferences.

4.2 Comparing the implementation of DimpVis

In this section, we further dig into the DimpVis interaction, and comparing the impelementation of `Libra.js` with D3 and Vega.

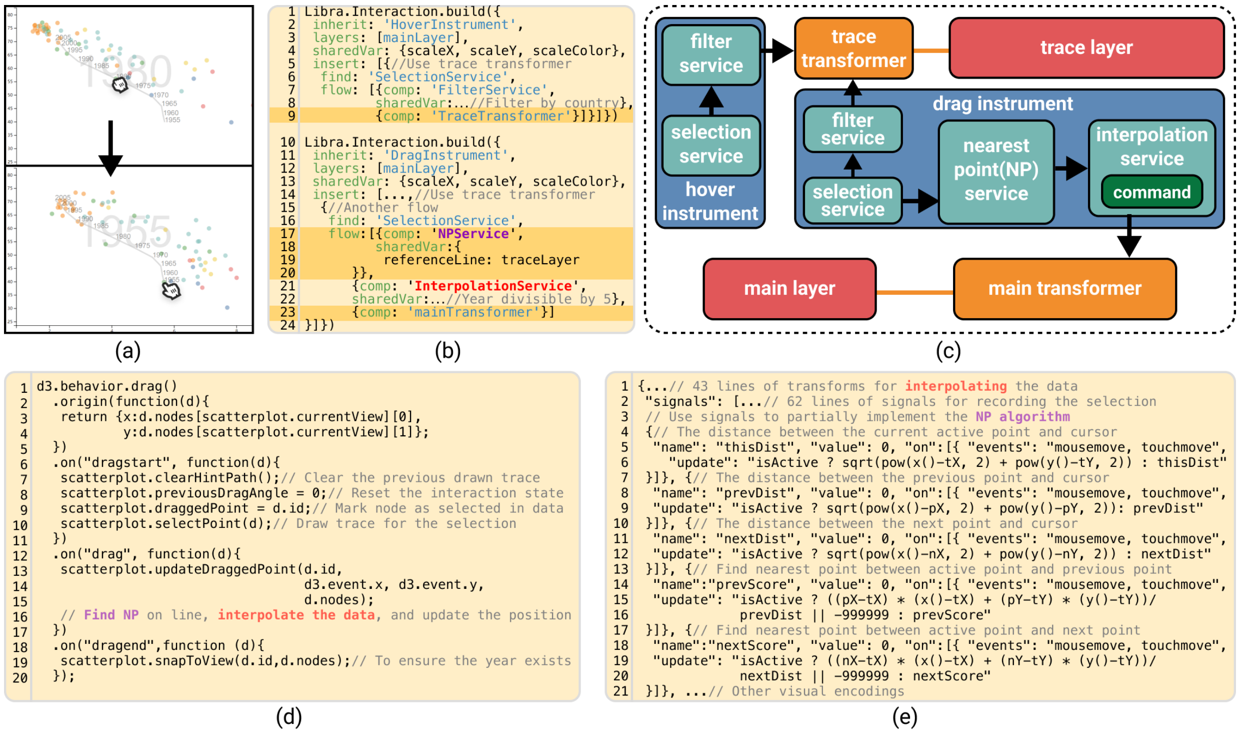
As shown in Figure 5, where the last two are provided by the DimpVis authors [1] and Vega authors [9]. Since the Vega-lite implementation [10] only provides the hover interactions without the drag ones, we do not consider it. Although the D3 implementation nicely structures the involved functions, it still requires **ID** to manually maintain the communication between different functions and the state of transient objects (e.g., time trajectory). Vega eases the implementation of callback functions but manages all selections in data space, whereas searching the nearest point in the time trajectory is natural in the visual space. Hence, its specification takes an approximate method based on the previous and next time points; they might yield incorrect nearest points in some cases.

4.3 Reusing Interaction to Other Visualizations

One of the key strengths of the Libra interaction model is its ability to reuse and adapt interactions across different types of visualizations with minimal effort. By defining interactions as standalone, modular instruments, developers can port the same interaction to new visualization applications while maintaining consistency and flexibility.

Figure 6 demonstrates this capability by applying the same interaction initially designed for the MNIST dataset to two distinct visualization types: a scatterplot matrix and a node-link diagram. For the Scatterplot Matrix (Figure 6(a-c)), the interaction designed for a single scatterplot was seamlessly extended to a matrix of scatterplots, where each cell represents a pairwise comparison of features. The modular design of the interaction ensured that it could be reused across all cells without requiring redundant implementation. Figure 6(d-f) was applied to the same interaction for a graph-based representation, where nodes and edges are the primary elements of interest. Despite the structural differences between scatterplots and node-link diagrams, the instrument's modularity allowed for a straightforward extension to this new context with minor adjustments to the interaction parameters.

In both cases, the interactions were implemented using a few lines of code by referencing the pre-defined instrument by name and applying it to the target visualization. Default parameters were automatically applied, ensuring functionality out of the box, while developers retained the ability to deeply customize these parameters for the specific requirements of their applications.



```
Libra.Interaction.build({
 inherit: 'HoverInstrument',
 layers: [mainLayer],
 sharedVar: {scaleX, scaleY, scaleColor},
 insert: [{//Use trace transformer
  find: 'SelectionService',
  flow: [{comp: 'FilterService',
         sharedVar:...//Filter by country},
         {comp: 'TraceTransformer'}]}]})

Libra.Interaction.build({
 inherit: 'DragInstrument',
 layers: [mainLayer],
 sharedVar: {scaleX, scaleY, scaleColor},
 insert: [...,//Use trace transformer
  {//Another flow
   find: 'SelectionService',
   flow:[{comp: 'NPService',
          sharedVar:{
           referenceLine: traceLayer
       }},
       {comp: 'InterpolationService',
       sharedVar:...//Year divisible by 5},
       {comp: 'mainTransformer'}]
}]})
```

```
d3.behavior.drag()
  .origin(function(d){
   return {x:d.nodes[scatterplot.currentView][0],
           y:d.nodes[scatterplot.currentView][1]};
  })
  .on("dragstart", function(d){
   scatterplot.clearHintPath();// Clear the previous drawn trace
   scatterplot.previousDragAngle = 0;// Reset the interaction state
   scatterplot.draggedPoint = d.id;// Mark node as selected in data
   scatterplot.selectPoint(d);// Draw trace for the selection
  })
  .on("drag", function(d){
   scatterplot.updateDraggedPoint(d.id,
                                  d3.event.x, d3.event.y,
                                  d.nodes);
   // Find NP on line, interpolate the data, and update the position
  })
  .on("dragend",function (d){
   scatterplot.snapToView(d.id,d.nodes);// To ensure the year exists
  });
```

```
{...// 43 lines of transforms for interpolating the data
 "signals": [...// 62 lines of signals for recording the selection
 // Use signals to partially implement the NP algorithm
 {// The distance between the current active point and cursor
  "name": "thisDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
    "update": "isActive ? sqrt(pow(x()-tX, 2) + pow(y()-tY, 2)) : thisDist"
 }]}, {// The distance between the previous point and cursor
  "name": "prevDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? sqrt(pow(x()-pX, 2) + pow(y()-pY, 2)): prevDist"
 }]}, {// The distance between the next point and cursor
  "name": "nextDist", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? sqrt(pow(x()-nX, 2) + pow(y()-nY, 2)) : nextDist"
 }]}, {// Find nearest point between active point and previous point
  "name":"prevScore", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? ((pX-tX) * (x()-tX) + (pY-tY) * (y()-tY))/
            prevDist || -999999 : prevScore"
 }]}, {// Find nearest point between active point and next point
  "name":"nextScore", "value": 0, "on":[{ "events": "mousemove, touchmove",
  "update": "isActive ? ((nX-tX) * (x()-tX) + (nY-tY) * (y()-tY))/
            nextDist || -999999 : nextScore"
 }]}, ...// Other visual encodings
```

Fig. 5: Implementing the DimpVis interaction technique. (a) Two snapshots of this interaction; (b) Libra specification (custom components in orange); and (c) communication between all components, where the orange ones are defined by **ID**. (d) D3 code requiring **ID** to provide callback functions for all three events. (e) Vega specification for finding the point nearest to the point dragged on the time trajectory.

4.4 Creating a new Interaction

To demonstrate the generative power of Libra, we present a novel combination of Dust & Magnet [12] and excentric labeling [2]. This combination enhances data exploration by enabling detailed inspection of point clusters while maintaining the force-based exploration mechanism.

To implement this combined technique, **ID** first defines four main layers: the magnet layer showing attributes as colored rectangles that can be dragged, the dust layer displaying all data items as points that can be hovered, the background layer for creating new magnets (all from Dust & Magnet), and a new label layer for showing arranged labels (from excentric labeling). She then associates the magnet layer with the drag instrument and implements three custom services: the magnet position service, dust layout service, and label layout service.

The magnet position service computes the positions of magnets based on user's drag actions, while the dust layout service calculates new positions of data points based on their attraction to magnets. These positions are shared with respective layers' graphical transformers to update the visualization. For the excentric labeling functionality, she attaches a hover instrument to the dust layer and implements a label layout service that arranges non-overlapping labels for points near the cursor.

The communication between these objects follows a clear pattern: when a drag action occurs, it triggers the magnet position service and dust layout service to compute new positions. Simultaneously, when the hover action is detected, the label layout service activates to generate and position labels for nearby points. The separation of services ensures that the drag and hover interactions do not conflict, allowing users to inspect point details even during magnet manipulation.

The interaction flow proceeds as follows: users can drag magnets to attract or repel data points, with the dust layout service continuously updating point positions based on magnetic forces. When users hover over regions containing multiple points, the label layout service automatically arranges and displays labels showing detailed information about nearby points. These labels update dynamically as users move their cursor, providing immediate feedback about the local data distribution. Users can also create new magnets by clicking on the background layer, which triggers the magnet position service to initialize the new magnet's position and the dust layout service to recalculate point positions.

4.5 Diverse Interaction Techniques

In this section, we show several diverse interaction techniques implemented with `Libra.js` and their corresponding interaction taxonmies in Table 1.

Cross-filtering. The cross-filtering technique [4, 8, 11] allows users to dynamically filter and explore multidimensional data through multiple coordinated visualizations. As shown in Figure 7a, selecting a region in one view highlights the data points within that region and updates the other views to show the selected points. **AD** can also implement this interaction technique by reusing the built-in brush instrument. She first separates this chart into one scatterplot layer and four histogram layers where each layer has its own associated data fields. As illustrated in Figure 7b, she inherits the brush instrument (line 2) and then associates this instrument with all scatterplot and histogram layers (line 3). Since each layer shares the same processing pipeline but uses different data fields, she sets the selection service to use different data fields with a function defined in lines 7-13, which uses the layer's index to find the corresponding data fields (line 9). Subsequently, she inserts a filter service (line 12) to update the data rendered in each layer.

Overview & Detail. As shown in Figure 8a, the overview and detail technique [3] combines two linked visualizations: a high-level overview of the entire dataset and a detailed view of a specific subset of the data. Users can navigate and explore the data by selecting regions in the overview, which are then magnified and displayed in the detail view. To implement this, **AD** first separates the visualization into two layers: the overview layer and the detail layer, as illustrated in Figure 8b. She then associates the overview layer with the brush instrument and inserts a scale service to

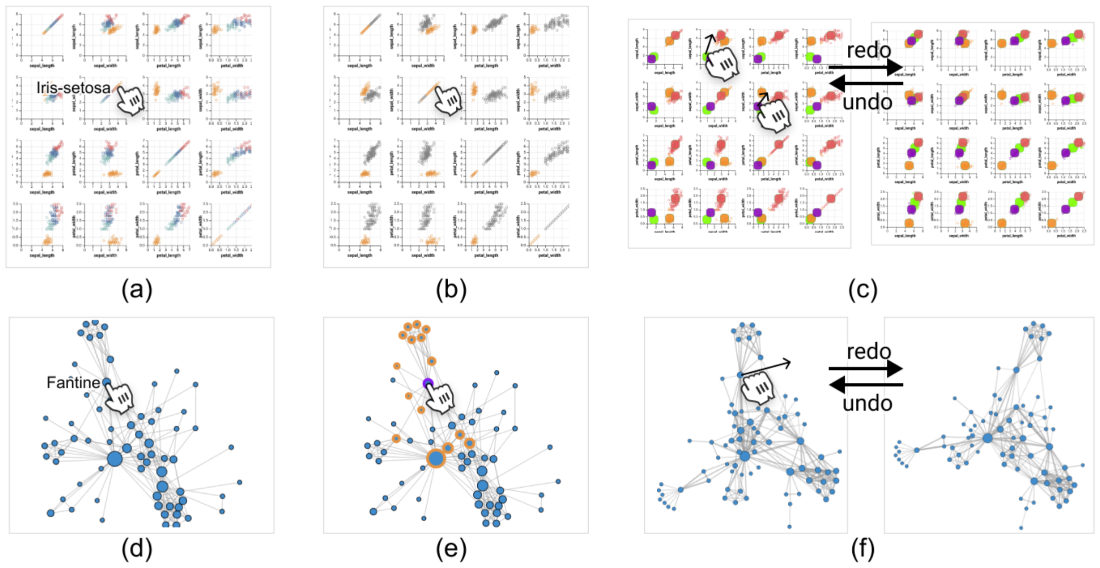


Fig. 6: Implement the same interaction illustrated in the teaser of the main paper to (a-c) the scatterplot matrix, and (d-f) the node-link diagram.

Table 1: The relationship between each visualization and its respective interaction category. The symbols ✓ and × indicate the inclusion or exclusion of an interaction within a given category, respectively.

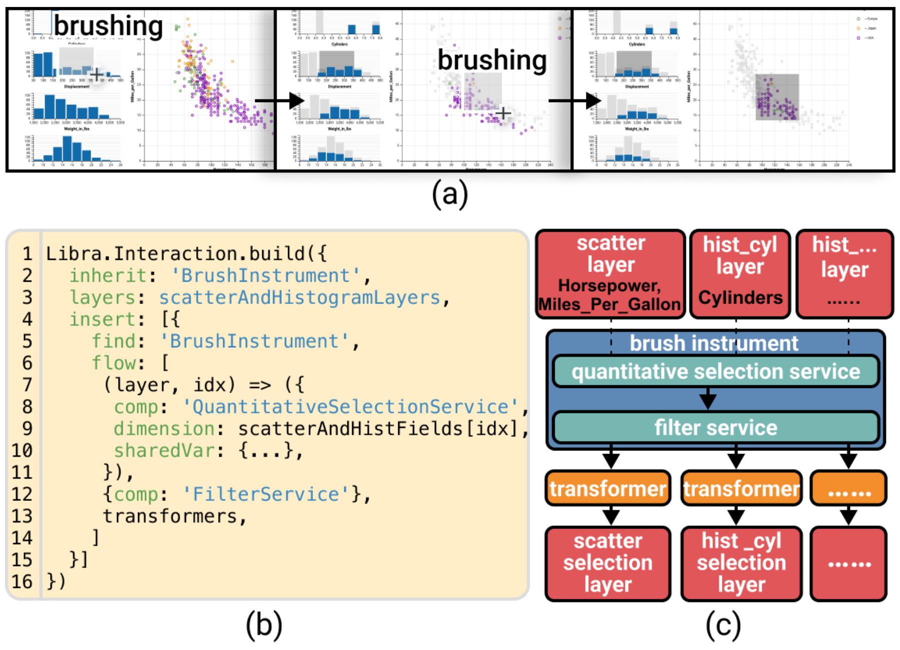
	Select	Explore	Reconfigure	Encode	Abstract/Elaborate	Filter	Connect
Hover on MNIST dataset	✓	×	×	×	✓	×	×
Click on MNIST dataset	✓	×	×	✓	×	✓	×
Interactive k-means on MNIST dataset	✓	×	×	✓	×	×	✓
Excentric labeling	✓	×	×	✓	✓	✓	×
Hover on bar chart	✓	×	×	×	✓	×	×
Multiple click on scatter plot	✓	×	×	✓	×	×	×
Brush on scatter plot	✓	×	×	✓	×	×	×
Pan & Zoom	×	✓	×	×	✓	×	×
Index chart	×	×	✓	×	×	×	×
DimpVis	✓	✓	×	×	✓	✓	✓
Figure 7	✓	×	×	✓	×	✓	✓
Figure 8	✓	✓	×	×	×	×	✓
Figure 9	✓	×	✓	×	×	×	×
Figure 10	✓	×	✓	×	✓	×	×
Figure 11	✓	×	×	✓	×	×	✓
Figure 12	✓	×	×	×	×	✓	✓

compute the corresponding scaling factor based on the selected region. This scaling factor is shared with the detail layer's graphical transformer to update the detailed view accordingly. Figure 8c shows the communication between the objects, where the brush action triggers the selection service and the scaling service, and the detail transformer to update the detail view.

Matrix Reordering. Matrix reordering [7] is an interaction technique that allows users to rearrange the rows and columns of a matrix visualization to reveal patterns and relationships in the data, as shown in Figure 9a. Users can interactively drag and drop rows and columns to reorder the matrix. To support this interaction, **AD** associates the main layer with the drag instrument, as shown in Figure 9b. She then inserts the scale inversion service that finds the original and new row or column indices based on the user's drag action. Those indices are shared with the reorder service to sort the data, and then the main layer's graphical transformer will re-draw the sorted matrix. Figure 9c illustrates the communication flow, where the drag action triggers the scale inversion service to compute the new indices, which are then used to reorder and redraw the matrix.

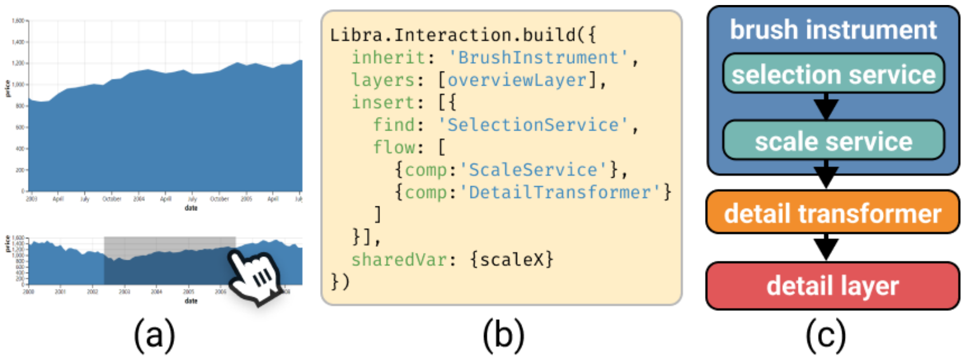
Dust & Magnet. The Dust & Magnet technique [12] enables users to explore and analyze multidimensional data by directly manipulating data points (dust) and attributes (magnets) on a two-dimensional screen space, as shown in Figure 10a. Users can place magnets as data attributes by clicking the empty region on the screen, and observe the movement of data points as they are attracted by these magnets based on their attribute values. To implement this technique, **ID** defines two main layers: the dust layer and the magnet layer. As illustrated in Figure 10b, she associates the magnet layer with the drag instrument and implements two custom services: the magnet position service and the dust layout service. The magnet position service computes the positions of the magnets based on the user's drag actions, while the dust layout service calculates the new positions of the data points based on their attraction to the magnets. These new positions are shared with the respective layers' graphical transformers to update the visualization. The definition of clicking is similar to the one of dragging. She also attached a hover instrument to the dust layer to highlight her interest points. Figure 10c shows the communication between the objects, where the drag action or the click action triggers the magnet position service and the dust layout service to compute the new positions, while the hover action is not conflicted with the dragging and clicking.

Interactive K-Means. Figure 11a shows an example of an interactive k-means clustering instrument applied to a multidimensional dataset visualized as a scatterplot matrix. Here, a user can interactively move a cluster centroid if she believes that it is misplaced. This triggers new



```
Libra.Interaction.build({
  inherit: 'BrushInstrument',
  layers: scatterAndHistogramLayers,
  insert: [{
    find: 'BrushInstrument',
    flow: [
     (layer, idx) => ({
      comp: 'QuantitativeSelectionService',
      dimension: scatterAndHistFields[idx],
      sharedVar: {...},
     }),
     {comp: 'FilterService'},
     transformers,
    ]
  }]
})
```

Fig. 7: Implementing a cross-filtering instrument for the coordinated scatterplot and four histograms. (a) Snapshots of the interaction; (b) the `Libra.js` specification that attaches the scatterplot layer to the brush instrument with filtering service, and (c) the objects involved and the communication between them.



```
Libra.Interaction.build({
  inherit: 'BrushInstrument',
  layers: [overviewLayer],
  insert: [{
    find: 'SelectionService',
    flow: [
      {comp:'ScaleService'},
      {comp:'DetailTransformer'}
    ]
  }],
  sharedVar: {scaleX}
})
```

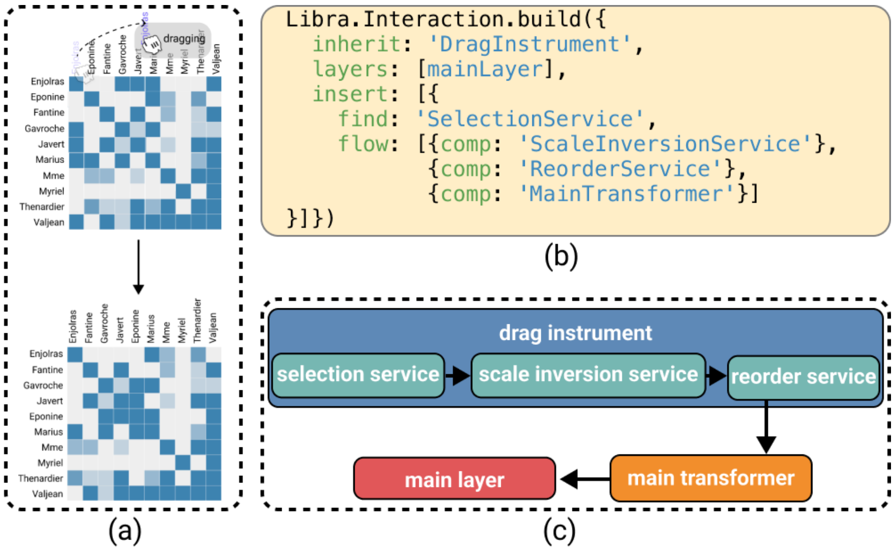
Fig. 8: Implementing an overview and detail instrument. (a) The snapshots of the interaction, showcasing the overview and the corresponding detailed view; (b) the `Libra.js` specification that attaches the overview layer to the brush instrument with the scale service, and (c) the involved objects and communication between them.

iterations of the k-means algorithm starting from the specified configuration, leading to a possible better solution. With a few interactions, the user can check if the clustering is stable or fix it. For example, the green and purple centroids are moved by the user on the left of Figure 11a, and then a new clustering result is quickly obtained, as shown in the right of Figure 11a.

Figure 11b provides the core part of the `Libra.js` specification. Given a scatterplot matrix, **ID** needs to define two main layers: a centroid layer and a cluster layer. Then, she associates the centroid layer with the drag instrument to perform interactive k-means clustering. It is done in three steps: i) using the selection service to find the changed centroid, ii) joining this centroid with other unmoved centroids (line 7), and iii) sharing all centroids with the k-means service for generating new clustering results (line 11). During the dragging action, the intermediate clustering results are shown in the transient layer. To quickly generate such a preview, the k-means clustering is executed with a small number of iterations. Once the centroid is dropped, the iterations continue to completion. Figure 11c shows the communication between these objects, where the drag action triggers the centroid selection and then the k-means service for updating the clustering result. To persist the clustering result, the command is executed to update the centroid data while re-drawing the layer consisting of all points after the clustering is computed. Since the k-means service is independent of the main visualization, this instrument can be extended with many other ML algorithms for supporting interactive visual analysis.

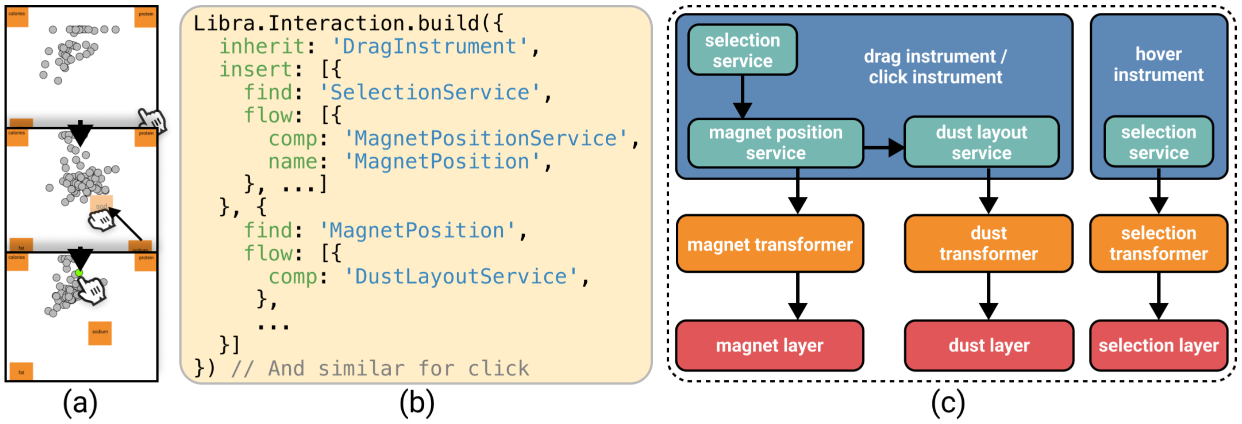
Dynamic Query. All the above interactions use direct manipulation of visualization objects. Yet, indirect manipulation is also required for many interactions. For example, dynamic queries [6] are a classic interaction technique for selecting data items via moving a slider widget. By taking the widget as an individual layer, `Libra.js` can handle it as the multiple-view visualization. Figure 12 shows an example where a drag instrument is associated with the widget layer, and the result of the selection service is passed to the graphical transformer for updating the main layer. Based on the layer representation, `Libra.js` unifies the interaction specifications for direct and indirect manipulation interfaces.

Furthermore, we have provided detailed technique documentation and have created a collection of examples that demonstrate the capabilities of Libra for reusing, compositing, and extending, shown in the following website:



```
Libra.Interaction.build({
  inherit: 'DragInstrument',
  layers: [mainLayer],
  insert: [{
    find: 'SelectionService',
    flow: [{comp: 'ScaleInversionService'},
           {comp: 'ReorderService'},
           {comp: 'MainTransformer'}]
}]})
```

Fig. 9: Implementing a matrix reordering instrument. (a) The snapshots of the interaction, displaying the initial state, user selection, and reordered matrix; (b) the `Libra.js` specification that attaches the matrix layer to the drag instrument with reordering service, and (c) the involved objects and communication between them.



```
Libra.Interaction.build({
  inherit: 'DragInstrument',
  insert: [{
    find: 'SelectionService',
    flow: [{
      comp: 'MagnetPositionService',
      name: 'MagnetPosition',
    }, ...]
  }, {
    find: 'MagnetPosition',
    flow: [{
      comp: 'DustLayoutService',
     },
     ...
  }]
}) // And similar for click
```

Fig. 10: Implementing the Dust & Magnet interactive visualization system. (a) The snapshots of the interaction, displaying the initial state, user manipulation of magnets, and the resulting movement of data points (dust); (b) the `Libra.js` specification that attaches the magnet layers to the drag instruments with corresponding services, and (c) the involved objects and communication between them.

https://libra-js.github.io/.

We encourage readers to play these examples to gain a deeper understanding of our model Libra.

5 Additional Metrics-Based Analysis

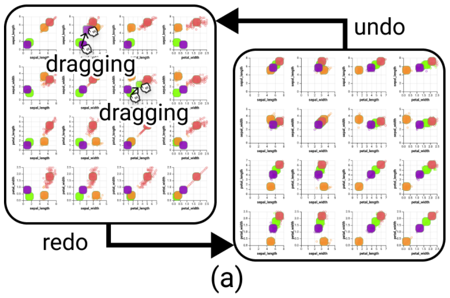
When examining only the interaction-related code (Figure 13), we observe a significant reduction in all metrics across libraries. Notably, `Libra.js`-D3 and `Libra.js`-Vega show particularly consistent performance across different interaction complexities, with similar values for sprawl and specification length. This consistency arises from `Libra.js` using a shared interaction model across libraries, with only library-specific code differing (e.g., updating the main layer). However, this also results in `Libra.js`-D3 having a slightly larger sprawl than `Libra.js`-Vega.

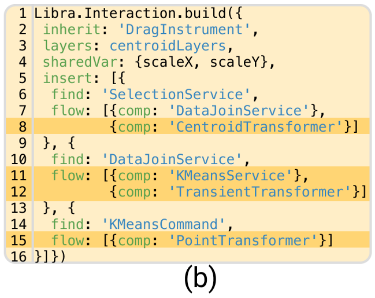
In contrast, D3 exhibits significant variations in vocabulary size, largely attributed to its approach of predefining many commonly used interactions. While this design simplifies the implementation of standard interactions, it places a substantial burden on developers for new or more complex interactions. Developers are required to handle low-level event management, state transitions, and interaction feedback manually, which often results in verbose and less maintainable code. This manual effort can also lead to inconsistencies in how interactions are implemented across different applications. Vega, on the other hand, shows notable variations in specification length and sprawl. These variations stem from the tight coupling between Vega's signals, event streams, and visual representations. While this coupling allows for efficient specification of straightforward interactions, it increases complexity for more intricate or customized interaction designs. The dependency on a specific dataflow model makes extending or reusing interactions more challenging, particularly for applications requiring a high degree of customization.

References

[1] DimpVis: Prototyping for direct interaction techniques with information visualizations. https://github.com/vialab/dimpVis, 2013. Accessed: 2023-03-14. 3

[2] J.-D. Fekete and C. Plaisant. Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 512–519, 1999. doi: 10.1145/302979.303148 4





```
Libra.Interaction.build({
 inherit: 'DragInstrument',
 layers: centroidLayers,
 sharedVar: {scaleX, scaleY},
 insert: [{
  find: 'SelectionService',
  flow: [{comp: 'DataJoinService'},
         {comp: 'CentroidTransformer'}]
 }, {
  find: 'DataJoinService',
  flow: [{comp: 'KMeansService'},
         {comp: 'TransientTransformer'}]
 }, {
  find: 'KMeansCommand',
  flow: [{comp: 'PointTransformer'}]
}]})
```

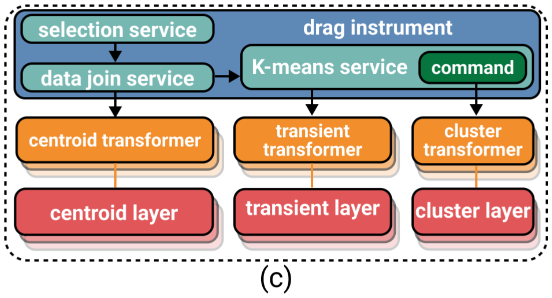
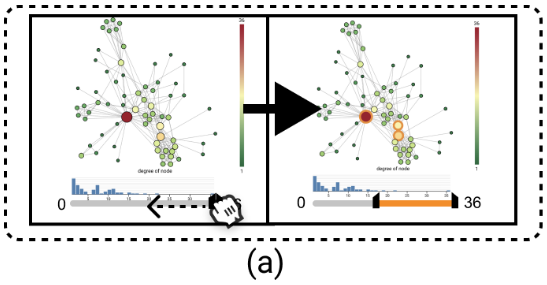
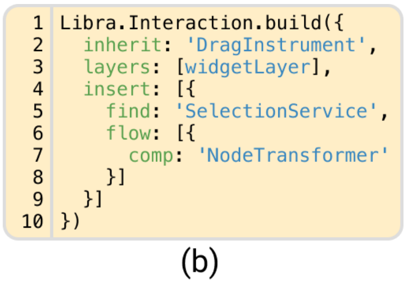


Fig. 11: Interactive k-means clustering of multidimensional data shown on a scatterplot matrix. (a) After dragging the purple and green centroids (left), the new clustering result is generated (right); (b) the `Libra.js` specification for this interaction where the custom components are shown in orange; and (c) The communications between all objects.





```
Libra.Interaction.build({
  inherit: 'DragInstrument',
  layers: [widgetLayer],
  insert: [{
    find: 'SelectionService',
    flow: [{
      comp: 'NodeTransformer'
    }]
  }]
})
```

Fig. 12: Selecting nodes in a node-link diagram with a widget-based dynamic query. (a) The snapshots for showing the results of a range query (bottom) and the selected nodes are highlighted in orange (top); (b) The `Libra.js` specification where the selection service on the widget layer updates the main node layer.

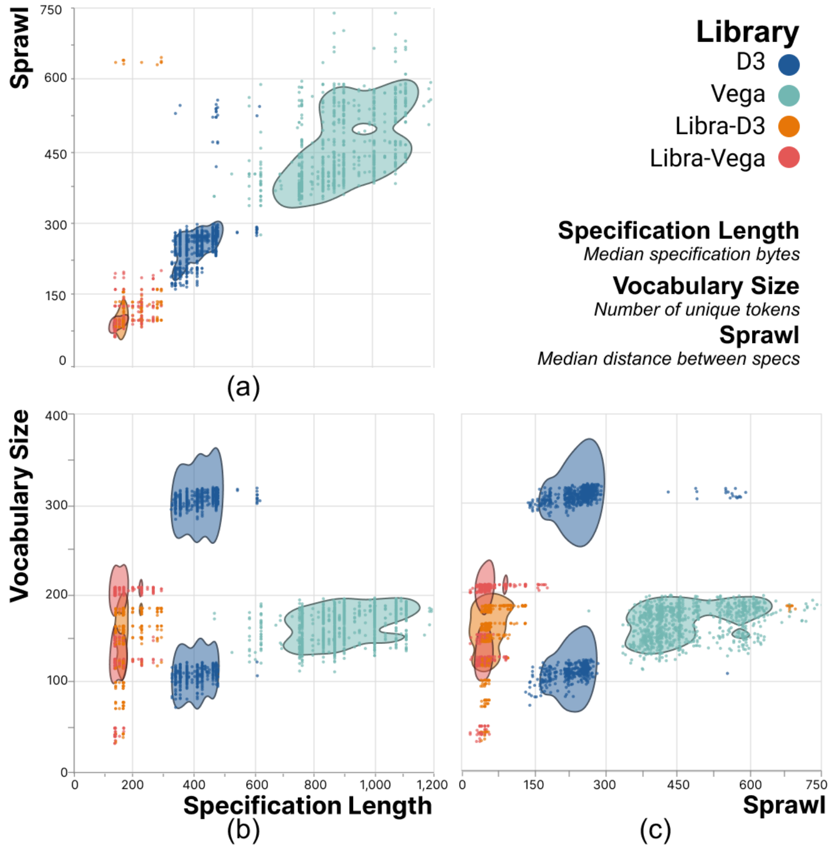


Fig. 13: We run 1k bootstrapped variations for each metric based on our gallery without the static visualization code and show the results via a scatter plot with a Kernel Density Estimation (KDE) of each pair of metrics: specification length, vocabulary size, and sprawl. Each scatter represents the medium value of one variation, while the shaded regions represent the areas containing 75% of the probability mass of each library's KDE distribution.

[3] W. Javed and N. Elmqvist. Stack zooming for multi-focus interaction in time-series data visualization. In *2010 IEEE Pacific Visualization Symposium*

(PacificVis), pp. 33–40. IEEE, 2010. 4

[4] A. Martin and M. Ward. High dimensional brushing for interactive exploration of multivariate data. In *Proceedings Visualization'95*, p. 271. IEEE, 1995. 4

[5] MDN. Web Speech API - Web APIs | MDN. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API`. Accessed: 2024-03-23. 2

[6] O. OCAL. Dynamic queries for visual information seeking dynamic queries let users" fly through. *Readings in Information Visualization: Using Vision to Think*, p. 235, 1999. 6

[7] H. Siirtola. Interaction with the reorderable matrix. In *1999 IEEE International Conference on Information Visualization (Cat. No. PR00210)*, pp. 272–277. IEEE, 1999. 5

[8] I. Square. Crossfilter: Fast multidimensional filtering for coordinated views, 2013. 4

[9] Global Development Example. `https://vega.github.io/vega/examples/global-development/`, 2018. Accessed: 2023-03-14. 3

[10] An interactive scatter plot of global health statistics by country and year. `https://vega.github.io/vega-lite/examples/interactive_global_development.html`, 2020. Accessed: 2023-03-14. 3

[11] C. Weaver. Cross-filtered views for multidimensional visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):192–204, 2009. 4

[12] J. S. Yi, R. Melton, J. T. Stasko, and J. A. Jacko. Dust & Magnet: multivariate information visualization using a magnet metaphor. *Information Visualization*, 4(3):239–256, 2005. doi: 10.1057/palgrave.ivs.9500099 4, 5