Beagle: Automated Extraction and Interpretation of Visualizations from the Web

by

Peitong Duan

S.B., Computer Science and Engineering, M.I.T., 2016

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author . . . . . .
Department of Electrical Engineering and Computer Science
January 31, 2017

Certified by . . . . . .
Michael Stonebraker
Adjunct Professor
Thesis Supervisor

Accepted by . . . . . .
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Abstract

In this paper, we present Beagle, an automated data collection system to mine the web for SVG-based visualization images, label them with their corresponding visualization type (*i.e.,* bar, scatter, pie, etc.), and make them available as a queryable data store. The key idea behind Beagle is a new SVG-based classification design to more effectively label visualizations rendered in a browser. Furthermore, Beagle is designed from the ground up to be extendable and modifiable in a straightforward way, to anticipate when new artifacts appear on the web, such as new JavaScript libraries, new visualization types, and better browser support for SVG. We evaluated Beagle's classification techniques on multiple collections of SVG-based visualizations extracted from the web, and found that Beagle provides a significant boost in accuracy compared to existing classification techniques, across a wide variety of visualization types.

0.1 Introduction

Data science has become important in an increasing number of domains, including web design, HCI, and visualization. For example, Kumar *et al.* recently developed the Bricolage algorithm [23], which uses a collection of web pages to learn mappings between different web designs, and uses these mappings to restyle web pages automatically. Reinecke *et al.* build computational models over website snapshots to predict how the visual complexity and colorfulness of a web page will affect a visitor's initial impression of the web page [28]. Harrison *et al.* propose a similar approach to predict first impressions of infographics [18]. In a related area, Saleh *et al.* compute low-level visual features (*e.g.,* color histograms) over a collection of infographics to measure style similarity between infographics [30].

While effective, it is clear that data-driven design systems consistently require a large input collection from which to build design rules and train machine learning models. For these systems to achieve high prediction accuracy, their input collections often contain tens to hundreds of thousands of examples, making manual gathering and tagging prohibitively time consuming to perform. As such, developing automated techniques for collecting and classifying these collections is of the utmost importance. However, existing work in this area focuses mainly on collecting and classifying web designs [22, 29, 19, 6]. Web designs style and format webpages, making them different from visualization designs that focus on visualizations alone. Little work has been done to support the automated collection and classification of visualization designs.

Furthermore, existing techniques for collecting and classifying web pages are unable to discern visualizations from other images (*e.g.,* logos), and have no notion of categorizing designs by type. Unlike web page designs, most visualization designs belong to well-known categories (or types), such as bar and pie charts. Therefore, the most important feature for an automated visualization collection and classification system is the ability to accurately label each visualization with its corresponding type. Without this information, visualization designers that want to use the collection will have no idea how many examples of each visualization type are available, or which types are represented, rendering the collection ineffective. For instance, an unlabelled collection would be useless if a user wants to study design examples of a certain visualization type. Such a collection would also be ineffective for training visualization design systems, as designs are based heavily on visualization type.

In this paper, we present Beagle, an automated system to extract visualizations rendered in a browser, label them, and make them available as a query-able data store. Beagle consists of three major components: a Web Crawler for identifying and extracting SVG-based visualizations from webpages, an Annotator for automatically labeling extracted visualizations with their corresponding visualization type, and a Data Store that organizes the visualizations into a relational format for storage in a database management system (or DBMS). The key idea behind Beagle is a new SVG-based classification design to more effectively label visualizations from the web that were rendered using SVG. Furthermore, Beagle is designed from the ground up to be extendible and modifiable in a straightforward way. Thus Beagle is designed to anticipate when new artifacts appear on the web, such as new JavaScript libraries, new visualization types, and more browser support for SVG. With Beagle, we make the following research contributions:

Identification and Extraction of SVG-based Visualizations: Existing work relies on manual gathering or other non-generalizable tricks to build large-scale visualization collections [30, 8, 31], and no techniques exist to create these collections automatically. The Beagle Web Crawler demonstrates how automated processes can be used to collect over 17000 visualizations from the web.

Flexible visualization Classification: Beagle's Annotator automatically labels each visualization with its corresponding visualization type. To do this, Beagle uses a new SVG-based classifier that calculates statistics over the graphical marks specified inside of an SVG object (*e.g.,* rectangles, lines and circles) to use as classification features. We found that our new SVG-based classifier can correctly classify our visualization collection with 82% accuracy. This represents a 20% boost in accuracy compared to existing techniques [31], in a multi-class classification test across 22 different visualization types.

General-Purpose Data Store: In addition to classifying the visualizations, we recognize that a large visualization collection is most useful when the designer can efficiently search for visualizations with certain characteristics. For example, designers may want to search for specific visualization types (*e.g.,* all bar charts), graphical marks (*e.g.,* all visualizations containing circles), or style properties (*e.g.,* all visualizations containing more than one fill color). We aim to provide a well-structured data analysis layer that supports a variety of search filters over thousands of visualizations. To do this, Beagle stores the dataset created by the Web Crawler and Annotator in a database. We designed a specialized relational schema to support a variety of queries over the dataset using a well-known query language (SQL). The schema is easily extendable to anticipate new visualization types and better SVG support in the future.

0.2 SVG Overview

Beagle's ability to automatically collect and label visualizations relies on identifying and analyzing SVG images embedded within a web page. Here, we provide a brief explanation of what SVG is, why we use it, and the specific properties that make it useful for Beagle.

0.2.1 What is SVG?

SVG [16] is an image format for two-dimensional graphics. SVG is XML-based, making it straightforward to analyze with an XML parser. Though SVG is also supported outside the web, we study the use case of running JavaScript code to generate and embed SVG images within a web page. An embedded SVG image is stored as a node within the Document Object Model (or DOM) tree of a web page as an SVG *object*, denoted by the <svg> tag. SVG objects behave like other objects in the DOM tree, such as <body> and objects, making SVG objects straightforward to locate within and extract from a web page.

0.2.2 Why Use SVG?

We focus on SVG for two reasons: 1) direct access to graphical marks, and 2) access to the code used to render the SVG visualization. Unlike raster images, SVG describes in plain text the graphical marks used to render an image, enabling Beagle to classify a large number of visualizations using a small, concise set of SVG elements. Furthermore, SVG-based visualizations are often created by JavaScript code located directly within the same web page (*e.g.,* D3.js code [10]), making SVG-based visualizations easier to find on the web, compared to raster-based visualizations.

0.2.3 How do We Use SVG?

In our classification analyses, we focus on the graphical marks, or shapes that are displayed within an SVG-based visualization. Specifically, we analyze five types of SVG marks: lines, circles, rectangles, text, and paths. Each of these marks is called an SVG *element*. Each SVG element has its own HTML tag: <line>, <circle>, <rect>, <text>, and <path>. Figure 0-1 contains a box plot and line chart illustrating uses of all five of these SVG element types. Each element type (*i.e.,* `rect`, `circle`, etc.) is associated with specific properties that define how the element should be rendered, such as width, height, radius, and positioning (*i.e.,* x and y). We refer to these properties as *attributes*.

Graphical marks can also be given additional styling properties, such as `fill` (*i.e.,* fill color) and `stroke-width` (*i.e.,* border width), which control the aesthetics of the element. We call these *style attributes*. We analyze four style attributes in Beagle: `fill`, `stroke`, `stroke-width`, and `font-size`.

0.3 System Design Overview

Beagle consists of three major components, illustrated in Figure 0-2: a Web Crawler to find and extract visualizations from web pages, an Annotator to automatically classify visualizations with their corresponding visualization types, and a Data Store to store the extracted visualizations as a relational database. The rest of this section outlines each of the three components within Beagle.

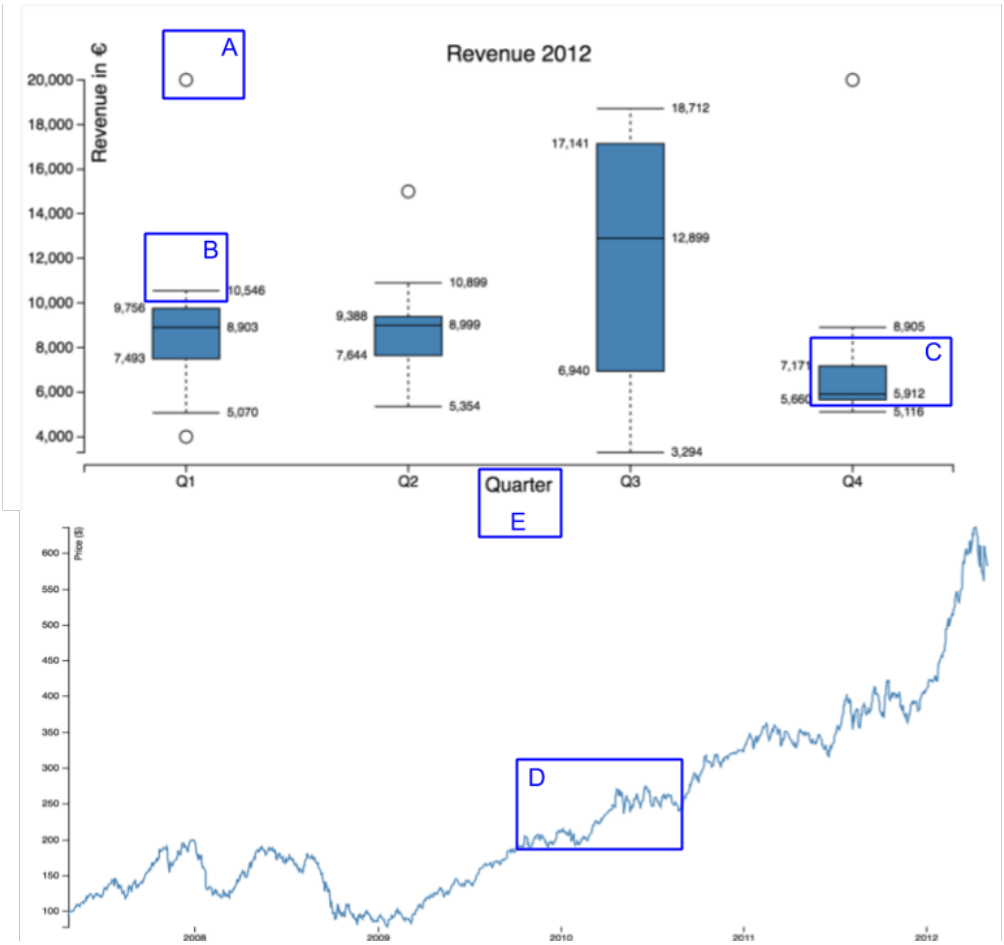

Figure 0-1: A box plot and line chart illustrating uses of all SVG element tables. The element in box A is a <circle> element, box B contains a <line>, box C contains a <rect>, box D contains a <path>, and box E contains a <text>.

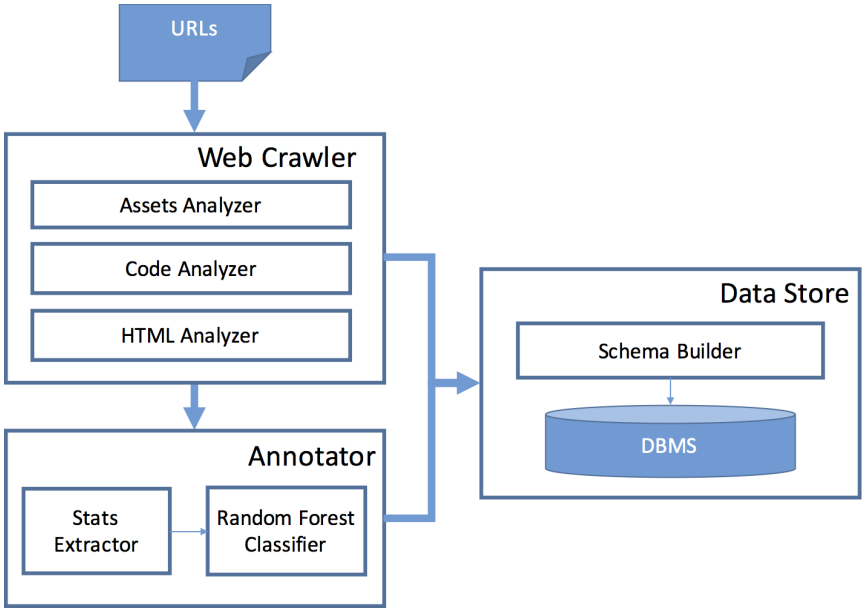

Figure 0-2: Overview of the Beagle architecture.

0.3.1 Web Crawler

The Web Crawler is responsible for finding web pages containing visualizations, and extracting these visualizations for later classification by the Annotator and storage in the Data Store. It takes a list of urls as input, and produces a visualization ID's file and a set of folders as output. The visualization ID's file contains: a unique ID for each extracted visualization, and the url of the web page containing the visualization. Each folder corresponds to a single visualization, and contains several files: the extracted SVG object for the visualization, a snapshot (*i.e.,* raster image) of the visualization, and the raw HTML for the web page containing the visualization. Given that web pages do not need to be visited in any particular order, the filtering process is easy to parallelize. To take advantage of this, the Web Crawler is designed to run in parallel across multiple processors and multiple machines. We detail the design of our Web Crawler in the Section "Finding and Extracting Visualizations".

0.3.2 Annotator

The Annotator manages the automatic classification of visualizations with their corresponding types. It utilizes a new SVG-based classifier that computes statistics over the elements within a SVG-based visualization. The Annotator uses these statistics to identify patterns in the frequency, positioning, sizing, and styling of SVG elements for different visualization types. We detail the design of our SVG-based classifier in the Section "Automatically Labeling Visualizations".

0.3.3 Data Store

The Data Store consists of a Schema Builder to translate the data from the other two components into a relational data format, and a DBMS for storing the resulting database. Storing the data as a database allows for fast data processing using the DBMS, and provides access to a well-known query language for interacting with the data (SQL). We explain the design and usage of our Data Store in the Section “Storing and Querying Visualizations”.

0.4 Finding and Extracting Visualizations

The aim of our Web Crawler is to find and extract a diverse set of SVG-based visualizations from the web. Given that our Web Crawler only takes a list of urls of input, an important part of the data collection process is to identify potential url’s to visit. In this section, we describe our initial url collection approach of randomly sampling url’s from the web, and how the Web Crawler was used to filter the resulting urls to identify relevant web pages. We then explain our final technique, where we collect thousands of visualizations from visualization-focused websites.

0.4.1 General Web Search

Our original url collection approach was to randomly sample urls from the web, and check the corresponding web pages for SVG-based visualizations. This approach enabled a broad search across visualization tools and web domains, providing an unbiased view of SVG usage across the web. We used the url index of the Common Crawl [11] as our source for urls. The Common Crawl is project that scraped the raw webpage data of every site on the web, and they update this data on a yearly basis. The Common Crawl data is indexed by webpage urls and other domain information, which means the url index contains almost every url on the web. All this data is publicly available, and we obtained our url collections by randomly sampling from from this url index.

Web Filtering Process

Given a set of urls selected from the Common Crawl, we must filter for pages that contain SVG visualizations. While the most direct method for detecting SVG-based visualizations is to search the HTML DOM tree for SVG objects, this process turns out to be prohibitively time consuming. Most modern SVG-based visualizations are generated using higher-level libraries such as D3.js [10], Plotly [25], and Raphael [7]. As such, a "superficial” search of an HTML page will not be able to find the desired SVG objects. Instead, such a web page needs to be "executed” by a browser to produce the SVG elements. Unfortunately, this code execution process can be extremely slow when our goal is to crawl and search through the entire internet. In response, we apply three-stage heuristics to speed up the search process: an Assets Analyzer, a Code Analyzer, and an HTML analyzer.

Assets Analyzer

This component is responsible for detecting visualization JavaScript libraries used within a particular web page. The Assets Analyzer also supports D3.js [10], Plotly [25],Raphael [7], Dygraphs [2], JIT [1], Dimple [4], Springy [3], and Sigma [5]. The Assets Analyzer takes a list of strings representing the names of the JavaScript libraries to search for. For a given web page, the Assets Analyzer inspects the list of assets to be downloaded by the web page, searching for the designated library names. Any relevant pages are then passed on to the Code Analyzer.

Code Analyzer

This component is responsible for searching for snippets of JavaScript code that could be used to generate an SVG-based visualization in the browser and detect whether or not any of the SVG-based visualizations are animated. Knowing whether a visualization is animated is important for the HTML Analyzer, and is explained in the next section. The Code Analyzer requires a list of visualization code snippets for each JavaScript library of interest. Each entry in the code snippet list is itself a sublist containing one or more strings. Some example snippet strings for D3 are “d3.select(”, “.data(”, and “.enter(” to detect SVG-base visualizations and "d3.timer", "d3.transition", and "setInterval" for to find animated visualizations. Since SVG charts are usually created with multiple function calls, various code snippets must co-occur in order for a visualization to be created using this particular JavaScript library. Thus, the code snippets are grouped so that each sublist of strings represents code snippets that must co-occur. The Code Analyzer downloads all JavaScript code requested by the web page, as well as the raw HTML of the page. It then searches each of these files for the presence of visualization code snippets. Any pages containing visualization code are then passed to the HTML Analyzer.

HTML Analyzer

This component is responsible for finding and extracting SVG objects from a web page. However, given that our Web Crawler is searching for JavaScript libraries that dynamically generate SVG in the browser, we have to actually run a browser before we can extract any SVG objects, otherwise the SVG will not be rendered in the page. To do this, we run the Selenium Web Driver [27] with Firefox [14], visit each candidate page in the Web Driver, execute the JavaScript code in the HTML, then search the DOM tree for SVG objects. If the page is indicated to have animations, the scraper will wait a few seconds before collecting the SVG element because the graphical elements in an animated SVG chart may take some time to appear or assemble into a chart.

General Web Crawler Results

We performed an analysis on the results of the General Web Crawl, and this study includes visualization-related statistics about the general web (e.g. what percentage of pages that used a chart-making library actually built an SVG chart) and observations on how people visualize data and utilize chart libraries on the web. This analysis can be found in the appendix. We found with the General Web search that only a small fraction of

websites actually use JavaScript visualization libraries. As such, we were only able to find a small set of visualizations to study (*i.e.,* less than 1200 samples). This motivated us to switch to an approach that utilized visualization-rich websites as sources for urls, and this method is detailed in the next section.

0.4.2 Targeted Web Search

In this url collection approach, we sought out specific "islands" of usage on the web, where users frequently deposit their data and visualizations at a single, centralized source website. We performed a broad search for islands that consistently contained SVG-based visualizations, investigating sites such as Tableau Public [33], Many Eyes [34], and bl.ocks.org [9] for D3 examples [10]. Ignoring the islands that use raster formats instead of SVG, we chose the two biggest islands to visit using our Web Crawler: bl.ocks.org, and the Plotly website [25].

Targeted Web Search URL Collection and Visualization Extraction

Plotly and D3 both had individual pages for each one of their charts, and we searched for the urls of all these chart pages to feed into the SVG extraction pipeline described in Section 0.4.1. For Plotly, urls to individual chart pages have the format "https://plot.ly/ <username>/<chart id>", where "<username>" is the user who created the chart, and "<chart id>" is the unique id assigned to the chart. We first utilized the Plotly Feed page ("https://plot.ly/feed/"), which contains a vast number of example charts along with the usernames of their creators, to garner a large collection of usernames. We then visited each user's profiles at "https://plot.ly/<username>", which lists all the charts created by the user. We collected the chart id's of all these charts from the user's profile to construct the urls for individual chart pages. For D3, we utilized the bl.ocksplorer search engine ("http://bl.ocksplorer.org/") which takes D3 API calls as input and returns all D3 visualizations that uses the API call. Examples of D3 API calls include "d3.layout.histogram", which is used to create histograms, and "d3.svg.area" used to create area charts. We compiled a list of such D3 API calls, programmatically searched for them, and collected the visualization urls of all the search results. We filtered out duplicate urls. Since url collection for both sites required browser interaction, we utilized the Selenium Webdriver with Firefox to conduct our url search. The Selenium Webdriver works by making calls to the firefox browsers to issue specified browser interactions, such as clicking a button.

We extracted SVG visualizations from these urls using the three-stage process outlined above. However, since we know that urls collected from these two sites call a visualization JavaScript library, we were able to bypass the Assets Analyzer, and ran the urls directly through the Code Analyzer to mark pages with animated visualizations, and then through the HTML Analyzer to extract the visualizations.

Targeted Web Search Results

Using the urls collected from our targeted web search (bl.ocks.org and Plotly), we ran the Beagle Web Crawler two separate times to find and extract SVG-based visualizations. We refer to the resulting visualizations as the

bl.ocks.org collection and the Plotly collection. Given that our islands are websites dedicated to a specific tool (or library), each url collected from our islands was likely to produce SVG-based visualizations, allowing us to collect thousands of visualizations in only a matter of days. For example, the bl.ocks.org crawl took one day to run, and resulted in 2418 SVG-based visualizations extracted for the bl.ocks.org collection. The Plotly crawl took roughly one week to run, and resulted in 15232 SVG-based visualizations extracted for the Plotly collection.

0.5 Automatically Labeling Visualizations

Before designers can use the visualizations extracted by the Web Crawler, they need to know what types of visualizations were extracted. Given that thousands of visualizations can be extracted by the Web Crawler, an automated process is required for efficiently labeling visualizations. In this section, we explain how the Annotator component in Beagle uses a new SVG-based classifier to automatically label extracted visualizations with their corresponding visualization type.

The Annotator uses basic statistics calculated over SVG elements to discern visualization types. Examples of some statistics that we calculate are: the average x position of specific elements (*e.g.,* `rect`'s, `circle`s, etc.), the average number of elements that share the same y position, the average width of specific elements, and the number of unique colors observed in the visualization. Each statistic (*e.g.,* average or count) represents a single classification feature. Our features can easily be computed over any SVG object and are applicable beyond the specific visualization types we observed, allowing for easy extension of our classifier to support more visualization types and SVG element types in the future. These features are then passed as input to an off-the-shelf classification library (Python's Scikit-Learn) to classify and determine the visualization type.

Our feature extraction code computes a large number of features (85 features), because we calculate a suite of statistics over the positions, sizes, and styles for four different SVG element types. These features are individually covered briefly in this section, and a description of each feature can be found in Tables 11, 12, 13, 14, 15, 16, 17 in the appendix.

Our features each belong to one of three groups (feature counts noted in parentheses): general features (9), styling features (19), and per-element features (57). We describe each of these groups and provide intuition for why these features were selected. The general features are used to count the number of times each type of element is observed in a visualization. The styling features are used to calculate statistics for how fill color, border color, border width, and font sizes are manipulated for all elements in the visualization, regardless of element type. The per-element features are used to calculate statistics on the various positions, sizes, and styles observed for each of the four SVG elements we study in Beagle: `circle` (13), `rect` (10), `line` (10), and `path` (24). The classifier is then evaluated using only the features in each group, and the accuracy from each group reflects its significance. Furthermore, each group is broken down into smaller subgroups of more closely-related features, and each subgroup is evaluated for a more in-depth analysis. For the rest of this section, we explain how we calculate the features in each of our feature groups and evaluate the accuracy of

each feature group and corresponding subgroups. The accuracy results showed that nearly all feature groups perform better on Plotly data. The only exceptions are for features that don't apply to Plotly. These exceptions are discussed in their respective subsections. Plotly performed better than D3 because there are twice as many D3 visualization types than Plotly visualization types (22 vs 11 as shown in Table 8), and additional reasons for Plotly's higher accuracy are included in their respective subsections.

We originally created 119 features, but found that many were redundant. For example, providing ratios of circle and `line` counts is considered redundant when given counts of both circle and `line` counts, as the ratios can be inferred from from the count values. We decided to remove the redundant features for several reasons. First, an excess of features can lead to overfitting. Another motivation is to minimize redundancy across features. The last and most important motivator is to avoid over engineering the classifier, because this limits the generalization of the classifier to other vis libraries. To trim our feature set, we identified the best performing sub groups of features, and removed the rest. All features (including the eliminated features) are described in Tables 11, 12, 13, 14, 15, 16, 17 in the appendix as mentioned above.

0.5.1 General features

This section describes the implementation and classification accuracy for general features. Each group of features is itself subdivided into subgroups of related features. "Implementation" describes the intuition behind general features and provides pseudocode for extracting certain features, and "Classification Accuracy" presents the results of the classifier evaluated using only features from subgroups as well as features from entire group, and an analysis of these results. All subsequent feature group sections follow this format.

Implementation

The intuition behind the general features is to get an overview of how prevalent certain elements are within each visualization. This may be an early indicator for certain visualization types. For example, heavy use of `circle`s may indicate a scatter or bubble chart. Our general features consist mainly of counts for the most commonly observed SVG element types. Specifically, `circle`, `rect`, `path`, `line`, and `text` types. For each SVG object (*ie* each extracted visualization), we count the instances of each element type. Special care is given to the `text` type such that labels and legends (eg. tick marks on the axis) are not included. This is done by filtering out any `text` that can be casted into a float. These counts are included as individual features within the "Counts" subgroup of features under the general features. Namely, the features within this subgroup are "Path Count", "Circle Count", "Rect Count", "Text Word Count", and "Line Count". The accuracy of the "Counts" subgroup as well as subsequent general features subgroups are shown in Table 1, and the significance of these subgroups are explained later in "Classification Accuracy". A series of tables in the appendix contains descriptions of each individual feature and the classification accuracy using that feature alone, for all 85 features. The table references are Table 11, 12, 13, 14, 15, 16, 17. All subsequent feature group sections follow the structure outlined above.

We also compute the number of horizontal and vertical "axes lines" in the visualization, which can help to distinguish between visualizations that often contain axes (*e.g.,* bar charts), and visualizations that do not (*e.g.,* geographic maps). We define axes lines as SVG elements with the following characteristics: 1) one or more `path` elements representing straight lines with at most two tick marks at the endpoints; 2) strictly horizontal or vertical in orientation; and 3) have length greater than half of the width of the visualization for horizontal `path`s, or greater than half the height of the visualization for vertical `path`s. These counts of vertical and horizontal axes lines are features within the "Axes Lines" subgroup, which consists of only two features: "Horizontal Axes Count" and "Vertical Axes Count". The pseudocode for counting and removing vertical axes lines is shown in Algorithm 1, and this same technique (with some minor adjustments) can be used for horizontal axes lines.

Algorithm 1 Counting and removing vertical axes lines

```
1:  FUNCTION remove_axes(paths)
2:  for path ∈ paths do
3:      d_attr ← path.get_attr("d")
4:      num_v ← COUNT(d_attr.contains("v"))
5:      num_h ← COUNT(d_attr.contains("h"))
6:      if num_v = 1 AND num_h <= 2 AND NOT d_attr.contains("z") then
7:          if COUNT(get_numbers_from_path(d_attr)) < 7
8:              AND get_axes_length(d_attr, True) > 0.5 * chart_height then
9:              vertical_axes.append(path)
10:             REMOVE(path)
11: return vertical_axes.length
12: ENDFUNCTION
```

The pseudocode shown in Algorithm 1 takes the set of all `path` elements in the visualization as input, and loops over these `path`s. For each `path`, the function `remove_axes()` extracts it's "d" attribute using the `get_attr()` function, which first searches for the attribute in the SVG element itself, and if the SVG element has no such attribute, the function searches through the element's "style" attribute, as styling attributes, such as fill color, may be specified there. `remove_axes()` next counts the number of vertical and horizontal segments (demarcated with "v" and "h", respectively) in the `path` by inspecting its "d" attribute (lines 4 and 5). Then two checks are performed on the `path` in line 6. Vertical axes lines are `path` elements with one long vertical segment and up to two horizontal tick marks at the two ends. The boxed region in Figure 0-3 shows an example of a vertical axis line with two horizontal tick marks at the the two ends (labeled "0" and "7"), so one check ensures that the `path` has a single vertical segment and fewer than three horizontal elements. The other check ensures that there is no "z" in the `path`'s "d" attribute, since its presence indicates that the `path` is closed as described in 'Path "d" Attribute' subsection of Section 0.5.3 under "Path Features". If a `path` passed these checks, and the length of its vertical segment is greater than half of the chart's height and there are fewer seven numbers in the d attribute, then the `path` element is considered a vertical axis line. Axes lines are usually long enough to span the majority of their corresponding dimension in the chart, which is

why the vertical axis line must be at least half the chart's height. The function `get_axes_length()` in line 8 calculates the height of the vertical axis line by determining the length of the vertical segment within the axis line, and is straightforward to implement. Axes lines are straight lines with at most two tic marks (one at each endpoint), which means the "d" attributes of their corresponding `path` elements cannot have too many numbers, because numbers in the "d" attribute indicate change of direction. Note that there may be additional tic marks on the axes lines as shown in Figure 0-3, but these tic marks are individual short `path` elements that are not part of the `path` element drawing the axes lines. We ignored these tic marks as they do not provide addition information beyond what axes lines give. The function `get_numbers_from_path()` gets all the numbers from the `path` element and the `path` is only considered when it has fewer than seven numbers. The "d" attribute of the vertical axis in Figure 0-3 is "M-6,0H0V450H-6", and has two horizontal tick marks, which corresponds to 5 numbers. This means axes lines should have at most 5 numbers, but we set the maximum threshold to 6 to maintain some flexibility with regards to coding style (e.g. if the tick marks are specified with the "l" character and requires two numbers for the destination coordinates as opposed to one as described in 'Path "d" Attribute'). This check filters out `path` elements with long vertical segments that are not necessarily axes lines, like line charts with a huge vertical jump. All these checks are performed in lines 6-8. Finally, the vertical axes lines are removed (line 10) and counted (line 11).

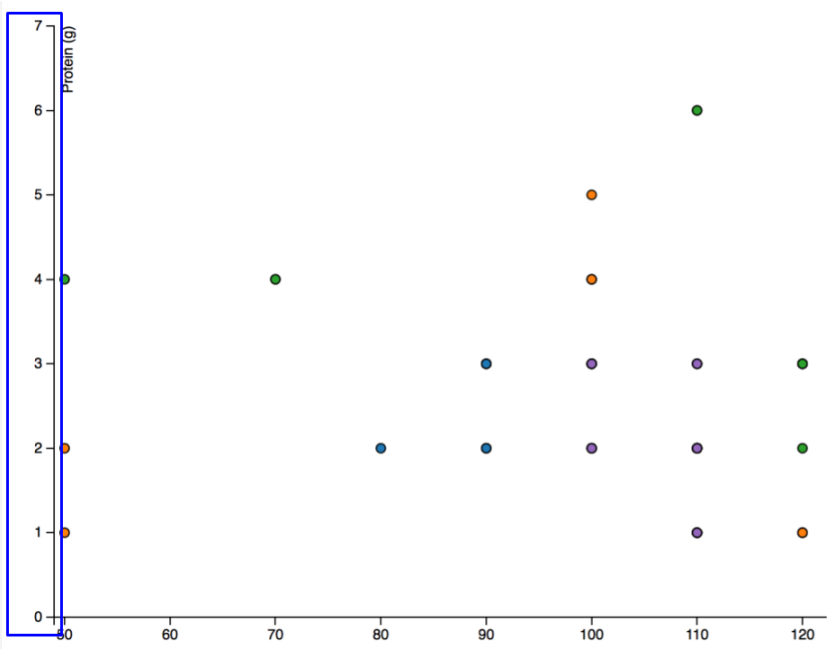

Figure 0-3: Vertical axis (boxed) in a scatter plot illustrating axes characteristics.

In addition to counting the number of non-numerical `text` elements, we also enumerate the number of unique x and y-positions a `text` element belongs in, which is useful for discerning charts with a heavy emphasis on text like word clouds and geographic maps. These `text` position counts are features within the

"Text" subgroup, which consists of the features "Text Unique X Count" and "Text Unique Y Count". The pseudocode for counting the number of unique x-positions across all `text` elements is shown in Algorithm 2. This method can be extended to count the number of unique x or y-positions, or other location-based features (e.g maximum y-position) for other SVG elements.

Algorithm 2 Counting unique x-positions of all `text` elements

```
1:  FUNCTION count_unique_text_x_positions(texts)
2:  x_set ← SET()
3:  for text ∈ texts do
4:      tx,ty ← get_absolute_location(text, 0, 0)
5:      if text.has_attr("x") then
6:          x_position ← tx+convert_str_to_float(text.get_attr("x"))
7:          x_set.add(x_position)
8:  return x_set.length
9:  ENDFUNCTION
10:
11: FUNCTION get_absolute_location(node, x, y)
12: if node is None then
13:     return (x,y)
14: if node.has_attr("transform") then
15:     transform ← node.get_attr("transform")
16:     coordinates ← get_coordinates_from_transform(transform)
17:     if coordinates then
18:         x = x + coordinates[0]
19:         y = y + coordinates[1]
20: return get_absolute_location(node.parent, x, y)
21: ENDFUNCTION
```

The first function `count_unique_text_x_positions()` in Algorithm 2 counts the number of unique x-positions observed across all SVG `text` elements. It takes the set of all `text` elements in the visualization as input, and for each `text` element, the code calls the function `get_absolute_location()`, which calculates the absolute position of the input element within the visualization. The absolute position of a SVG object (exact coordinates of its location within the visualization), is not simply the x,y-coordinates specified in its location attribute. This is because SVG objects are often grouped together, in invisible parent containers, and may even be nested in several containers. The containers are given x, y positions, and the individual SVG element is given an x,y position within the parent container. The sum of the element's coordinates and the positions of all its parent containers form the absolute location of the object. The element's x,y-position only makes sense in the context of its parent containers, and is useless on its own. The SVG element's relative position within it's parent container is specified as x and y-coordinate translations within the element's "transform" attribute. To extract absolute location, `get_absolute_location()` recurses up the SVG DOM tree starting at the node of interest (line 20), and at each node, extracts the x and y shifts in the transform attribute (lines 15-16) with the function `get_coordinates_from_transform()`. Transform attributes are strings with the form "translate(x,y)", where "x" and "y" are numerical strings representing translations in the x and y directions.

Subgroup	D3	Plotly
Counts	0.6420208500400962	0.6598218471636194
Axes Lines	0.45068163592622296	0.5482419127988748
Text	0.3204490777866881	0.473511486169714
Overall	0.6707297514033681	0.7889357712142522

Table 1: Classifier accuracy on the D3 and Plotly datasets using only features from subgroups of the general features. The "Overall" accuracy is that of all general features.

`get_coordinates_from_transform()` extracts the two numerical strings, converts them into floats, and returns a tuple of the form (x,y). This function can be implemented easily using regular expressions. Once the x and y translations of the current node has been extracted, `get_absolute_location()` updates the aggregate x and y shifts accordingly (lines 18 and 19). Recursion stops once the root of the DOM tree has been reached (line 14). Users also have the option of specifying x and y-coordinates of `text` elements via "x" and "y" attributes, so to account for this possibility, we have a function `count_unique_text_x_positions()` that adds the "x" attribute value to the calculated value from `get_absolute_location()` in line 6. Note that an SVG element's location must be specified in a single setting, so an element will not have a "transform" attribute if it has "x" and "y" attributes. The `text`'s element's absolute x-coordinate is added to the set of `text` x-coordinates (line 7), which stores the unique x-coordinate values over all `text` elements, so its final size is the number of unique x-coordinates across all `text` elements.

Classification Accuracy

Note that classification accuracy is higher for Plotly than D3 according to Table 1, and another reason (in addition to the fewer number of chart types for Plotly mentioned above) is that some D3 charts types are distinguished only by other feature groups. For instance, radii features of `circle` elements from the `circle` features group is used to distinguish scatter and bubble charts in D3, and is unnecessary in Plotly, since Plotly has scatter plots but not bubble charts.

The "Counts" subgroup performed much better than the other two groups for both datasets, and this is due to the comprehensiveness of the "Counts" subgroup, which includes counts of every SVG element type, while the other two subgroups target single SVG elements. The "Text" subgroup performed the worst, which is to be expected, since its features target text-heavy visualizations, such as word clouds, and is not useful towards many other chart types. Note the overall accuracy is not much higher than the best subgroup accuracy, and this could be due to redundancy in the features of different subgroups. For instance, high unique x and y-coordinate counts for `text` elements and no axes lines are both strong signals for a word cloud.

0.5.2 Styling features

Implementation

Differences in visualization types can be found in the way that the SVG elements are styled. For example, how they are colored (fill color and border colors), given border and line thicknesses, and in the case of text, how they are sized. Styling is often treated as orthogonal to the layout of elements in the DOM tree, so we analyze styling using a separate set of features.

When it comes to fill color and border (or stroke) color, we have found that recording the specific colors used in a visualization is unimportant, because choosing a color palette is largely an individual user preference. However, we found that counting the number of unique colors observed in the visualization can be useful, because many visualizations have color applied in a systematic way (*i.e.,* each bar in a bar visualization is often given a unique color). To create a feature using this information, we compute the total number of unique fill and stroke colors as two separate features. We also compute other statistics on colors, such as average number of occurrence of a `path` fill color (i.e. number of path elements that use a fill color averaged over all fill colors). All stroke color statistics are included as individual features in the "Stroke Color" subgroup, which consists of the features "Path Unique Stroke Color Count", "Average Path Stroke Color Count", "Path Stroke Color Count Variance", "Average Line Stroke Color Count", and "Line Stroke Color Count Variance". Likewise we have a subgroup for fill-related statistics, which includes the features "Filled Path Count", "Path Unique Fill Color Count", "Average Path Fill Color Count", "Path Fill Color Count Variance", "Average Circle Fill Color Count", "Circle Fill Color Count Variance", "Average Rect Fill Color Count", and "Rect Fill Color Count Variance". In addition, we consider the maximum and minimum stroke widths, as well as maximum and minimum font sizes. The "Stroke Width" subgroup contains this information about stroke widths and consists of the features "Maximum Path Stroke Width" and "Minimum Path Stroke Width". Given that visualization text can have large variations in font size, such as for word clouds, we also compute the total unique font sizes observed, as well as the variance in font size. The "Font Size" subgroup includes all font-related statistics of `text` elements, and has the features "Maximum Text Font Size", "Minimum Text Font Size", "Text Font Size Variance", and "Text Unique Font Size Count". The accuracy of all these styling features subgroups and the accuracy of the entire styling features group are shown in Table 2.

However, styling can also be inherited from parent objects within the DOM, as well as by using Cascading Style Sheets (or CSS). This potentially results in SVG elements appearing to have no style attributes. To address this, we first look for style attributes within the element itself. If no styling attributes are found, we iteratively search the element's ancestors in the DOM tree for styling attributes. If the element is still without styling attributes, it is very likely that they are provided in one or more CSS files in the webpage, with the SVG element's class names linking it to style attributes specified in the CSS files. A possible extension for future work is to cross reference each element's CSS classes with the classes specified in the webpage's CSS files. Pseudocode computing the number of `path`s with each fill color averaged over all fill colors is shown

in Algorithm 3, and this code also illustrates the style attribute search technique. Other style features can be extracted using an extension of this idea.

Algorithm 3 Computing number of occurrences of each `path` "fill" color averaged over all `path` "fill" colors

```
1:  FUNCTION get_avg_fill_color_occurence(paths)
2:  fill_count ← {}
3:  for path ∈ paths do
4:      fill_color ← find_style_attribute(path, "fill")
5:      if fill_color then
6:          if fill_count.get(fill_color) then
7:              fill_count[fill_color] ← fill_count[fill_color] + 1
8:          else
9:              fill_color[fill_color] ← 1
10: return AVERAGE(fill_count.values())
11: ENDFUNCTION
12:
13: FUNCTION find_style_attribute(element, attribute)
14: output_element ← element
15: while output_element do
16:     output ← get_attribute(output_element, attribute)
17:     if output then
18:         return output
19:     output_element ← output_element.parent
20: return None
21: ENDFUNCTION
```

The function `get_avg_fill_color_occurence()` first enumerates the number of `path` elements with a particular "fill" color for all "fill" colors present in `path` elements of the visualization. The function then averages the counts over all colors, which becomes the value for the "average fill color occurrence" feature. It takes all `path` elements in the visualization as input and extracts the fill color of each `path` using the function `find_style_attribute()` in line 4, passing in the `path` element and the attribute of interest ("fill"). The remaining lines in `get_avg_fill_color_occurence()` keep track of the count of all "fill" colors in a pythonic dictionary (similar to a hashmap) in lines 6-9, and finally returns the average count over all fill colors in line 10. Since the "fill" attribute of a `path` element is a styling attribute, it's value could be set in one of the `path`'s ancestors as described earlier. The function used to extract the "fill" color attribute (`find_style_attribute()`) follows the iterative style-attribute search algorithm mentioned above. We cannot use the function `get_attr()` to extract style attributes, because this function does not search the element's ancestors for missing attributes, since it is geared towards non-styling attributes that cannot be inherited, such as the radius of a `circle` element. `find_style_attribute()` takes the SVG element (`element`) and the attribute of interest (`attribute`) as input and starting at the input element, traverses up the SVG DOM tree, searching for the input attribute and returning the first hit.

Subgroup	D3	Plotly
Stroke Width	0.4123496391339214	0.5694327238631036
Stroke Color	0.4380112269446672	0.7197374589779653
Fill	0.46351242983159585	0.8216596343178622
Font Size	0.3199679230152366	0.37093295827473044
Overall	0.6105854049719326	0.8975152367557431

Table 2: Classifier accuracy on the D3 and Plotly datasets using only features from the subgroups from styling features group. The "Overall" accuracy is that of the entire style features group.

Classification Accuracy

The "Stroke Width", "Stroke Color", and "Fill" subgroups all performed better for Plotly than for D3. This is due to the fact that the creators of Plotly chose to represent nearly all graphical markers (e.g. circles in a scatter plot) with `path` elements. As a result of this emphasis on `path`s, certain characteristics of `path` features are strong indicators for Plotly chart type, and this is not the case for D3. For instance, if a Plotly visualization has a high count of `path` elements with fill, it is most likely a scatter chart, as it is unlikely for other Plotly chart types to have as high of a filled `path` count. On the other hand, both hexabins and sunbursts (present only in D3) usually have high filled `path` counts. These three subgroups yield similar levels of accuracy for D3 as none of them not provide sufficient information to distinguish many D3 chart types well. The "Font Size" subgroup performed the worst for both datasets, as expected, as it is targeted towards text-heavy visualizations, and is not a good indicator for other chart types. For Plotly, the overall accuracy is not much higher than the best performing subgroup ("Fill"), which could again, be due to redundancy.

0.5.3 Per-Element features

Lastly, we examine the attributes of each SVG element type and use this information to classify visualization designs. For example, rectangles that share the same y position in a visualization might indicate a bar chart, and circles that share the same radius in a visualization might indicate a scatterplot. Figure 0-4 illustrates both of these examples. In Chart A (bar chart), the y-positions (of the bottom left corners) of all the rectangles representing bars are the same, as shown in the boxed region. Other chart types with rectangles do not have this property, as shown in Chart B (heat map), where the rectangles are stacked vertically (see boxed region). In Chart C, all circles representing points in the scatter plot have the same radii (see boxed region). This is not the case for circles representing axes in radial charts as illustrated by the boxed region in Chart D. Note that all features are normalized to standardize the feature vectors. X-positions are normalized by dividing by visualization width, y-positions by visualization height, all line and path lengths by visualization diagonal, and all shape widths by either width or height (whichever is larger). We extract a base set of features for every type, and then shape-specific features (e.g. the radius size for `circle` elements).

Features extracted for all SVG elements : For all elements of the same type within a given visualization (*e.g.,* for all `circle`s, `rect`'s, etc.), we calculate a standard set of statistics. Half of these features are

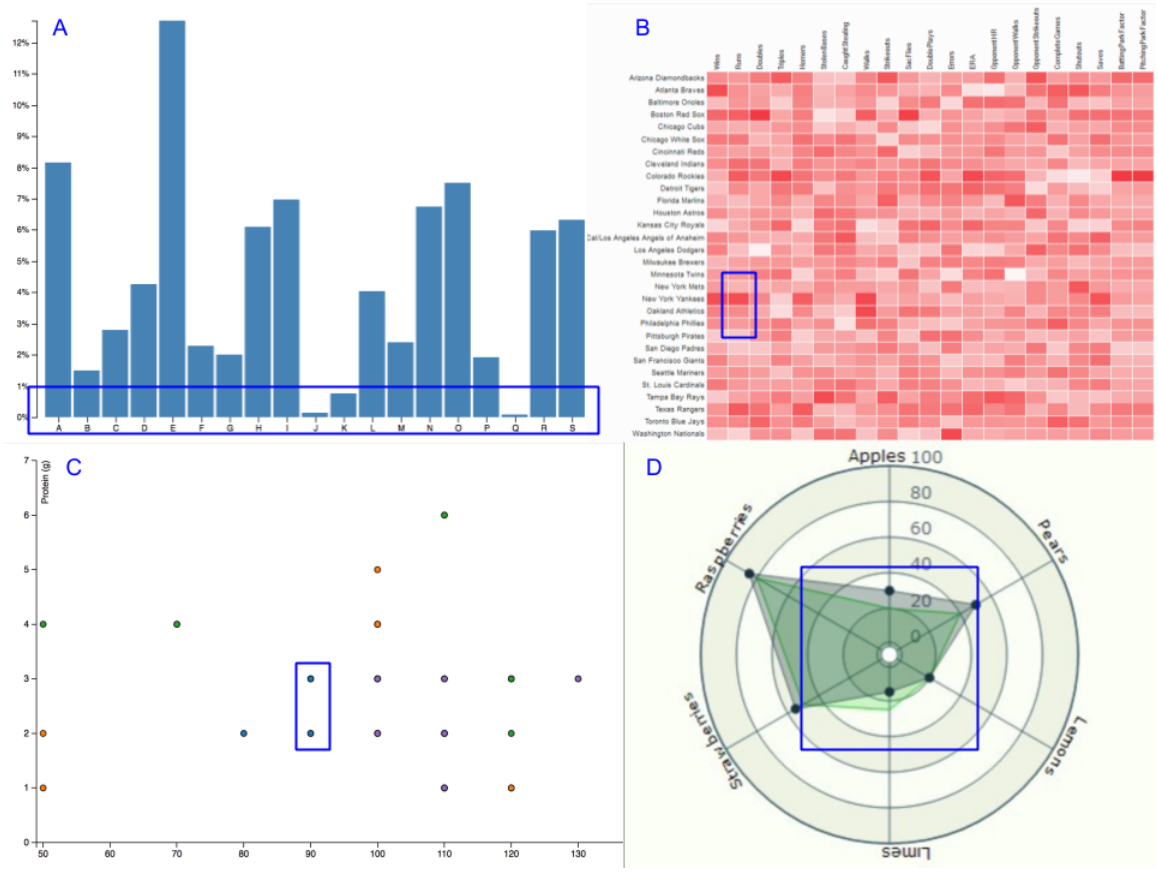

Figure 0-4: Four chart types illustrating characteristics of rectangle y-positions and circle radii for certain chart types.

calculating the maximum and minimum x-position, and then repeating for y-position. The remaining features calculate: the number of unique x-positions, the number of unique y-positions, and the average number of elements that share positions. For example, for all `rect`'s within the visualization, how many share the same y position on average? Note that we consider the y position of a `rect` to be its y position plus its height, since the y-position of an SVG element is defined as distance from the top of its container. Also, locations of `path` and `line` elements are that of their starting points. Tracking x-position and y-position allows us to identify positioning patterns. For example, vertical bar charts will contain rectangles with the same y-position, and evenly spaced x-positions. In contrast, scatter charts will have high numbers of both unique x and unique y-positions, helping the classifier to discern bar charts from scatter charts. All the x-position statistics for each SVG element type are included as individual features within the "X" subgroup of features in the Per-Element features group for that shape (e.g. all per-element features for `circle`s are in the "Circle Features" group, and a subset of these features pertaining to the x-position are in the "X" subgroup for `circle` elements). The features in the "X" subgroup are "Maximum X", "Minimum X", "Unique X Count", "Average X Count", and "X Count Variance", and they are calculated for `circle`, `rect`, `path`, and `line` elements. Likewise, there is a "Y" subgroup for y-coordinate statistical features, which include "Maximum Y", "Minimum Y", "Unique Y Count", "Average Y Count", and "Y Variance". See Algorithm 2 under Section 0.5.1 for an example of extracting position features.

Circle Features

Implementation For `circle`s, we calculate the maximum radius out of all `circle` elements in the visualization and variance in the radii. We considered including both maximum and minimum radius, but found that the minimum radius feature provided no additional benefit beyond the maximum radius and radii variance features. Thus, we omitted the minimum radius. We found that maximum radius was better than minimum radius because visualizations with `circle`'s of varying sizes, such as bubble charts, are likely to have `circle`s with rather large radii. The smallest radii in bubble charts are unlikely to be smaller than the radii of `circles` in scatter plots, so the minimum radius is worse at distinguishing between these two chart types. We also consider the maximum number of circles with identical radii. Radii variance helps to discern visualizations with equal-size circles (*e.g.,* scatter charts), from those with varying circle sizes (*e.g.,* bubble charts and radial charts). All these radii features are included in the "Radii" subgroup of `circle` features. Specifically, the features within the "Radii" subgroup are "Maximum Circle Radius", "Circle Radius Variance", and "Maximum Circle Radius Count". The accuracies of all subgroups in the circle features group ("X", "Y", "Radii"), as well as the accuracy of the entire group can be found in Table 3. Pseudocode for extracting the largest radius out of all `circle` elements is shown in Algorithm 4, and this method can be extended to extract all radius-based features.

Algorithm 4 Calculating the maximum radius over all `circle` elements in the visualization

```
1: FUNCTION get_max_radius(circles)
2: radii_list ← []
3: for circle ∈ circles do
4:     radii_list.append(convert_str_to_float(circle.get_attr("r")))
5: return MAX(radii_list)
6: ENDFUNCTION
```

Subgroup	D3	Plotly
X	0.40096230954290296	0.19503047351148617
Y	0.3863672814755413	0.19465541490857946
Radii	0.40930232558139534	0.1944678856071261
Overall	0.4396150761828388	0.19484294421003281

Table 3: Classifier accuracy on the skewed D3 and Plotly datasets using only features from the subgroups from "Circle" features group. The "Overall" accuracy is that of the entire "Circle" features group.

This function takes in a list of all `circle` elements in the visualization as input, and iterates through the list. For each `circle` element, the code extracts the circle's radius (specified in its "r" attribute), and converts the value into a float, since everything extracted from SVG is of type string, and adds it to a list of all radii seen so far (lines 3-5). The function then returns the maximum value of the list.

Classification Accuracy

As shown in 3, the "Circle" features group performed significantly better on D3 than on Plotly because Plotly rarely uses `circle` elements. This is supported by the fact that every subgroup performs equally as poor on Plotly, including the group as a whole, since the `circle` features from every chart type are likely quite similar. To verify this, we queried the Datastore (Section 0.7) to count the number of Plotly visualizations using `circle` elements and found there were only 25 out of the 6544 total visualizations. The "Circle" group performs decently for D3, as this group is targeted towards a select number of chart types: scatter, bubble, and radial, and each group has a similar accuracy since they are equally good at discerning between these chart types. For example, radial charts haves axes made up of concentric circles, which will be signaled by high radius variance from the "Radius" subgroup and low unique x count from the "X" subgroup. The overall accuracy is not much higher than that of individual subgroups, due to redundancy, since each subgroup targets the same set of chart types.

Rect Features

Implementation For `rect` elements, we do not calculate any unique features other than the x and y position features described above. Height and width information do not contribute additional information when given x and y position information. This is because `rect` graphical elements are usually adjacent to each other (e.g. treemap) or equally spaced apart (bar graphs), so dimensional information can be inferred by taking the difference in corresponding in location information. The pseudocode for calculating x and y features should

Subgroup	D3	Plotly
X	0.4920609462710505	0.612751992498828
Y	0.4848436246992783	0.5879043600562588
Overall	0.5008821170809944	0.6347866854195968

Table 4: Classifier accuracy on the D3 and Plotly datasets using only features from the subgroups from `rects` features group. The "Overall" accuracy is that of the entire `rects` features group.

be similar to the pseudocode for calculating `text` location features shown in Algorithm 2. The accuracies of the "X" and "Y" subgroups for `rect` features are shown in Table 4.

Classification Accuracy

The "X" and "Y" subgroups had similar accuracy in both datasets, which is expected because x and y positioning are generally used equally in most visualizations. For example, if a chart has low variance in x-position, it could be bar chart with vertical bars or a heatmap, and having a low variance in y-position indicates that the chart could be a chart with horizontal bars or a heatmap. The higher accuracy for "X" could be due to the fact that there are more bar charts with vertical bars than horizontal bars in the dataset. In addition, overall accuracy of the entire group is only slightly higher than the individual subgroups. This could be due to redundancy. However, having both x and y position information enables the differentiation of bar charts from heatmaps (for the case with D3), since heatmaps will have high variance in both x and y positions, while bar charts have high variance in either x or y. However, there are only 32 heatmaps total, out of the 1247 charts as shown Table X, and since there are considerable more bar chart (154), the classifier is likely to erroneously classify every heatmap as a bar chart when using a single subgroup, which results in a $32/1247 \approx 0.0257$ decrease in accuracy, which is comparable to the difference in accuracies between that of the group and individual subgroups.

Path Features

This section contains additional sections detailing the "d" attribute of `path` elements that specifies the `path`'s trajectory. "d" attributes were analyzed extensively in `path` features, and are rather complicated with many different parts. The 'Path "d" Attribute' section describes the "d" attribute in detail, and 'Path "d" Attribute Examples' provides examples illustrating various "d" attribute features.

Path "d" Attribute The "d" attribute of a `path` element is a string with instructions for drawing the trajectory of the `path` [13]. The string consists of a combination of the following instructions: moveto, lineto, curveto, arcto, and closepath. All these rules are detailed in the appendix, and we present a concise version here discussing only the instructions analyzed by the classifier. "Lineto" draws a straight line between the current position and the destination point, which becomes the new current position. "Lineto" is specified as "L x,y", where "x" and "y" are destination coordinates, and "H x" and "V y" are used for horizontal and vertical

Command	Definition	Syntax
Lineto	Draws a straight line between the current position and the destination point (given by "x" and "y" in Syntax)	"L x,y" "H x" for horizontal lines "V y" for vertical lines
Curveto	Draws a bezier curve	"Q cx,cy,x,y"
Arcto	Draws an elliptical arc	"A rx,ry,xAxisRotate,LargeArcFlag,SweepFlag,x,y"
Closepath	Draws a straight line from the current position to the start of the path, creating a closed shape	"Z"

Table 5: Table listing the definitions and syntax of `path` "d" attribute instructions analyzed by the classifier. See the 'Path "d" Attribute' section for more detailed explanations.

lines, respectively, where "x" specifies the x-coordinate of the endpoint of the horizontal line, and "y" specifies the y-coordinate of the endpoint of the vertical line. For all d-attribute instructions, "x" and "y" correspond to the destination coordinates. "Curveto" draws bezier curves, and bezier curves are parametric curves traced by polynomial functions [12]. Our features focus on quadratic bezier curves only, and they are specified as "Q cx,cy,x,y". See the appendix for descriptions of "cx" and "cy" as we did not consider these attributes. "Arcto" declares an elliptical arc and is specified as "A rx,ry, xAxisRotate, LargeArcFlag, SweepFlag x,y", where "rx" and "ry" specify the radius in the x and y directions, respectively. See the appendix for descriptions of the remaining attributes. "Closepath" draws a straight line from the current position to the first point in the path, creating a closed shape. It is specified with a "z" in the "d" attribute. There are lowercase versions of all these instructions (e.g. "a" for "Arcto"), where all x and y-coordinate attributes are shifts relative to the previous position. Table 5 summarizes these "d" attribute commands for quick reference, and these commands are illustrated in two example `path` elements described in 'Path "d" Attribute Examples'.

Path "d" Attribute Examples The boxed chord segment of a chord chart shown in Figure 0-5 has the following "d" attribute:
"M272.9461639226633,-305.9418108070596A410,410,0,0,1,391.69303884469844,-121.14686673870506
Q0,0,-380.088988411556,-153.72820459590378A410,410,0,0,1,-263.8763386673507,-313.79814832390844
Q0,0,272.9461639226633,-305.9418108070596Z".
The first "Arcto" instruction is ("A410,410,0,0,1,391.69303884469844,-121.14686673870506"), where "rx" and "ry" values of 410 indicate a circular arc, and (391.69303884469844,-121.14686673870506) are the coordinates of the arc's endpoint. This instruction draws the circular arc connecting bottom left and top left vertices of the path element along the circumference of the chord chart. The next command is a "Curveto" ("0,0,-380.088988411556,-153.72820459590378"), which draws a quadratic bezier curve that has an endpoint at (-380.088988411556,-153.72820459590378). This instruction draws the top curve connecting the top left and top right vertices across the chord chart. The subsequent instructions are another "Arcto" drawing the right arc connecting the top right and bottom right vertices, another "Curveto" drawing a curve connecting the bottom right and bottom left vertices, and an unnecessary "ClosePath" that connects the end point of the `path` element at (272.9461639226633,-305.9418108070596) to its starting point at the same location.

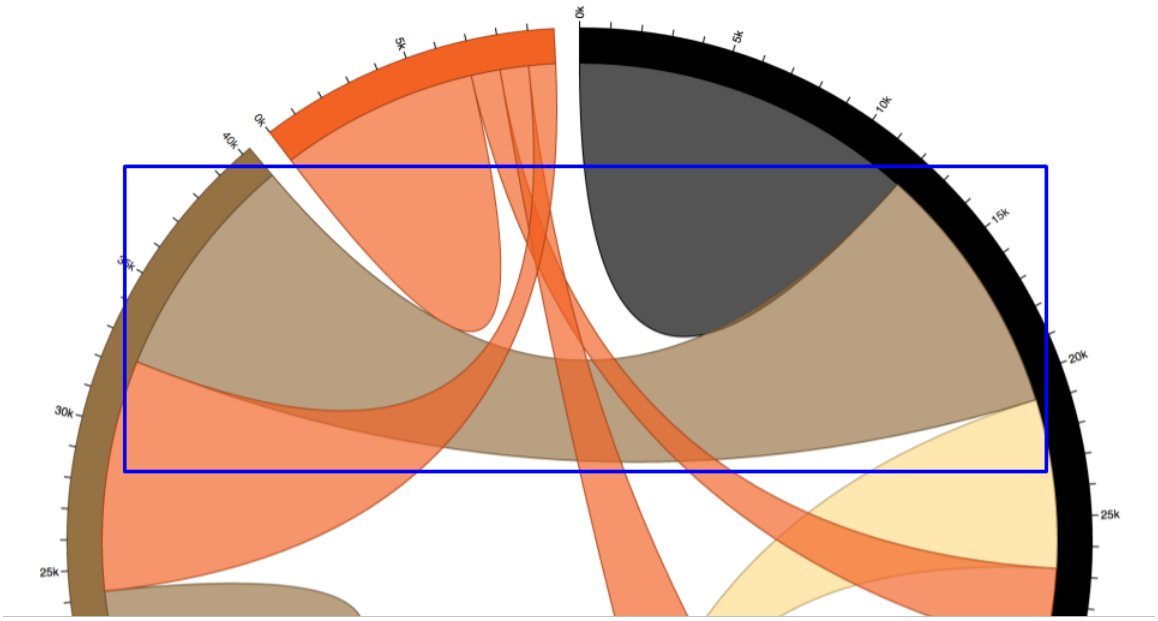

Figure 0-5: Chord segment (boxed) in a chord chart illustrating "d" attribute commands.

The boxed hexagon in a hexabin shown in Figure 0-6 has the following "d" attribute:
"m0,-20l17.32050807568877,9.999999999999998l3.552713678800501e-15,20l-17.320508075688764,
10.000000000000004l-17.32050807568878,-9.999999999999991l-1.0658141036401503e-14,
-19.999999999999993l17.32050807568876,-10.000000000000014z".
It starts at (0,-20), as specified by the "Moveto" command "m0,-20". The next instruction is a "Lineto" ("l17.32050807568877,9.999999999999998") that draws one of the six sides of the hexagon with a line starting at (0,-20) and extending 17.32050807568877 in the x direction and 9.999999999999998 in the y direction since the "l" is lowercase. The next four commands are all "Lineto", drawing the next four sides of the hexagon. The final side is drawn with a "ClosePath" command connecting the start and end points of the hexagon, and is specified with a "z" at the end of the "d" attribute.

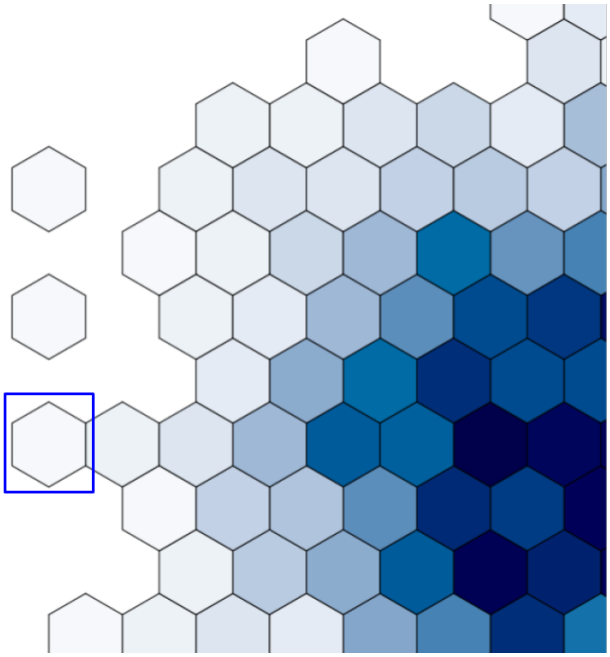

Figure 0-6: Hexagon (boxed) in a hexabin illustrating "d" attribute commands.

Implementation For `path` elements, we mainly consider the number of characters used to specify each path. We do this by analyzing the `path`'s "d" attribute, which is a string containing an ordered list of commands for how to draw the path (*e.g.,* move to point A, draw a line to point B, etc.). The "d" attribute is explained in detail in the 'Path "d" Attribute' section. Specifically, we calculate the maximum, minimum, mean and variance in the length of the "d" attribute, across all `path`s. This feature is extremely useful for paths, because the longer a "d" attribute is, the more detailed it is. Highly detailed paths are often seen when drawing complex shapes, such as countries or states in geographic maps. These statistics on "d" lengths are features within the '"d" Length subgroup, and the individual features are "Maximum Path d-attribute Length", "Minimum Path d-attribute Length", "Average Path d-attribute Length", "and "Path d-attribute Length Variance". The

pseudocode to find the maximum length (in number of characters) of the "d" attribute over all `path`'s is shown in Algorithm 5.

Algorithm 5 Finding the longest "d" attribute over all `path`s

```
1: FUNCTION get_max_d_length(paths)
2: d_lengths ← []
3: for path ∈ paths do
4:     d_lengths.append(path.get_attr("d").length))
5: return MAX(d_lengths)
6: ENDFUNCTION
```

This function takes the list of all `path` elements in the chart as input, and gets the length each `path`'s "d" attribute (line 4). The function stores all "d" attribute lengths and then returns the maximum value in the list (line 5).

However, path elements are complex, and can be used in place of other SVG elements. As such, we perform additional analyses to account for these cases. To find polygon-heavy visualizations (*e.g.,* voronoi and hexabin visualizations), we compute the number of closed elements (*i.e.,* paths that start and end in the same place) found across all paths. Then we calculate statistics over all paths that contain at least one closed element. Specifically, the maximum, minimum, mean, and variance in the length of the "d" attribute of these paths. All closed `path`s have a "z" at the end of their "d" attributes as mentioned in the 'Path "d" Attribute' section, so the "Z" subgroup contains features describing closed path elements; namely, "Maximum Closed Path d-attribute Length", "Minimum Closed Path d-attribute Length", "Average Closed Path d-attribute Length", and "Closed Path d-attribute Length Variance". We also compute the number of arc and curve calls made within a `path`, which are used to draw them with `path` elements. Counting arc calls helps to distinguish visualizations that contain circles (*e.g.,* scatter, radial, and bubble charts, and chord diagrams). Arc elements also have radii attributes, which provide useful information; for instance, the variance in arc radii helps discern sunbursts, which consist of a myriad of stacked arc segments with varying radii. A `path` arc has both horizontal and vertical radii, since these arcs are not necessarily circular, and `path` arc features are calculated separately for horizontal and vertical radii. Arcs are demarcated with an "A" in the "d" attribute as stated in 'Path "d" Attribute', so the subgroup "A arcs" includes features describing arcs, namely "Maximum Circular Arc Counts per Path Element", "Path Rx Variance", and "Path Ry Variance". Pseudocode for computing the variance in the horizontal radii ("rx" attribute of the arc, as described in 'Path "d" Attribute') over all `path` elements in the visualization is shown in Algorithm 6. This technique can be extended to extract all arc-related features. `path` elements can also be used to draw bezier curves, which are demarcated with a "Q" in the "d" attributed as mentioned in 'Path "d" Attribute'. These "Q" curves are present in visualizations with a lot of bezier curves, such as chord charts, so we compute statistics on the occurrence of bezier curves, such as the total number of bezier curves in a visualization, and the average and minimum number of bezier curves per `path`. These

Subgroup	D3	Plotly
X	0.5310344827586206	0.8407876230661041
Y	0.5021651964715317	0.8496952648851384
Z	0.4773055332798717	0.946460384435068
"d" Length	0.561186848436247	0.9052977027660571
"A" Arcs	0.35878107457898956	0.4671354899203001
"Q" Curves	0.4259823576583801	0.2582278481012658
Overall	0.643785084202085	0.9638068448195031

Table 6: Classifier accuracy on the D3 and Plotly datasets using only features from the subgroups from "Paths" features group. The "Overall" accuracy is that of the entire "Paths" features group.

statistics are included in the subgroup "Q curves", which consists of the features 'Minimum Number of "Q" Curves Per Path', 'Average Number of "Q" Curves Per Path, and 'Path "Q" Curve Count'.

Algorithm 6 Computing the variance over the horizontal radii of all `path` arcs

```
1:  FUNCTION compute_rx_var(paths)
2:  rx_radii ← []
3:  for do path ∈ paths
4:      d ← path.get_attr("d")
5:      for i = 1, i <= d.length,i+ = 1 do
6:          if d[i].lower() ='a' then
7:              rx ← get_numbers_from_path(d[i+1:] : d.length))[0]
8:              rx_radii.append(rx)
9:  return VARIANCE(rx_radii)
10: ENDFUNCTION
```

This function computes the variance in the horizontal radii of all `path` arcs in the visualization. It takes a list of all visualization `path` elements as input, and for each `element`, the code locates all the arcs, which are demarcated with the "A" or "a" character in the `path`'s "d" attribute (line 6). In a `path` arc, the first two numbers following the "a" character are the values for the "rx" and "ry" (vertical radius) attributes of the arc, as explained in the 'Path "d" Attribute' section. Line 7 of the code extracts the first number following each "a" symbol of the d-attribute to get the "rx" values of all the arcs in the `path`, using the function `get_numbers_from_path()` that was described earlier. Each "rx" value extracted is added to a list, which keeps track of all "rx" values, and the variance of all "rx" values stored in the list is returned (line 12), which gives the variance of "rx" values of all `path` elements in the visualization.

Classification Accuracy

As expected, the "Path" group performed very well on Plotly, due to its heavy use of `path` elements (see Table 6). The low accuracy of the "A Arcs" subgroup on Plotly, which is comparable to the 19 percent accuracy achieved by `circle` elements (see Table 3), is due to the fact that Plotly rarely uses bezier curves in `path` elements, if at all. The high accuracies of the "X", "Y", "Z", and "d Length" subgroups indicate that there are characteristics unique to many visualization types among features in each of these groups. For instance,

Subgroup	D3	Plotly
X	0.5185244587008822	0.19465541490857946
Y	0.5164394546912591	0.19484294421003281
Overall	0.5323175621491579	0.19531176746366619

Table 7: Classifier accuracy on the skewed D3 and Plotly datasets using only features from the subgroups from `line` features group. The "Overall" accuracy is that of the entire `line` features group.

scatter charts have low variance in `path` "d" attribute length, and their lengths may fall within a certain range that was calculated by the classifier from training. The overall accuracy of this "Path" features group is only slightly lower than that of the entire classifier (96.4 vs 97.1), which means this group alone is already a great classifier for Plotly. This is again, expected due to Plotly's heavy use of `path` elements.

"A" arcs performed considerably lower than these four groups for D3, which could be due to the fact that it targets a few visualization types: radial, scatter, pie, and donut, and has fewer features than the high performing groups (3 vs 4). "X", "Y", "Z", and "d Length" all had relatively similar performances, with "d Length" performing the best, possibly due to the fact that it targets geographic maps, which, as explained earlier, have a high representation in the D3 dataset as shown in Table 7. "Q Curves" perform better than "A arcs" in the D3 dataset, which is unexpected as bezier curves are generally less frequently used. It could be the case otherwise in D3, where bezier curves are used heavily in chord charts.

Line Features

Implementation For `line`s, we do not calculate any additional features other than those for x, y positions. This is due to the fact that lines do not have any unique attributes. The accuracies of the "X" and "Y" subgroups for `line` features are shown in Table 7.

Classification Accuracy

The accuracies of each subgroup and the entire group are very close to that of the `circle` features group on Plotly, which means Plotly hardly uses `line` elements. As with `circle`s, we queried the Datastore (Section 0.7) to count the number of Plotly visualizations using `line` elements and found there were only 25 as well.

`line` elements are used by visualizations with a lot of short straight line segments, such as graphs and box plots, and are hence, targeted towards these visualization types. The "X" and "Y" subgroups performed equally well in D3, which could be because `line` elements behave similarly in the targeted visualization types with respect to these subgroups. `line`s are used for edges in graphs, and edges are usually scattered through the graph, so there will be high variance in both x and y positions. Likewise, `line`s are used for the "whiskers" and the line segment demarcating the median in a box plot, which means there will be similar variance in

both x and y positions, but lower than the variance present in graphs. The accuracy of the entire group is very close to that of the two subgroups due to redundancy in visualization type signal given by each subgroup as explained earlier.

Python Setup

All feature extraction code was written in Python. Beautiful Soup's XML parser was used to analyze all SVG files extracted during data collection. All features extracted from the SVG data were passed into the Python Scikit-Learn [24] random forest classifier for evaluation. A random forest classifier consists of multiple decision tree classifiers, each fitting on a certain sub-sample of the dataset, and uses averaging across decision trees, which improves accuracy and mitigates over-fitting. Tuning on various numbers of decision trees in the forest yielded 14 decision trees as the optimal number. The details of the classification method and the analysis results will be presented in the section "Accuracy Results for Labeling Visualizations".

0.6 Accuracy Results for Labeling Visualizations

The most important component of Beagle is the Annotator. The Annotator automatically classifies each visualization with its corresponding type, which allows designers to filter the final collection for the specific visualization types they want to study. As such, it is necessary to analyze how accurately the Annotator can classify visualizations, to ensure that visualization designers can rely on the classification labels provided by Beagle. In this section, we evaluate the Annotator's SVG-based classifier using our two visualization collections, bl.ocks.org and Plotly, and present classification accuracy results for each collection.

0.6.1 Dataset Details

We use the bl.ocks.org and Plotly visualization collections for our validation experiments. Table 8 provides a summary of these details for each visualization collection, and Figure 0-7 shows an example of every chart type present in the entire dataset.

Note that for our experiments, we only use half of the visualizations extracted by the Web Crawler. This is because many visualizations contain additional complex features that are outside the scope of our classification evaluation, such as animated visualizations, visualizations requiring user input (*e.g.,* visualizations with text boxes and radio buttons), coordinated views (*i.e.,* multiple visualizations within a single SVG object), and diagrams and infographics (*e.g.,* diagrams of the solar system). We explain how we filter for irrelevant visualizations, and label the remaining visualizations with their corresponding visualization types in the following section ("Recording Ground Truth").

Collection	Size	Total Types	Visualization Type Labels
bl.ocks.org	1247	22	area (32), bar (154), box (11), bubble (70), chord (34), donut (31), heatmap (32), geographic map (379), graph (60), hexabin (21), line (157), radial (13), pie (7), sankey (11), scatter (118), treemap (10),voronoi (25),waffle (12),word cloud (6), sunburst (28), stream graph (13), parallel coordinates (23)
Plotly	6544	11	area (10), bar (1364), box (259), contour (118), donut (193), filled-line (126), geographic map (184), line (1198), pie (26), radial (17), scatter (3049)
Both Collections	7791	24	area(42), bar (1518), box (11), bubble (70), chord (34), contour (118), donut (224), filled-line (126), geographic map (583), graph (60), heatmap (32), hexabin (21), line (1355), parallel coordinates (23), pie (33), radial (30), sankey (11), scatter (3167), stream graph (13), sunburst (28), treemap (10), voronoi (25), waffle (12), word cloud (6)

Table 8: General information about each visualization collection in our evaluation.

Figure 0-7: Collage showing every chart type present in the dataset. The chart types are (from left to right, top to bottom) area, bar, box, contour, filled-line, geographic map, graph, heatmap, hexabin, bubble, line, parallel coordinates, sankey, treemap, scatter, stream graph, voronoi, waffle, word cloud, radial, sunburst, pie, donut, and chord.

bl.ocks.org Dataset

The bl.ocks.org collection was created by crawling the bl.ocks.org website, which contains thousands of visualizations created using the D3 JavaScript library [10]. This collection contains 1247 D3 visualizations, and 22 different visualization types, listed in Table 8. We attribute the high number of visualization types to the fact that D3 is more flexible than many common visualization tools, such as Microsoft Excel, R's ggplot2 [35], and Python's Matplotlib [21]. The bl.ocks.org website also provides designers with a low-risk environment for experimenting with visualizations, resulting in wider variety. Note that D3 supports all five of the SVG elements analyzed in the Beagle Annotator, allowing for a diverse set of statistics to be calculated over D3 visualizations.

The bl.ocks.org collection contains many visualization types that are complex, and easily confused with several other visualization types. For example, chord visualizations are circular (like pie, donut, sunburst, and radial charts), with lines or curved areas (like sankey visualizations, and line and area charts). Voronoi visualizations contain many polygons (like maps) or many lines (like line charts or parallel coordinates), and often contain circles (like scatter and bubble charts). Similarly, Graphs contain many lines (like line charts, parallel coordinates and voronoi visualizations) and many circles (like scatter and bubble charts). However, very few of these complex visualization types are studied in previous work [31, 26, 36, 32]. Furthermore, the visualization types that are evaluated in previous work are a subset of [26, 36, 32] (or nearly a subset of [31]) the types in the bl.ocks.org collection. Thus, the bl.ocks.org collection is expected to be more challenging to classify than collections studied in previous work.

Plotly Dataset

The Plotly collection was created by crawling the plot.ly website. Although we scraped around 15k visualizations from Plotly, we ignored any Plotly visualizations that are rendered as raster images (3D visualizations, 2D histograms, and heatmaps), because they lack any useful SVG data. After removing these samples, we are left with a collection containing 6544 visualizations, across 11 different visualization types. Given that Plotly is more like a traditional visualization tool, there is less variety in the design of Plotly visualizations, resulting in fewer visualization types. The available visualization types in the Plotly collection are similar to the visualization types studied in previous work [26, 36, 32, 31].

In contrast to D3, all Plotly visualizations only consist of `path`, rect and `text` elements, even when rendering circles and rectangles (*e.g.,* for scatter and bar charts). Thus, only a subset of the Annotator's statistics are relevant, potentially posing a challenge for Beagle's classification techniques.

0.6.2 Recording Ground Truth

We treat the automatic classification process as a supervised learning problem. We use supervised learning as a starting point, because we want to verify that our techniques would actually work in the supervised case. If not, semi-supervised methods, which are more scalable and a better approach long-term, is of limited use. We plan

to switch to semi-supervised or active learning in the future, when for instance, we want to label large scale scrapes with Beagle.Since we are using supervised learning, in order to train the SVG-based classifier in our Annotator, we need to apply classification labels to each visualization in our collections (*i.e.,* the visualization type). We considered the visualization types that appeared in both of our data collections, and created a single set of labels to cover them. The final set of labels is provided in Table 8, along with the number of samples observed for each visualization type and each data collection.

General Approach

As expected, we found that bl.ocks.org has far more variety in its visualization types compared to Plotly. As a result, we decided to first identify as many visualization types as we could using the bl.ocks.org collection, then fill in gaps as necessary for Plotly. After an initial pass over the bl.ocks.org visualizations, we found roughly 22 visualization types (third row of Table 8). We found that Plotly has two visualization types that are not well-represented by the initial bl.ocks.org label set: "filled-line" visualizations and "contour" visualizations. Thus, we added these two visualization types to our set, resulting in a final list of 24 visualization types (bottom row of Table 8).

Consolidating Visualization Types and Styles

Given strong similarities between certain visualization types (*i.e.,* choropleth maps and map projections) and variations (*i.e.,* grouped and stacked bar visualizations), we opted to merge certain visualization types and variations to form a less redundant set of labels for classification. Here, we identify the visualization groups and style groups that we merged, and the reasoning for merging each group.

Geographic Maps: We found four styles of geographic maps in our search: choropleth, contour, map projections, and general maps. We found that choropleth maps and map projections were very similar to general maps. As such, we consolidated these three styles into a single label of "geographic map". However, contour maps were strikingly different from the other maps, and seemed to be a mixture of area charts and maps. As such, we kept contour maps as a separate visualization type.

Graphs and Trees: We found three styles of graph-based visualizations: general graphs, trees, and dendrograms. However, we also found that based on how a tree or dendrogram was organized (*e.g.,* radially, or from left to right), it was difficult to discern whether a particular visualization should be labeled as a tree, graph or dendrogram. The fundamental structure of these visualizations were the same. As such, we consolidated all three into a single "graph" label.

"Stacked," "Grouped," and "Filled" Visualization Variations: We observed several variations on common visualization types, such as bar visualizations (*e.g.,* grouped and stacked bar visualizations) and area visualizations (*e.g.,* filled and stacked area visualizations). We found that even though these visualizations had some minor differences, the visualizations were still very similar in structure. As such, we grouped each set of variations into a single visualization type. For example, all bar variations were grouped as "bar" and all area variations as "area".

Applying Labels

Manual labeling was performed for all 1247 visualizations in the bl.ocks.org collection. To do this, snapshots were taken of the SVG objects, and were labeled by the authors. Irrelevant visualizations were ignored.

For the Plotly collection, each visualization had an HTML`title` tag embedded within the corresponding webpage that included the visualization type. The `title` object was extracted for each Plotly web page, and filtered for the visualization type. In addition, Plotly allows one to export the code used to render the visualization in multiple programming languages. Python code was exported for each Plotly visualization, and the Python code was filtered to remove coordinated views. Plotly does not currently support animations, user inputs in visualizations, or arbitrary diagrams that do not map directly to a corresponding data table, so these issues were not a concern for the Plotly collection.

0.6.3 Experimental Setup

For each experiment, the overall accuracy is calculated over ten separate runs of stratified, five-fold cross validation. With five-fold cross-validation, each dataset is shuffled and partitioned into five groups (or folds). For each fold, the classifier is trained on the other four folds (or 80% of the data), and tested on the fold that is left out (20% of the data). We use stratified cross validation, to ensure that each visualization type contains an equal number of visualizations in every fold. To calculate our accuracy results, we sum the number of correct answers across all folds and runs, and divide this sum by the total number of records evaluated (*i.e.,* $\mid D_C \mid * 5$, where $\mid D_C \mid$ is the size of collection C).

Considering Dataset Skew

When a visualization collection is skewed (*i.e.,* a small number of visualization types represent the majority of the collection), a classifier can achieve moderately high accuracy by successfully classifying only the most popular visualization types. However, this bias can result in poor classification accuracy for more obscure visualization types, which is problematic for the designers interested in studying them. As such, it is critical that our classification component can accurately label most visualization types.

To account for skew in the number of visualizations per visualization type, we performed two experiments for each visualization collection: *skewed* and *skew-free*. In the skewed experiment, a maximum number of 400 visualizations is allowed per visualization type, and skew is permitted across visualization types (*i.e.,* any number less than or equal to 400 is an acceptable number of visualizations, for any visualization type). We capped the number of visualizations per type to 400, so we could capture skew in the dataset without overwhelming the number of samples for the less common types. The sample size limit of 400 was selected based on the D3 dataset, where the visualization type with the most samples had around 400 visualizations. For both experiments, all visualization types are included in the evaluation.

In the skew-free experiment, we ensured that an equal number of visualizations were provided for each visualization type. Specifically, we measure classification accuracy when exactly 10 visualizations are selected

		Skewed	Skew-Free	
		-	10	20
bl.ocks.org	Beagle	83.2	73.8	83.7
	RV	62.9	38.9	46.4
Plotly	Beagle	97.1	86.5	91.2
	RV	88.5	54.7	65.2

Table 9: Multi-class classification accuracy results (in percent) for Beagle and for ReVision (RV), for both of our visualization collections. Both dataset configurations were evaluated (skew and skew-free).

per visualization type. Any visualization type that has an insufficient number of visualizations is ignored in this experiment (*i.e.,* less than 10). For example, word clouds in the bl.ocks.org collection are ignored in this experiment, because there are only 6 examples available. The evaluation is then repeated with a larger input set, where twice as many visualizations are provided per visualization type (20 per type). This setup ensures that no classifier can rely on achieving high accuracy for only a small number of visualization types.

0.6.4 Comparing with ReVision

To evaluate how Beagle improves on existing classification techniques, we compare Beagle to a well-known raster image classifier: the ReVision visualization classifier [31]. The ReVision code was provided by the ReVision authors. Here, we provide a brief overview of the ReVision classifier.

ReVision classifies raster images of visualizations by first building a dictionary, or a *codebook*, of distinctive image patches that occur frequently in visualizations. ReVision then uses the codebook to extract features from images for visualization classification. In these experiments, the ReVision codebook is built using the training data only in our five-fold cross validation.

The Plotly collection is similar to the original collections used to evaluate ReVision [31]. As such, ReVision is expected to perform as well for Plotly as it has in the past (*i.e.,* about 80% classification accuracy). However, the bl.ocks.org collection has double the number of visualization types (22 types, versus 10 types [31]), where many of them are difficult to discern from other types. Thus, we expect ReVision to have lower accuracy for the bl.ocks.org collection.

0.6.5 Classification of the bl.ocks.org Collection

The goal of these experiments is to evaluate for a targeted, yet extremely diverse corpus (with 22 visualization types), how accurately we can classify visualizations. The bl.ocks.org collection is expected to be the most challenging to classify, because it not only covers the majority of visualization types observed in previous work (8 out of 10 types [31]), but also doubles the number of visualization types that must be classified. Thus, the bl.ocks.org collection represents the worst-case scenario in our evaluation. In these experiments, we tested both our skew-free and skewed dataset configurations.

Skewed Data Classification

This experiment evaluates how well Beagle performs (and how Revision performs) when classifying all 22 visualization types in the entire bl.ocks.org collection. The results are provided in Table 9, labeled "bl.ocks.org/Beagle/Skewed". Beagle achieves 82.7% accuracy overall, which is similar to past work [31, 32, 26], but here we classify double the number of visualization types (22 visualization types, 1247 total visualizations), where many of them are challenging to discern from other types.

Skew-Free Classification

Table 9 shows the accuracy results for classifying the bl.ocks.org collection using our proposed approach (Beagle). For the skew-free setup, we modified the bl.ocks.org collection such that there was an equal number of visualizations available per visualization type, before running Beagle (*i.e.,* 10 visualizations per type, and 20 visualizations per type). Visualization types with insufficient numbers of visualizations (*i.e.,* less than 10, or less than 20, respectively) were not evaluated in this experiment.

The first row of Table 9 shows the overall accuracy of Beagle (labeled "bl.ocks.org/Beagle/Skew-Free"). With only 10 visualizations per visualization type (10 visualizations, 19 visualization types, 190 visualizations total), Beagle is able to correctly classify 73.2% of visualizations. With 20 visualizations per visualization type (14 visualization types, 280 visualizations total), Beagle's accuracy improves to 81.9% accuracy, showing that Beagle is effective over a diverse set of visualization types.

0.6.6 Classification of the Plotly Collection

These experiments evaluate how well our classification techniques apply to visualizations rendered by a completely different non-JavaScript tool (Plotly).

Skewed Classification

In this experiment, no more than 400 visualizations are selected for any given visualization type, resulting in 2133 visualizations being selected for classification. Note that a different random sample is selected for each cross-validation run. As shown in Table 9 under "Beagle/Plotly/Skewed", Beagle achieves 97.1% accuracy overall, providing significantly higher accuracy compared to the bl.ocks.org collection (82.8% accuracy).

Skew-Free classification

The classification accuracy results are provided in Table 9, labeled "Beagle/Plotly/Skew-Free". Beagle provides high classification accuracy across the Plotly collection. For the 10 visualizations case (11 visualization types, 110 visualizations total) the accuracy is 85.3%, while for the 20 visualizations case (9 visualization types, 180 visualizations total) the accuracy improves to 93.6%. We also see higher classification accuracy compared to the bl.ocks.org collection, which is expected, because Plotly has fewer visualization types to classify, and consists almost entirely of visualization types that have been classified with high accuracy in previous work.

The most important features for classifying Plotly visualizations are the features that calculate statistics on the length of the "d" attribute for the `path` element. Given that Plotly relies almost entirely on `path` elements to render visualizations, and the "d" attribute provides the most information about a given `path` element, this result is unsurprising. The feature that calculates the maximum distance between the start and end points of a `path` is also instrumental for improving classification accuracy. Calculating the distance between the start and end points of a `path` can help differentiate "open" `path`s (*e.g.,* lines and curves) from "closed" `path`s (*e.g.,* rectangles).

Given Beagle's high accuracy on both collections, we conclude that our classification techniques are tool-agnostic, and can be used to classify visualizations made using drastically different tools.

0.6.7 Beagle and ReVision

We repeat our previous experiments using ReVision, and discuss the differences in accuracy between Beagle and ReVision, and why these differences occur.

Skewed Classification

All visualization types are evaluated in this experiment. The ReVision classifier achieves 62.9% accuracy on bl.ocks.org, and 88.5% on Plotly. In contrast, Beagle achieves 82.8% accuracy overall for the bl.ocks.org collection, and performs even better on the Plotly collection (97.1%). ReVision's low classification accuracy on the bl.ocks.org collection is due to a combination of low-resolution information (*i.e.,* pixels instead of graphical marks) and a high number of complex visualization types. When a significant number of complex visualization types are introduced (*i.e.,* going from Plotly to bl.ocks.org), distinctive features become more subtle, and a complex combination of fine-grained features are required to discern visualizations. However, it is more challenging in general to identify distinctive visualization characteristics from the raw pixels of raster images than from precise graphical objects. Thus, ReVision also performs worse than Beagle on the Plotly collection.

Skew-Free classification

We found that the ReVision classifier performs significantly worse than Beagle in our skew-free setups. This is unsurprising, considering the ReVision classifier utilizes computer vision techniques to identify and label visualizations, and many computer vision techniques require large numbers of examples to be effective. We observed similar results for the Plotly collection, though ReVision generally had 20% higher accuracy on the Plotly collection than the bl.ocks.org collection. The need for a large training set is further demonstrated in ReVision's significantly higher classification accuracy on the Plotly collection for the skewed case (2133 visualizations total), compared to the skew-free cases (110 - 180 visualizations total).

0.6.8 Combining the Collections

Though our classification techniques are effective for our individual collections, it is also important for Beagle to accurately classify visualizations when confronted with more than one tool, and even non-visualization images. In these experiments, we evaluate our classification performance when combining the two collections (bl.ocks.org and Plotly), as well as when including non-visualization images.

To combine the collections, we selected equal numbers of visualizations from each collection, for each visualization type. For types that only appear in one collection, we include everything from this collection (*e.g.,* all contour charts from Plotly). For types with more examples in one collection than the other, we sampled from the larger collection to match the size of the smaller one. Using our skew-free setup, we find that our classification techniques provide 88% overall accuracy.

We also evaluate whether Beagle can differentiate between visualizations and SVG images that are not visualizations. To do this, we randomly select 250 visualizations from each collection (500 total), and 500 non-visualization images from over 120 different websites. We train a binary classifier on the 1000 images using Beagle's SVG-based features. When evaluating with five runs of five-fold cross validation, we find that our classification techniques provide 95% overall accuracy. Using this binary classifier, Beagle can employ a two-stage classification process for corpora containing both visualizations and SVG images that are not visualizations. The binary classifier can be applied in the first stage to filter for non-visualization images, then Beagle's multi-class classifier can be applied in the second stage to classify the remaining visualizations.

0.7 Storing and Querying Visualizations

To help designers quickly and easily search for visualizations within our collection, we propose a Data Store to automatically format the data collected by our Web Crawler and Annotator into a database that can be queried using a well-known query language: SQL. In this section, we describe the database schema we apply to our extracted visualizations dataset, show how one can execute queries using the schema, and present three interesting ways to query the Data Store.

0.7.1 Schema

Our schema supports a variety of analyses, while still being easy to extend, since SVG adoption will likely grow in the future. The schema contains three groups of tables: Identifier Tables, Mapper Tables, and Information Tables. Here, we explain the purpose for each table within the schema.

Identifier Tables

We translate repetitive text labels (*e.g.,* visualization types, SVG element types, etc.) to integer ID's, and use Identifier Tables to translate between the two. Any Identifier Table is easily extended by appending new records to the table. Existing text ID's can easily be edited directly in the corresponding Identifier Table,

without modifying other tables in the schema. Each Identifier Table has two columns: one for ID's, and another to store the corresponding text label.

Collection Types: for collection types (bl.ocks.org and Plotly).
Visualization Types: for visualization types.
Element Types: for SVG element types.
Attribute Types: for SVG element attribute types (*e.g.,* x position, y position, width etc.).
Style Attribute Types: for style attributes (*e.g.,* `fill`, `stroke`, `stroke-width`, etc.).

Mapper Tables

Mapper Tables track relationships between entities, such as mappings between visualizations and their corresponding SVG elements, or between each SVG element and its attributes. As such, Mapper Tables only contain relationship pairings, and thus completely ignore details for individual entities (*e.g.,* the position or size of an SVG element). Each table consists of two columns of ID's, one column for each side of the pairing.

Collection-Visualizations: maps each collection to its corresponding list of visualizations.
Visualization-Elements: maps each visualization to its corresponding list of elements.

Information Tables

Information Tables store detailed data about each visualization and SVG element that appears in the dataset. All of the tables are designed to be easy to extend and modify.

Visualizations: This table contains information about each visualization in the collection: the visualization ID, visualization type ID, height, width, library used to create the visualization, and corresponding url. Each row in the table corresponds to a single visualization. If a visualization belongs to multiple types, the visualization will be listed once for each type it has. For example, if a visualization consists of a scatter plot overlaid on a geographic map, the visualization will be listed twice, once with the type ID for scatter, and once with the type ID for geographic map.
Elements: This table contains information about every SVG element in the collection: the element ID, and element type ID. Each visualization corresponds to one record in the Visualizations Table.
AttributesThis table contains information about specific attributes for the SVG elements stored in the Elements Table. Since the attributes associated with different types of SVG elements can vary widely, we create one row in the Attributes Table for each attribute specified in a given SVG element. Thus, for each SVG element in the Elements Table, we create a number of records in the Attributes Table equal to the number of attributes specified in the SVG element. Each row records: the attribute type ID, the SVG element ID, and a string representing the value of to the attribute. A units column is added because SVG attributes may have varying units (e.g. the "stroke-width" attribute of a `path` element may be specified in "px" or "em".)

Style Attributes: This table is similar to the Attributes Table, with style attributes instead of SVG attributes. A units column is also included, since units often vary in style attributes.

Schema

Figure 0-8 demonstrates how these tables work together to provide a broad view of the dataset. The orange tables (`element_types` and `attribute_types`) are Identifier tables. The green table (`visualization_elements`) is a Mapper table. The black tables (`elements` and `attributes`) are Information tables. Figure 0-9 shows examples of a few tables in our database. The first row of the `attributes` table contains the `width` attribute of a single `rect` element, which is stored in the first row of the `elements` table. We also see which visualization contains this `rect` element in the `visualization_elements` table (*i.e.,* visualization 1). The `element_types` and `attribute_types` tables are used to identify the `rect` and `width` types, respectively.

0.7.2 Using the Schema

In this section, we describe how to write SQL queries that use our database schema.

The Identifier tables are useful for enumerating possibilities within the dataset. For example, to count the number of different visualization types in the dataset, one can perform a count query on the Visualization Types table: `SELECT count(*) FROM visualization_types`.

The Mapper Tables are designed to support efficient group-and-count style queries. For example, to count the total number of visualizations per collection, one can group the Collection-Visualizations table by collection ID, and count the number of visualizations in each group: `SELECT collection_id, count(*) FROM visualizations GROUP BY collection_id`.

Similarly, to count the total number of `circle` elements in each visualization, one can filter the Visualizations-Elements Table for `circle` elements, then group by visualization ID's (assuming `circle` elements have ID=1): `SELECT visualization_id, count(*) FROM visualizations_elements WHERE element_id=1 GROUP BY visualization_id`.

Furthermore, we can perform joins between Identifier and Mapper Tables to filter queries using human-readable text labels, rather than numerical identifiers. For example, to filter the previous query using the string "circle" instead of the ID for type `circle` (*i.e.,* 1), one can perform an equality join with the Element Types table, using the element ID's as join keys: `SELECT ce.visualization_id, count(*) FROM visualizations_elements AS ce, element_types AS et WHERE ce.element_id=et.id AND et.label = "circle" GROUP BY ce.visualization_id`.

The Attributes Tables provide the ability to filter and group elements using attribute values, allowing one to perform more sophisticated analyses.For example, for each visualization, suppose we want to calculate the maximum number of `rect` elements with identical widths. To do this, we first group the `rect` objects by visualization ID and width:

```
SELECT ce.visualization_id,
```

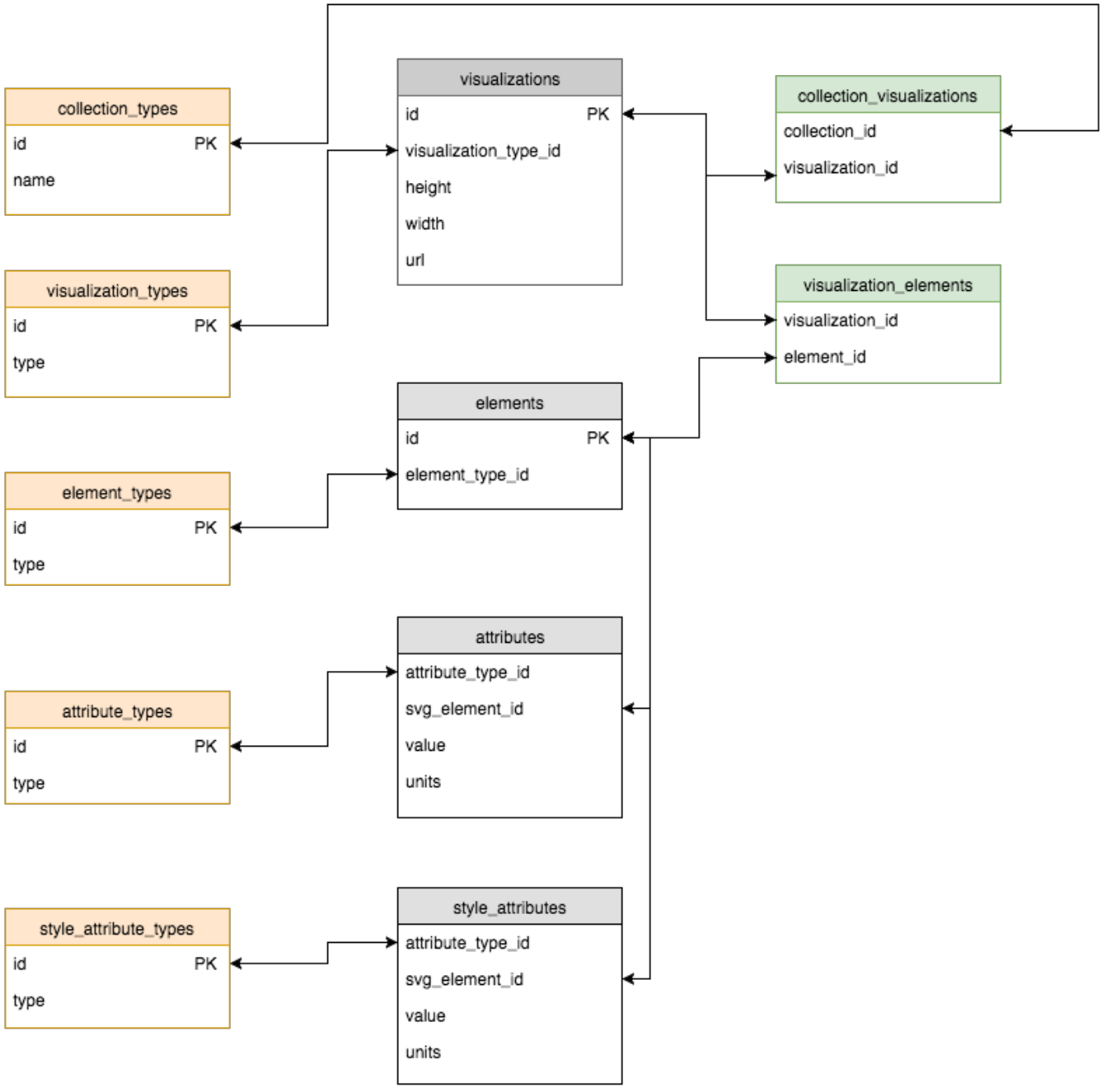

Figure 0-8: The schema of the database, with arrows pointing to corresponding elements that are used to connect different tables. Columns labeled with "PK" are the primary keys.

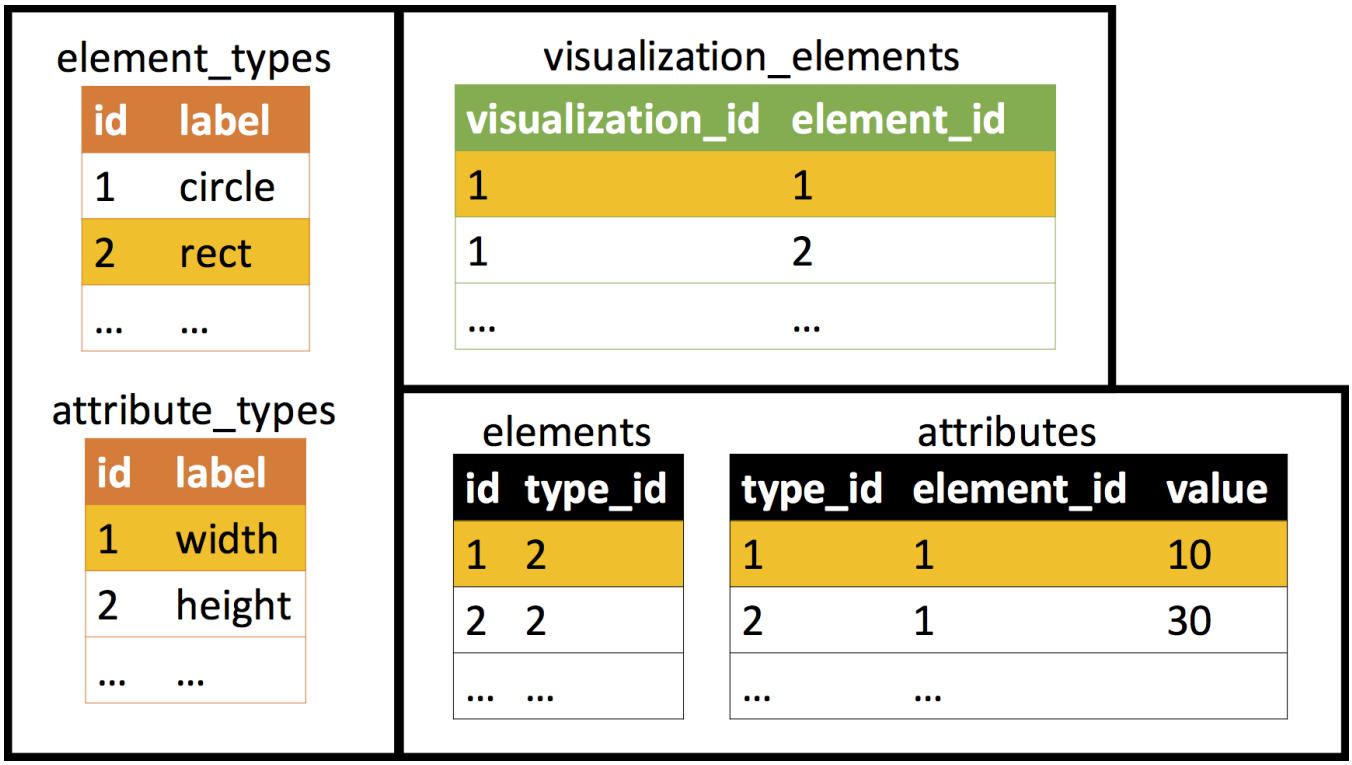

Figure 0-9: An example of what the data might look like in a subset of our database tables. The highlighted rows show how these five tables connect.

```
   a.value AS width, count(*)
FROM visualization_elements AS ce,
   attributes AS a,
   element_types AS et,
   attribute_types AS at
WHERE ce.element_id=et.id
   AND a.attribute_id=at.id
   AND ce.element_id=a.element_id
   AND et.type="rect"
   AND at.type="width"
GROUP BY ce.visualization_id, at.label
```

Call this *query1*. To extend *query1* to calculate the maximum number of `rect` elements with identical width, we perform an additional group by query, using *query1* as a subquery: `SELECT visualization_id, max(width) FROM ( [query1] ) GROUP BY visualization_id,`

```
SELECT visualization_id, max(width)
FROM ([query1])
GROUP BY visualization_id
```

where `[query1]` denotes where *query1* above should be inserted as a subquery.

0.7.3 Interesting Queries

In this section, we present several queries to perform on the data store that may be of interest to users. They are the number of hidden elements in each visualization (where "display" is set to "none" or "visibility" is set to "hidden" or "collapse"), the average number of colors used per chart type, and the average number of `circle` elements per chart type. We give the SQL code, an analysis of the results, and a performance measurement of each the of queries.

Number of hidden elements per visualization

As we collected D3 visualizations, we observed that many of the ones with hidden elements were animated or interactive, which inspired this first query. A case where hidden elements are present in an interactive chart is a geographic map where `circle` elements of varying radii appear over regions of the map when a user clicks on a button to display population by region. Thus, hidden units can be used as a way to detect whether or not a chart supports user interaction or is animated. Note, there may be false negatives as not all charts with interactivity or animations have hidden units (e.g. animated charts where the components do not appear/disappear). An analyst who wants to study only non-animated and non-interactive charts would find this method useful in filtering out charts that do not fit these criteria. Furthermore, a visualization designer could use this hidden unit detection method to determine how frequently charts of a given type are animated or interactive, and for charts that are animated or interactive, how many hidden units are used, and utilize these pieces of information to make design decisions, such as whether to make the chart animated/interactive. An interesting extension to this query would be to find counts of the types of SVG elements that are hidden, to see what types of elements are most commonly hidden.

The query shown below calculates the average number of hidden elements present in SVG charts by type for charts with at least one hidden element. It selects all "style" attributes corresponding to hidden objects ("display", "visibility") and selects all these attributes whose value indicates that the element is hidden (e.g. when "visibility" is set to "hidden"). These attributes are matched with the elements they modify and then grouped by visualizations the elements belong in. The number of hidden elements per visualization is counted, the visualizations are grouped by type, and the average hidden element count is taken by type. Note that this query looks only at the "style_attributes" table (see the Datastore schema in Figure 0-8). SVG elements have a "visibility" attribute that also handles their display, and is stored in the "attributes" table. An element is also hidden if its "visibility" attribute is set to "hidden" or "collapse". Thus, to get an accurate average count of all hidden elements per chart by type, we would need to union this query on "style_attributes" shown below with a nearly identical query on the "attributes" table that inspects the "visibility" attribute. In general, since certain SVG features can be specified by either SVG attributes or within the "style" attribute (such as element visibility), we plan to update the schema so the "attributes" and "style_attributes" tables are merged into a

single table, which would simplify attribute querying.

```
SELECT visualization_types.type, AVG(get_hidden.total_hidden)
FROM visualization_types, visualizations,(
  SELECT visualizations.id as vis_id, COUNT(distinct visualization_elements.element_id)
  AS total_hidden
  FROM visualization_types, visualizations, visualization_elements, style_attributes,
  style_attribute_types
  WHERE visualizations.id = visualization_elements.visualization_id
  AND visualization_elements.element_id = style_attributes.svg_element_id
  AND style_attributes.attribute_type_id = style_attribute_types.id
  AND ((style_attribute_types.type = 'visibility'
  AND (UPPER(style_attributes.value) = 'HIDDEN'
  OR UPPER(style_attributes.value) = 'COLLAPSE'))
  OR (style_attribute_types.type = 'display'
  AND upper(style_attributes.value) = 'NONE')) group by vis_id) AS get_hidden
WHERE visualization_types.id = visualizations.visualization_type_id
AND visualizations.id = get_hidden.vis_id
GROUP BY visualization_types.type;
```

This query took 23.4 seconds to run. The full query, which includes a union with a very similar query on the "attributes" table, should take approximately double this time. The results of the full query are shown in Figure 0-10. In addition, the results of a query that counts the total number of charts by type that had at least one hidden element, where the six chart types with the highest fractions are shown in Figure 0-11. Full results of these queries are showing Tables and in the appendix.

Based on the results in Figures 0-11 and 0-10, the fraction of charts with hidden units and the average number of hidden units for charts with hidden units vary significantly by chart type. Sunbursts and parallel coordinates had the highest fraction of charts with hidden units. Since presence of hidden units is correlated with animation or user-interactivity, these results signal that a high percentage of sunbursts and parallel coordinates in our dataset are animated or interactive. We looked at the 15 sunbursts that had hidden units, and found that 14 of them were animated or interactive. The animations and interactivity featured relevant text or other graphical features appearing when users hovered over the sunburst, and additional slices appearing as users hovered over the visualization. Figure 0-12 illustrates an example of an interactive sunburst with hidden units, where slices and text appear as the user hovers over the visualization, and the link to this sunburst is "http://bl.ocks.org/fabiovalse/0dfe7280086553c4a233". However, there were a total of 21 sunbursts with animations and interactivity, which means 7 of them do not have hidden units. These sunbursts consisted of animations where the slices changed positions or disappeared after some user action, so there are no units

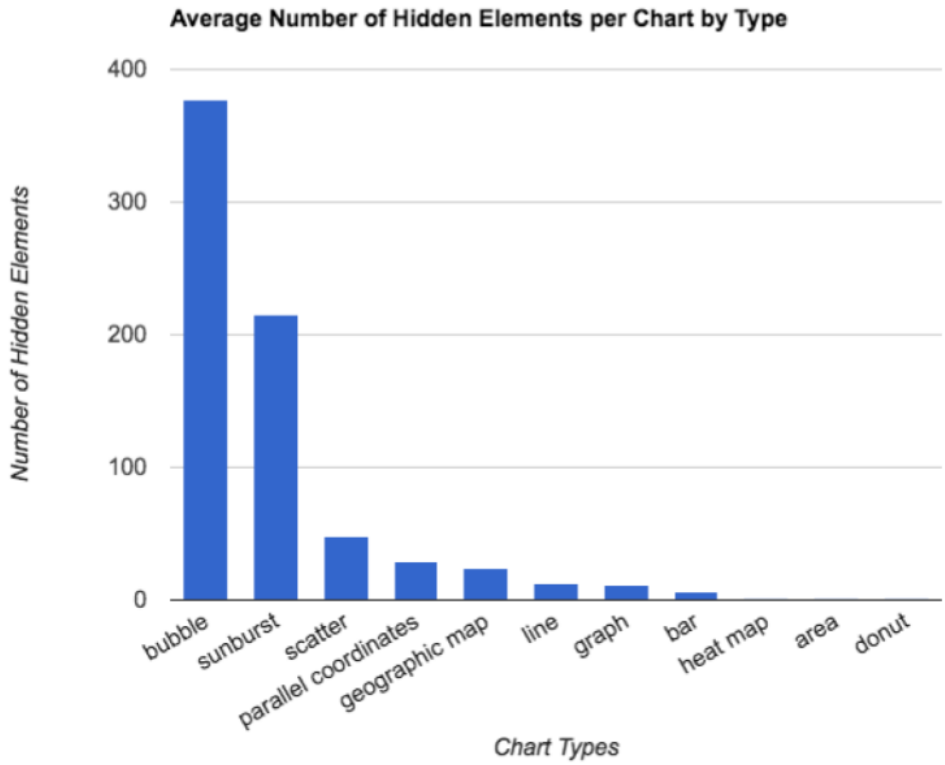

Figure 0-10: A bar chart comparing the average number of hidden units per chart (with at least one hidden unit) for each chart type.

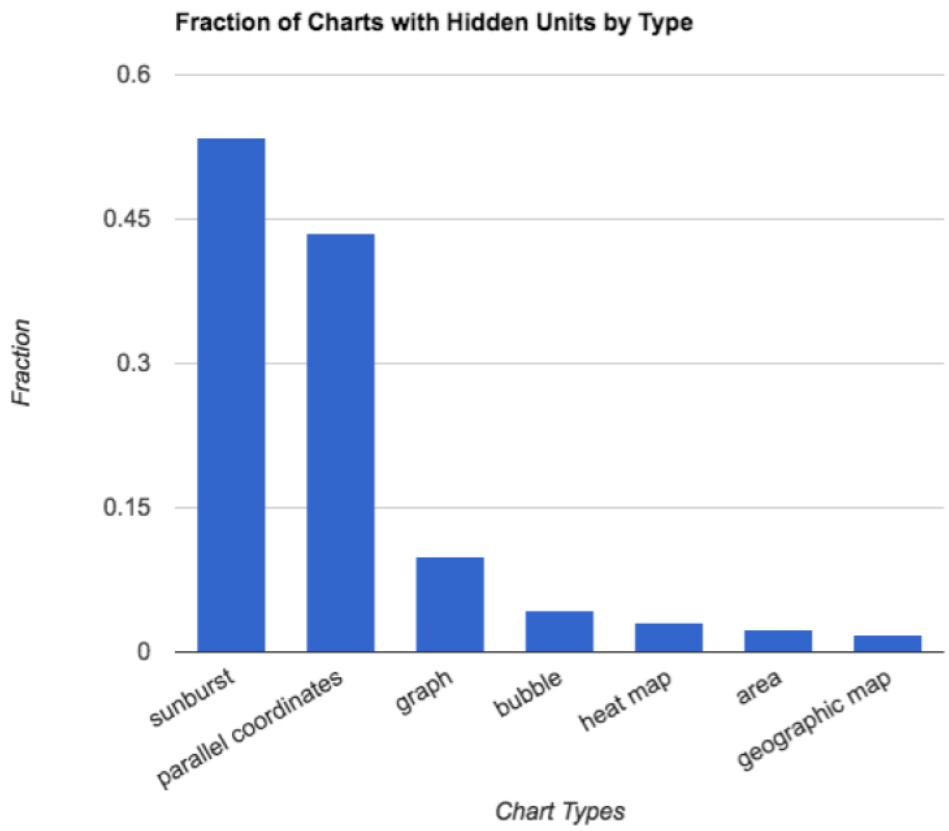

Figure 0-11: A bar chart comparing the top six chart types with the highest fraction of charts with hidden units.

that are hidden initially. We looked at the 10 parallel coordinates charts with hidden units and found that they were all interactive, where when the user highlights a portion of the axes, and a previously hidden rectangle appears and all lines the visualizations that did not fall within the rectangle vanishes. The process is shown in Figure 0-13, and visualization can be found at "http://bl.ocks.org/syntagmatic/4446b31c6cd746eedaeb". There are 12 parallel coordinates visualizations that were animated or interactive, and the visualizations without hidden units had lines that changed colors over time, but no graphical objects were hidden. Based on the analysis of these two chart types, we could assume that having hidden units is a strong indicator of user interactivity or animation, but we will have to look at a larger sample size with more chart types from a variety of different sources to verify this conclusion. One possible explanation for the high fraction of sunbursts and parallel coordinates with hidden units is that samples for these two types all come from D3 only (see Table 9), and D3 has great support for incorporating animation and interactivity, and these additional features work especially well for sunbursts and parallel coordinates. Parallel coordinates often have a lot of lines, so giving users the option to select which lines to display increases interpretability, as shown in Figure 0-13. Having relevant text show up as the user hovers over a region of the sunburst conveys additional information to the user without cluttering up the already multifarious sunburst, as illustrated in Figure 0-12. The Plotly library has less support for animation and user interactivity, so Plotly charts are less likely to be animated or interactive. Geographic maps and bubble charts have samples from Plotly, which is why their fraction of charts with hidden units are lower. Although heatmaps come from D3 only, heatmaps are best represented without animations/interactivity, since they rely on blocks of varying hues and gradation of colors to represent data, and animations and user interactions may distort this display of data.

Out of the charts with hidden units, sunbursts and bubble charts have the highest numbers of hidden units (see 0-10). Since sunbursts and bubble charts often contain vast numbers of individual graphical markers, which are usually the target of animations/user interactivity, these charts have the highest numbers of hidden units per chart. Since scatter plots and bubble charts use a marker for each individual data point, and markers are usually the target for animations/interactivity, one would expect these to charts to have similar counts of hidden elements. A possible explanation for this is that the bubble charts with hidden elements have more markers than scatter plots with hidden elements, since there are not very many bubble charts with hidden units neither type has very many samples with hidden elements according to Figure 0-11 and Table 18. Namely, bubble charts have 0.04*70 $\approx$ 3, while scatter plots have 0.003*3167 $\approx$ 10.

Average number of colors used per chart type

The average number of unique colors is interesting for studying general trends in color usage across chart types. Visualization designers can use these trends as guidance when deciding on the number of colors to include in their visualization. For example, suppose a user want to create a parallel coordinates chart, and is deciding whether or not to give each line a distinct color. Upon seeing that the average number of distinct colors used in a parallel coordinates chart is very low, the designer reasons that the lines in parallel coordinates

Figure 0-12: Illustration of a sunburst with hidden slices and text that appear when the user hovers over a portion of the visualization. The visualization is initially blank, as shown in the top image, and hidden slices and text start appearing as the user hovers over portions of the graph, as illustrated in the bottom two images.

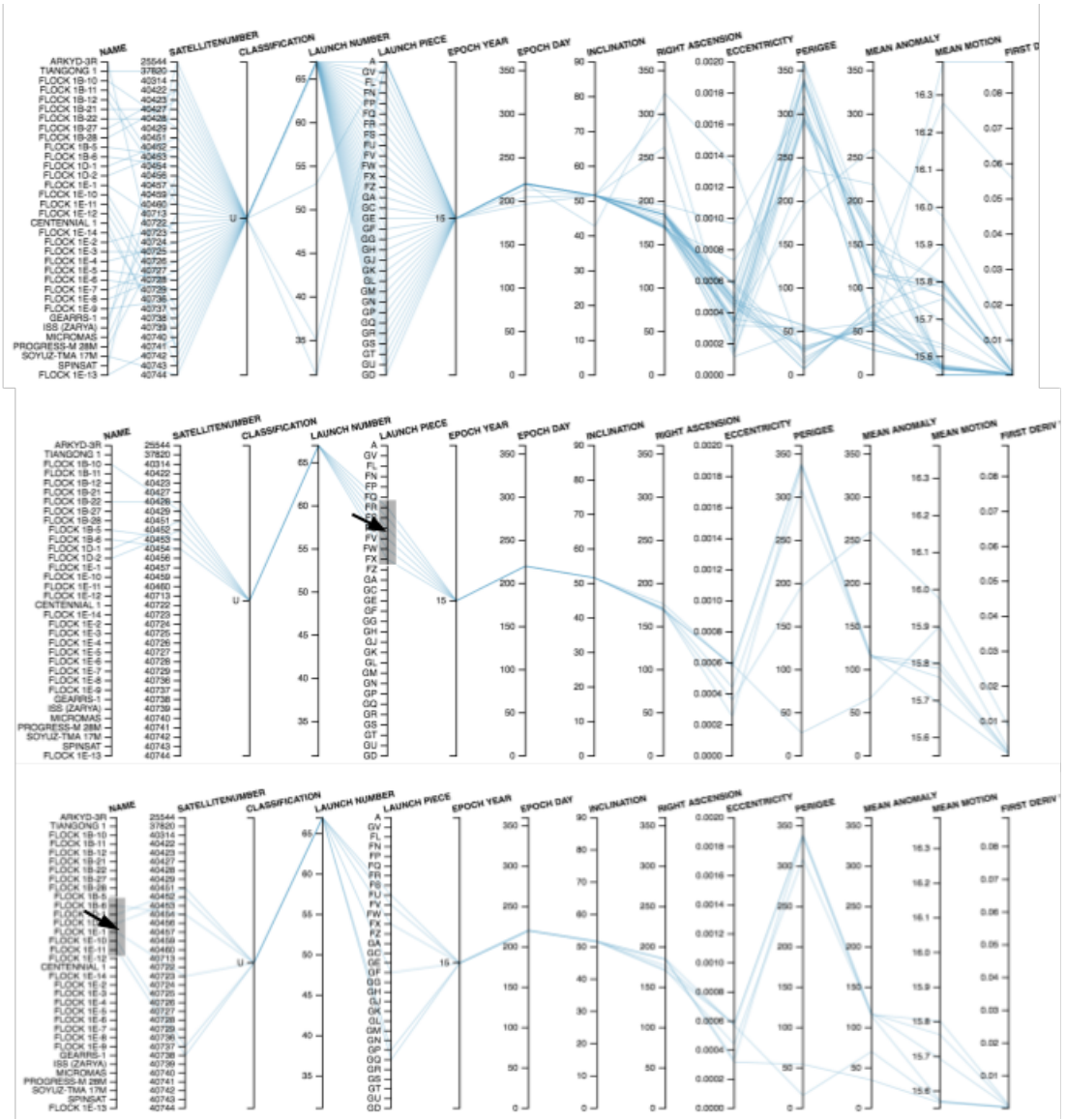

Figure 0-13: Illustration of a parallel coordinates diagram that uses a hidden rectangle to select which lines to display. All lines are visible initially, as shown in the top image, and a rectangle appears as the user selects which lines to display, as illustrated in the bottom two images.

charts are usually the same color, and uses this information to decide on the color scheme of one's chart. There are charts with color information specified in external CSS and store no color information in their SVG. For this query analysis, we filtered out charts with no color specified in the SVG.

To perform this query, "style" attributes associated with color, which are of type "stroke" and "fill" for "style" are selected from "style_attributes". Their attributes are then matched to their corresponding SVG elements, and grouped by the visualization the elements belong in. A count of the distinct values (that are not "none") for these colorful attributes used in each visualization is performed. Note that that only visualizations with at least one color specified appear in this group. The visualizations are then grouped by type, and the average count is taken. This SQL query on "style_attributes" table only is shown below. Similar to the previous query on hidden element count, there are SVG attributes associated with color in the "attributes" table, namely attributes of type "fill", "stroke", and "color". To attain an accurate account of all unique colors used per chart, one would need to union this query on "style_attributes" with a nearly identical query on "attributes".

```
SELECT visualization_types.type, AVG(get_colors.total_colors)
FROM visualization_types, visualizations, (
  SELECT visualizations.id as vis_id, COUNT(distinct style_attributes.value) AS total_colors
  FROM visualizations, visualization_elements, style_attributes, style_attribute_types
  WHERE visualizations.id = visualization_elements.visualization_id
  AND visualization_elements.element_id = style_attributes.svg_element_id
  AND style_attributes.attribute_type_id = style_attribute_types.id
  AND (style_attribute_types.type = 'stroke' or style_attribute_types.type = 'fill')
  AND UPPER(style_attributes.value) != 'NONE'
  GROUP BY visualizations.id) AS get_colors
WHERE visualization_types.id = visualizations.visualization_type_id
AND visualizations.id = get_colors.vis_id
GROUP BY visualization_types.type;
```

This query took 191.5 seconds to complete. The full query, which includes a union with a very similar query on the "attributes" table, should take approximately double this time. The results of the full query are shown in Table 20 in the appendix, and the seven visualization types with highest numbers of distinct colors per chart is illustrated in Figure 0-14.

Note the counts in Figure 0-14 include colors for all components of the graph, including axes, borders, tic-marks, etc. Based on these results, the number of colors used vary a lot by chart type with choropleth, hexabin, and heat map all using a huge number of different colors. This is reasonable since these chart types use color to represent a dimension of data and hence, make use of a myriad of different hues and color gradation. This results in a huge number of different colors used in these chart types. The results also show

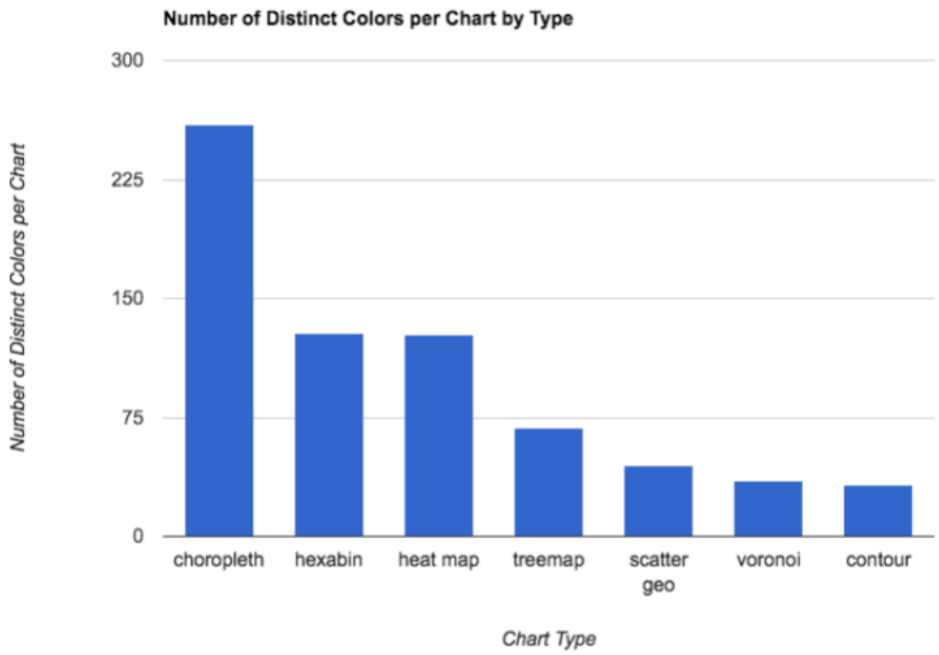

Figure 0-14: A bar chart comparing the top seven chart types with the highest counts of distinct colors per chart.

that similar chart types use very close numbers of distinct colors, which is to be expected. According to Table 20 in the appendix, donut and pie both used around 10, and while scatter and bubble had a difference of 5, which is insignificant and could be due to the nature of the chart samples for each type.

Since we filtered out charts with color specified externally, we thought it would be interesting to count the number of such charts by type. This information is useful to visualization designers in that it helps them figure out how to style their code. For example, when a designer plans on creating a geographic map and sees that it is common for geographic maps to have color specified outside of the chart SVG, the designer realizes it is perhaps better to place the color in an external CSS file for geographic maps.

The first part of this query, namely extracting all the visualizations with color specified in the SVG, is identical to the above SQL query that calculates the average number of distinct colors. Like before, all attributes associated with color are located, matched to their corresponding element id, and then grouped by the visualizations containing these colorful elements. These visualizations have color specified internally and must be filtered out, so we select all visualizations not in this group of visualizations with color set internally, and group the selected visualizations by type. The number of visualizations per type is then counted. Like before, the SQL query for extracting from "style_attributes" only is shown below. To attain an accurate count of visualizations with colors specified externally, one would need to select all visualizations not in the union of the two queries extracting visualizations with color specified internally from "style_attributes" and from "attributes".

```
SELECT visualization_types.type,count(*)
FROM visualization_types, (
  SELECT visualizations.id AS vis_id2, visualizations.visualization_type_id
  AS type_id2
  FROM visualizations
  WHERE visualizations.id NOT IN (
    SELECT visualizations.id AS vis_id
    FROM visualizations, visualization_elements, style_attributes, style_attribute_types
    WHERE visualizations.id = visualization_elements.visualization_id
    AND visualization_elements.element_id = style_attributes.svg_element_id
    AND style_attributes.attribute_type_id = style_attribute_types.id
    AND (style_attribute_types.type = 'stroke' OR style_attribute_types.type = 'fill')
    AND upper(style_attributes.value) != 'NONE'
    GROUP BY vis_id)) AS get_external
WHERE visualization_types.id = get_external.type_id2
GROUP BY visualization_types.type;
```

This query took 54.7 seconds to complete. The full query should take approximately double this time. The results of the full query is shown in Table 0-15 in the appendix, and the six charts types with highest counts of charts with color specified externally illustrated in Figure 0-15.

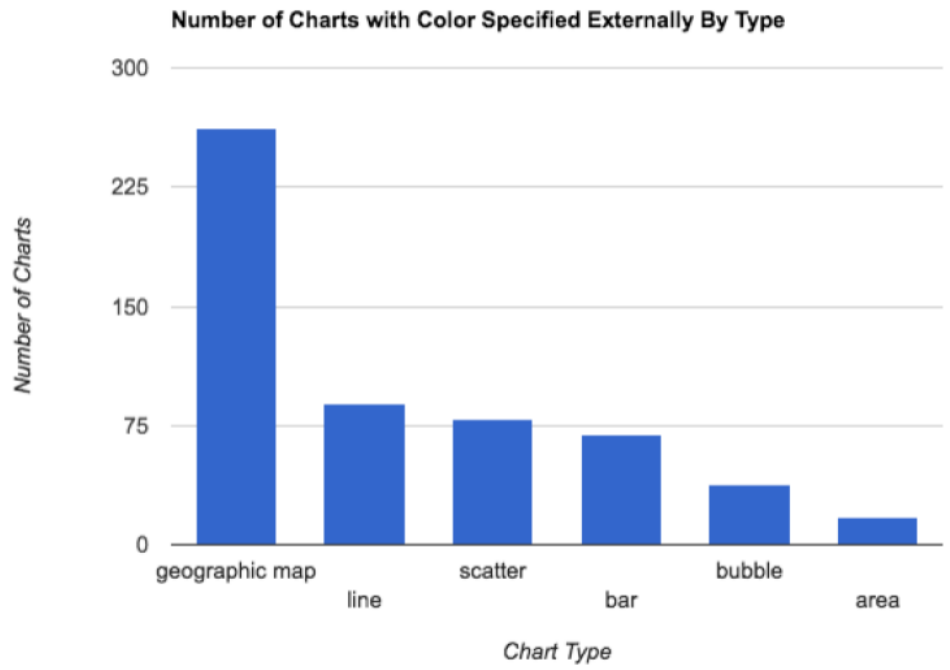

Figure 0-15: A bar chart comparing the top seven chart types with the highest counts of distinct colors per chart.

Geographic map had the highest count of charts with colors specified (262) externally according to 0-15. Given that there are 563 geographic maps total according to Table 7, this is roughly 45 percent of all geographic maps in the dataset. While there are a lot of geographic maps (379) according to Table 7, there are considerably more line and bar charts both of which had significantly fewer charts with color specified externally. Geographic maps usually have graphical elements overlaid on maps (e.g. circles over map regions where the `circle` size indicates the population size of the region). Since maps are very difficult to create from scratch, as they require a lot of detailed `path` elements, creators of geographic maps likely attained the map portion as an image or SVG from elsewhere. It could be possible that a large portion of the 379 geographic maps on D3 (Table 7) reused the SVG map creation code from a small sample of charts on D3, and all these charts specified color in external CSS, and the geographic map creators that reused these map SVG code followed this coding style

of specifying color externally. It is also possible that these chart creators created a separate SVG object to contain the graphical elements overlaid on the copied SVG map, and decided to improve code organization by having a separate external CSS snippet specifying color for both SVG objects. Line charts and scatter plots had the next two highest external color specification counts, but there are significantly more charts of each type (1355 and 3049, respectively), which is roughly 6.5 percent and 2.6 percent, respectively. These percentages are insignificantly small and could reflect the percent of users who chose to specify the colors externally for code organization reasons. This number is low because SVG elements have attributes that enable users to specify colors easily. For instance, `circle` SVG elements have a "fill" attribute that users can set to specify the `circle`'s fill color.

Average number of `circle` elements per chart type

The average number of `circle` SVG elements used for each chart type is interesting to visualization designers because they would like to see how prevalent `circle` elements are for the chart type they're creating. This information will help them decide what components of their visualizations could be represented by `circle` elements based on what others have done in their respective visualizations. For example, assume the designer is creating a graph, and is unsure of how to represent the vertices. Upon seeing the high prevalence of `circle` elements, the designer deduces that `circle` elements are the popular choice for representing graph vertices, and proceeds to make a decision based on this information. This element usage statistic could be extended to all graphical SVG elements (`path`, `rect`, etc.) to provide users with a comprehensive summary of the types of elements and their prevalence for each chart type, which will help designers decide what SVG elements to use in the visualizations they are creating.

To calculate the average number of `circle` elements, we first counted all the `circle` elements in the "elements" table. We then grouped all the `circle` elements by the id of the visualization they belong in and counted the number of `circle`s for each visualization. We set the count to 0 for visualizations without `circle` elements. We then grouped the visualization `circle` counts by visualization type and took the average. The SQL statement for this query is as follows:

```
SELECT avg(circle_count.coalesce), visualization_types.type
FROM (
  SELECT coalesce(circles.count,0), visualizations.id, visualizations.visualization_type_id
      FROM (
      SELECT count(*), visualization_id
      FROM visualization_elements
      WHERE element_id in (
        SELECT elements.id
```

```
        FROM elements, element_types
        WHERE elements.type_id = element_types.id AND element_types.type = '<circle>')
      GROUP BY visualization_id) AS circles
    FULL JOIN visualizations on visualizations.id = circles.visualization_id)
    AS circle_count
  INNER JOIN visualization_types ON visualization_types.id = circle_count.visualization_type_id
GROUP BY visualization_types.type;
```

This query took 7.65 seconds to complete, and output of this query gives the average number of `circle` elements per visualization for each chart type. The seven chart types with the highest counts of `circle` elements per chart is shown in Figure 0-16. Full results are shown in Table 22 in the appendix.

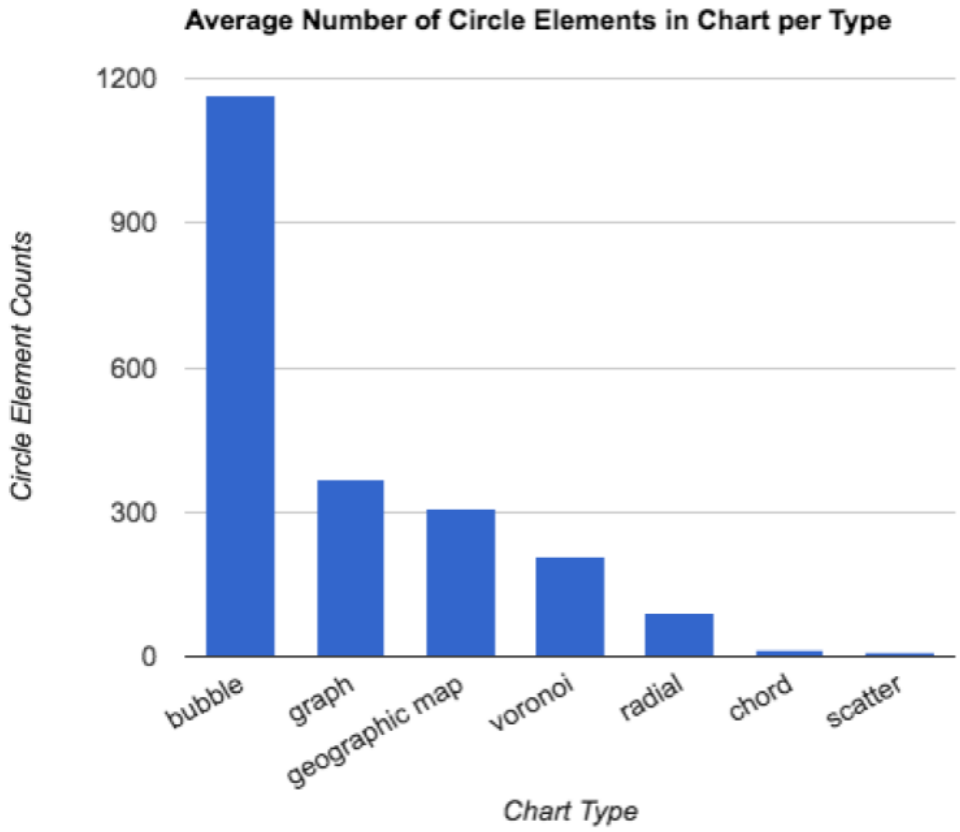

Figure 0-16: A bar chart comparing the top seven chart types with the highest counts of `circle` elements per chart.

Based on the results in Figure 0-16, the number of `circle` elements used varies greatly by chart type with bubble, graph, geographic map, and voronoi all having high counts of `circle` elements. This is reasonable since `circle` elements of varying radii are the markers representing data in bubble charts, and `circle`

elements are valid choices for vertices in graphs, and for demarcating cities and centers of mass for geographic maps and voronoi charts, respectively. The results also show that `circle` elements are a popular choice for vertices in graphs. Surprisingly, scatter plots have a relatively low `circle` count. One would expect their `circle` counts to be comparable to that of bubble charts, since they both use circular markers to represent data. This discrepancy is due to the fact that Plotly uses `path` elements to represent all graphical markers, and hence, the scatter plot markers are `path` elements for Plotly charts. There is huge number of Plotly scatter charts (3049 according to Table 9) in the dataset and no bubble charts, as shown in Table 1, while there are only 118 D3 scatter charts that actually use `circle` elements to represent data. Hence, this significant skew towards Plotly scatter charts brought the average `circle` count for scatter charts down. We should note that the results of the query on our dataset is not representative of trends on the general web, as Plotly is heavily biased towards `path` elements, and Plotly charts constitute a significantly majority of our dataset.

0.8 Related Work

Beagle spans several areas of research: web mining, data-driven design, and automated visualization classification. In this section, we highlight related projects in these areas.

Within web mining, Beagle is closely related to systems that: 1) extract visualizations or web page designs, and 2) make the resulting data accessible via query interfaces. Regarding web page design, Webzeitgeist [22] and D.Tour [29] provide a query interface to explore web page designs. Within visualization, Saleh *et al.* [30] propose techniques for similarity-based search of infographics collected from the web. However, to the best of our knowledge, Beagle is the first to automatically identify and extract visualizations from general webpages.

Beagle is motivated by several existing data-driven design systems. The most relevant involve: 1) automated re-design of web pages and visualizations, and 2) computational models to describe specific design elements. The Bricolage system learns mappings between web page designs, and uses them to transfer the layout of one page to another. The ReVision system [31] extracts data from bar and pie charts, and re-renders this data using different visualization types. Reinecke *et al.* model the visual complexity and colorfulness of web pages, in order to predict first impressions of a web page. Harrison *et al.* [18] extend this approach to infographics. Beagle provides automated techniques to build the large input corpora needed by data-driven visualization techniques.

Beagle also extends existing work in the automated classification of visualizations [32, 31, 26, 20]. Most projects focus on classifying raster images of charts [31, 26, 20]. Prasad *et al.* [26] and The ReVision system [31] apply computer vision techniques to extract features from bitmap images for classification. Huang and Tan [20] extract graphical marks from raster images, and use them to classify visualizations. However, extracting data from bitmap images is imperfect, and only works for a small number of chart types [31]. Shao and Futrelle [32] analyze vector images from PDF's to classify 36 images, representing five visualization types. We extend existing work by calculating a suite of statistics over SVG as classification features. Our features apply to any SVG image, and can be used to accurately classify thousands of visualizations, and as many as 22

different visualization types.

0.9 Discussion and Conclusion

We presented Beagle, an automated system for collecting, labeling and analyzing visualizations created on the web. The Beagle architecture has three stand-alone components. The Web Crawler filters web pages to locate and extract SVG-based visualizations, and was used to extract over 17000 visualizations. The Annotator uses novel SVG-focused classification techniques, and achieves 82% classification accuracy, in a multi-class classification test with 22 visualization types. The Data Store translates the final collection to relational format for storage in a database, and supports searching of the database using a well-known query language (SQL). However, the system can be improved further. In this section, we discuss two limitations to Beagle, and areas for future work.

One drawback to the Beagle Web Crawler is that a human has to specify Beagle code snippets for each visualization library of interest. As a result, a human has to manually update Beagle when a new visualization library needs to be added. A future direction could be to change how these code snippets are identified, to allow for automation. One possible approach is to search the JavaScript code for variables that correspond to SVG objects in the DOM, and use these variables as indicators of code that could be generating visualizations.

There are also limitations in the analyses our Data Store supports. For example, when performing a retrospective analysis of visualizations, one lacks the ability to extract the motivation for why these visualizations were created. One way to address this is to analyze the text of the html page containing each visualization. But it is generally challenging to infer a person's motivation for creating a particular visualization after-the-fact.

0.9.1 Future Work

The goal of Beagle is to enable data-driven visualization design. Beagle can collect and classify thousands of visualizations, and can be extended with existing techniques [31, 17] to collect the corresponding datasets for these visualizations. The extracted visualization-dataset pairs can then be used as input for machine learning techniques, to predict how new datasets should be visualized in the future.

Another avenue for future work is to study how visualizations are designed through code. Our Web Crawler can be augmented to save additional assets from webpages. Visualization techniques could be captured through JavaScript code and then consolidated into "summary" snippets, similar to the techniques used in the Overcode system [15]. With design "summaries", designers could look up the most popular ways to create certain visualizations, and study differences (and similarities) in designs across visualization types.

Chart Type	Count	Libraries
chord	2	Raphael
bar	45	D3, Raphael
heatmap	1	D3
area	7	D3
graph	11	D3, Raphael
treemap	1	R3
scatter	27	Jit, Raphael
geographic map	17	D3, Raphael
donut	41	D3, Raphael
sankey	1	D3
line	27	D3, Jit, Raphael
bubble	9	D3, Raphael

Table 10: Table showing the counts of each visualization type and the libraries creating that type of all extracted SVG visualizations from the collection of 20 million urls.

0.10 Appendix

.1 General Web Search Results

We decided to look into the urls we filtered to make observations on how people visualize charts on the general web. We performed a preliminary analysis on 10 million urls selected randomly from the Common Crawl url index, and discovered that 25.4k webpages called a chart-making library. However, only 20.5k of these urls had an SVG element, and most of the pages (84.7 percent) with SVG elements had SVG charts, while the rest had logos and no charts. This means 19.3 percent of those who called a SVG chart library do not use it, which could be due to removal of logos and/or charts that were previously on the page, or preparation for future addition of logos and/or charts. We noticed that a lot of filtered urls came from the same small number of domains; we will refer to these domains as "repetitive domains". An example of a repetitive domain is "stackexchange.com", where vast number of filtered urls, such as "http://tex.stackexchange.com/users/888/trevis?tab=activity" had this domain. The 17.4k webpages that contain an SVG chart mostly came from repetitive domains and were hence, predominantly a single chart type. The repetitive domains were dominated by user profiles on stack exchange and other forums, and all these forums pages contain line charts that show the user's activity, as illustrated in Figure -17. A few other frequently occurring domains include weather and stock forecast sites that used line charts to illustrate their data. To attain a greater variety of visualization types from the general web, we decided to create a new dataset from the Common Crawl that did not include any urls with these repetitive domains. This dataset contained 20 million such urls that were sampled randomly. These urls were filtered through the code analyzer and yielded 510 total pages that contain chart-making code, and the HTML analyzer extracted 4720 SVG elements from these pages. However, a large portion (45.3 percent) of these SVG elements were still logos, and Table 10 below shows the counts for each visualization type found and the libraries creating that type in the set of extracted SVG visualizations.

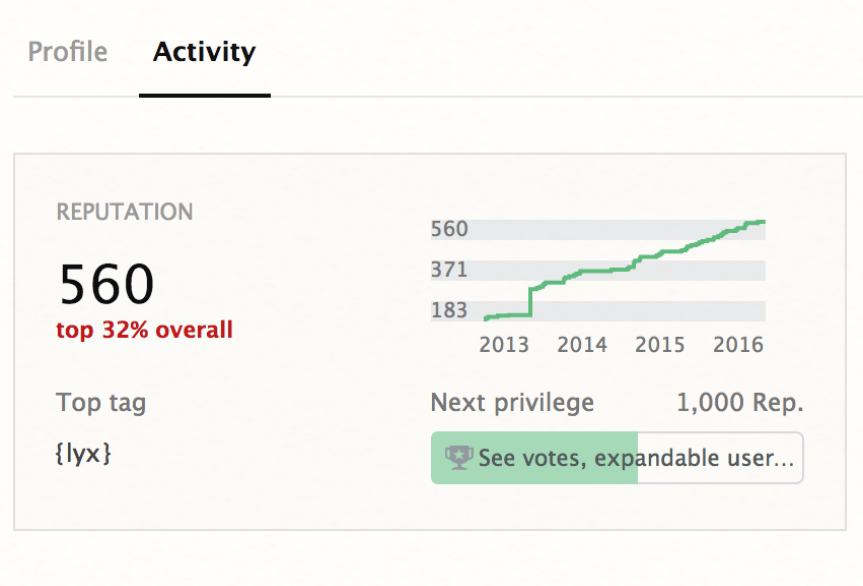

Figure -17: A line chart showing user activity from stack exchange, the most frequently occurring domain in the dataset, representing 73.8 percent of pages with SVG visualizations.

It can be concluded that a high percentage of users utilize these chart-making libraries to create logos for their websites, which makes these visualization libraries a convenient tool for SVG logo/icon creation. In addition, only three out of the eleven visualization libraries were used to create visualizations in the 20 million url collection according to Table 10, which means the other eight libraries are perhaps rarely used or utilized mostly for logo creation. D3 and Raphael were both used in large numbers of visualization types, which is to be expected since they are both well-known and support a myriad of different visualization types.

.2 Path "d" attribute

The "d" attribute of a `path` element is a string with instructions for drawing the trajectory of the `path` [13]. The string consists of a combination of the following instructions: moveto, lineto, curveto, arcto, and closepath. "Moveto" corresponds to picking up the drawing instrument and moving it to another position in the SVG canvas without drawing a line between the current and destination positions. It is specified as "M x,y", in the "d" attribute, where "x" and "y" are the destination coordinates that become the new current location. Most "d" attributes start with a "moveto" command to specify the starting position of the SVG element. "Lineto" is similar to "Moveto" except a line is drawn between the previous and current locations. "Lineto" is specified as "L x,y", where "x" and "y" are destination coordinates, and "H x" and "V y" are used for horizontal and vertical lines, respectively, where "x" specifies the x coordinate of the endpoint of the horizontal line, and "y" specifies the coordinate of the endpoint of the vertical line. "Curveto" draws quadratic and cubic bezier curves,and bezier curves are parametric curves drawn by polynomial functions [12]. Our features focus on quadratic curves only, and they are specified as "Q cx,cy,x,y", where "cx" and "cy" are x,y coordinates of the

control point, and "x" and "y" are x,y coordinates of the destination location (relative to the starting position). "Arcto" declares an elliptical arc and is specified as "A rx,ry, xAxisRotate, LargeArcFlag, SweepFlag x,y", where "rx" and "ry" specify the radius in the x and y directions, respectively, "xAxisRotate" changes the x-axis relative to the current reference frame, "LargeArcFlag" has a value of 0 if the smaller arc between the two endpoints is drawn and 1 if the larger arc is drawn. "SweepFlag" is 1 if the arc is swept in the clockwise direction and 0 for the counter-clockwise direction. Finally, "x" and "y" are the coordinates of the arc's end (destination) point. "ClosePath" draws a straight line from the current position to the first point in the path, creating a closed shape. It is specified with a "z" in the "d" attribute.

.3 Single Feature Descriptions and Classification Accuracies

This section contains tables describing all 119 features we came up with, including the features that were trimmed. In addition to the brief description, the accuracy of the classifier using that feature only is included.

Feature	Feature Description	D3	Plotly
Vertical Axes Count	Number of vertical axes lines	0.43736968724939856	0.35208626347866856
Horizontal Axes Count	Number of vertical axes lines	0.4202085004009623	0.3734756349797303
Circle Count	Count of all circle elements	0.3945469125902165	0.19484294421003281
Maximum Circle X	x-coordinate of the circle with the largest x-coordinate	0.371130713712911	0.1949367088607595
Minimum Circle X	x-coordinate of the circle with the smallest x-coordinate	0.36150761828388134	0.19484294421003281
Average Circle X Count	Average count of each x-coordinate by occurrence	0.3626303127506014	0.1949367088607595
Circle X Count Variance	Variance in count of each x-coordinate by occurrence	0.33039294306335204	0.19146741678387247
Circle Unique X Count	Count of all unique x-coordinates for centers	0.3635926222935044	0.1944678856071261
Maximum Circle Y	y-coordinate of the circle with the largest y-coordinate	0.36728147554129914	0.19503047351148617
Minimum Circle Y	y-coordinate of the circle with the smallest y-coordinate	0.3696872493985565	0.19465541490857946
Average Circle Y Count	Average count of each y-coordinate by occurrence	0.3563753007217322	0.1945616502578528
Circle Y Count Variance	Variance in count of each y-coordinate by occurrence	0.3355252606255012	0.18931082981715894
Circle Unique Y Count	Count of all unique y-coordinates for centers	0.3648757016840417	0.1945616502578528
Average Circle Fill Color Count	Average count of each fill color by occurrence	0.3504410585404972	0.18912330051570558
Circle Fill Color Count Variance	Variance over count of each fill color by occurrence	0.3339214113873296	0.1875293014533521
Maximum Circle Radius	Radius of circle with the largest radius	0.3870088211708099	0.19465541490857946
Maximum Circle Radius Count	Count of the most frequently occurring radius value	0.38941459502806736	0.1945616502578528
Circle Radius Variance	Variance over all radii	0.3457898957497995	0.19362400375058603

Table 11: Definitions of all circle and axes features used followed by the classification accuracy of using that feature only on the D3 and Plotly dataset. Note all attributes mentioned refer to the SVG element targeted by the feature only. For example the "Average count of each x-coordinate by occurrence" description for the "Average Circle X Count" feature refers the the x-coordinates of circle elements only.

Feature	Feature Description	D3	Plotly
Path Count	Total number of path elements	0.47506014434643146	0.5719643694327239
Filled Path Count	Total number of path elements with fill	0.3151563753007217	0.6913267698077825
Maximum Path X	x-coordinate of the path element starting at the largest x-position	0.4415396952686447	0.6845757149554618
Minimum Path X	x-coordinate of the path element starting at the smallest x-position	0.42919005613472333	0.6503516174402251
Average Path X Count	Average count of each x-coordinate by occurrence	0.454691259021652	0.5723394280356305
Path X Count Variance	Variance in count of the x-coordinate for each starting position by occurrence	0.28291900561347233	0.5336146272855133
Path Unique X Count	Count of all unique x-coordinates for starting positions	0.4343223736968725	0.5838724800750117
Maximum Path Y	y-coordinate of the path element starting at the largest y-position	0.40064153969526867	0.6631973745897797
Minimum Path Y	y-coordinate of the path element starting at the smallest y-position	0.4202085004009623	0.6187529301453352
Average Path Y Count	Average count of each y-coordinate by occurrence	0.4068965517241379	0.5916549460853259
Path Y Count Variance	Variance in count of the y-coordinate for each starting position by occurrence	0.2814755412991179	0.5757149554617909
Path Unique Y Count	Count of all unique y-coordinates for starting positions	0.38011226944667204	0.5759024847632442
Minimum Path Stroke Width	Stroke width of the path element with the smallest stroke width	0.32028869286287087	0.4141584622597281
Average Path Stroke Color Count	Average count of each stroke color by occurrence	0.3839615076182839	0.5065166432255039
Path Stroke Color Count Variance	Variance in count of each stroke color by occurrence	0.3153167602245389	0.4251289263947492
Path Unique Stroke Color Count	Number of unique stroke colors used	0.3980753809141941	0.5807782466010314
Maximum Path Stroke Width	Largest stroke width used in a path element	0.41459502806736165	0.4344116268166901

Table 12: Table showing single feature descriptions and classification accuracy results for path elements.

Feature	Feature Description	D3	Plotly
Minimum Path d-attribute Length	Minimum number of characters in d-attribute out of all path elements	0.42502004811547717	0.5462728551336147
Maximum Path d-attribute Length	Maximum number of characters in d-attribute out of all path elements	0.5080994386527666	0.8324425691514299
Average Path d-attribute Length	Average number of characters in d-attribute of all path elements	0.4423416198877306	0.7250820440693858
Path d-attribute Length Variance	Variance over number of characters in d-attribute of all path elements	0.3546110665597434	0.5528363806844819
Maximum Closed Path d-attribute Length	Maximum number of characters in d-attribute out of all closed path elements	0.4372093023255814	0.8973277074542897
Minimum Closed Path d-attribute Length	Minimum number of characters in d-attribute out of all closed path elements	0.4056134723336006	0.6986404125644632
Average Closed Path d-attribute Length	Average number of characters in d-attribute of all closed (d-attribute ending with z character) path elements	0.4118684843624699	0.849789029535865
Closed Path d-attribute Length Variance	Variance over number of characters in d-attribute of all closed path elements	0.326383319967923	0.483075480543835
Average Path Fill Color Count	Average count of each fill color by occurrence	0.29895749799518845	0.6280356305672762
Path Fill Color Count Variance	Variance in count of each fill color by occurrence	0.30681635926222933	0.36802625410220347
Path Unique Fill Color Count	Number of unique fill colors used	0.3143544506816359	0.6986404125644632
Maximum Circular "A" Arc Counts per Path Element	Maximum number of circular arcs in any path element	0.3339214113873296	0.37674636661978433
Path Rx Variance	Variance over rx attributes of arcs in all paths	0.34947874899759424	0.296952648851383
Path Ry Variance	Variance over ry attributes of arcs in all paths	0.35028067361668	0.29554617909048286
Path "Q" Curve Count	Count of all "Q" Curves	0.33327987169206097	0.25419596812001877
Minimum "Q" Curve Count Per Path	"Q" Curve count for the path with the fewest "Q" Curves	0.40769847634322376	0.1940928270042194
Average "q "Arc Count Per Path	Average number of "Q" Curves per path	0.33360064153969526	0.2547585560243788

Table 13: Continuation of table showing single feature descriptions and classification accuracy results for path elements.

Feature	Feature Description	D3	Plotly
Rect Count	Count of all rect elements	0.41379310344827586	0.556305672761369
Maximum Rect X	x-coordinate of the rect with the largest x-coordinate	0.40272654370489175	0.5479606188466948
Minimum Rect X	x-coordinate of the rect with the smallest x-coordinate	0.3991980753809142	0.19034224097515237
Average Rect X Count	Average count of each x-coordinate by occurrence	0.3767441860465116	0.2505391467416784
Rect X Count Variance	Variance in count of each x-coordinate by occurrence	0.33793103448275863	0.4923581809657759
Rect Unique X Count	Count of all unique x-coordinates used	0.36150761828388134	0.4360056258790436
Minimum Rect Y	y-coordinate of the rect with the smallest y-coordinate	0.3927826784282277	0.18724800750117207
Maximum Rect Y	y-coordinate of the rect with the largest y-coordinate	0.4296712109061748	0.4840131270511017
Average Rect Y Count	Average count of each y-coordinate by occurrence	0.3942261427425822	0.4052508204406939
Rect Y Count Variance	Variance in count of each y-coordinate by occurrence	0.3624699278267843	0.4935771214252227
Rect Unique Y Count	Count of all unique y-coordinates used	0.38283881315156376	0.46591654946085326
Average Rect Fill Color Count	Average count of each fill color by occurrence	0.35797914995990376	0.49413970932958273
Rect Fill Color Count Variance	Variance in count of each fill color by occurrence	0.3307137129109864	0.47801218940459445

Table 14: Table showing single feature descriptions and classification accuracy results for rect elements.

Feature	Feature Description	D3	Plotly
Line Count	Total number of line elements	0.4513231756214916	0.19371776840131272
Maximum Line X	x-coordinate of the line starting at the greatest x-position	0.4423416198877306	0.19512423816221283
Minimum Line X	x-coordinate of the line starting at the smallest x-position	0.44266238973536487	0.19474917955930615
Average Line X Count	Average count of each x-coordinate by occurrence	0.44025661587810744	0.19437412095639944
Line X Count Variance	Variance in count of each x-coordinate by occurrence	0.4400962309542903	0.19474917955930615
Line Unique X Count	Count of all unique x-coordinates for starting points	0.4663993584603047	0.1941865916549461
Maximum Line Y	y-coordinate of the line starting at the highest y-position	0.4171611868484362	0.1949367088607595
Minimum Line Y	y-coordinate of the line starting at the lowest y-position	0.44346431435445066	0.1949367088607595
Average Line Y Count	Average count of each y-coordinate by occurrence	0.43817161186848436	0.19540553211439288
Line Y Count Variance	Variance in count of each y-coordinate by occurrence	0.4463512429831596	0.19503047351148617
Line Unique Y Count	Count of all unique y-coordinates for starting points	0.434803528468324	0.19474917955930615
Average Line Stroke Color Count	Average count of each stroke color by occurrence	0.336327185244587	0.19484294421003281
Line Stroke Color Count Variance	Variance over count of each stroke color by occurrence	0.3151563753007217	0.19165494608532582
Text Word Count	Number of text elements that are not numbers	0.4171611868484362	0.5379278012189405
Text Unique X Count	Number of unique x-coordinates used	0.3199679230152366	0.42090951711204877
Text Unique Y Count	Number of unique x-coordinates used	0.3190056134723336	0.4301922175339897
Maximum Text Font Size	Largest font size used in text elements	0.31836407377706494	0.2677918424753868
Minimum Text Font Size	Smallest font size used in text elements	0.31836407377706494	0.2717299578059072
Text Font Size Variance	Variance in font sizes over all text elements	0.3119486768243785	0.1925925925925926
Text Unique Font Size Count	Number of unique font sizes used over all texts	0.31980753809141943	0.370182841068917

Table 15: Table showing single feature descriptions and classification accuracy results for line and text elements.

Feature	Feature Description	D3	Plotly
Circle X Variance	Variance over all x-coordinates for circles	0.3629510825982358	0.19231129864041258
Circle Y Variance	Variance over all y-coordinates for circles	0.36808340016038493	0.1894045944678856
Minimum Circle Radius	Radius of circle with the smallest radius	0.37866880513231754	0.19474917955930615
Path X Variance	Variance over x-coordinates of starting positions of all paths	0.32862870890136325	0.6164088138771683
Path Y Variance	Variance over y-coordinates of starting positions of all paths	0.31242983159583	0.5618377871542428
Circular Arc Count	Total number of circular arcs (a arcs where rx = ry) in all path elements	0.3467522052927025	0.48251289263947494
Minimum Circular Arc Count per Path Element	Minimum number of circular arcs in any path element	0.4296712109061748	0.2904828879512424
Average Circular Arc Counts per Path Element	Average number of circular arcs per path element	0.34418604651162793	0.5129864041256447
Minimum Non-circular Arc Count per Path Element	Minimum number of non-circular arcs in any path element	0.4041700080192462	0.19071729957805908
Path Unique Rx Count	Count of unique values for the rx attributes of a arcs in paths	0.34017642341619886	48598218471636195
Path Unique Ry Count	Count of unique values for the ry attributes of arcs in paths	0.3408179631114675	0.48748241912798873
Maximum Number of "Q" Curves Per Path	"Q" Curve count for the path with the most "Q" Curves	0.2521331458040319	0.2521331458040319
Path z Count	Total number of occurrences of z in d attributes of all path elements	0.39855653568564553	0.6071261134552274
Maximum Path Displacement	Largest distance between the start and end points of a path	0.37658380112269446	0.7199249882794186

Table 16: Table showing unused single feature descriptions and classification accuracy results for circle and path elements.

Feature	Feature Description	D3	Plotly
Rect X Variance	Variance over all y-coordinates	0.38235765838011226	0.5443975621190811
Rect Y Variance	Variance over all y-coordinates	0.4234161988773055	0.5355836849507736
Minimum Rect Width	Width of the narrowest rect element	0.3648757016840417	0.1875293014533521
Maximum Rect Width Count	Count of the most frequently occurring width	0.3648757016840417	0.18734177215189873
Average Rect Width Count	Average count of each width value by occurrence	0.3648757016840417	0.18762306610407875
Rect Unique Width Count	Count of all unique widths	0.39599037690457095	0.4378809188935771
Rect Unique Height Count	Count of all unique height	0.39823576583801124	0.4481950304735115
Minimum Rect Height	Height of the shortest rect element	0.3648757016840417	0.4488513830285982
Maximum Rect Height Count	Count of the most frequently occurring height	0.3648757016840417	0.1875293014533521
Average Rect Height Count	Average count of each height value by occurrence	0.3648757016840417	0.1874355368026254
Line X Variance	Variance over all y-coordinates	0.43416198877305534	0.19465541490857946
Line Y Variance	Variance over all y-coordinates	0.44170008019246193	0.19503047351148617
Minimum Line Length	Length (distance between endpoints) of the shortest line	0.44891740176423417	0.19474917955930615
Maximum Line Length	Length of the longest line	0.45501202886928627	0.19362400375058603
Line Length Variance	Variance over lengths of all lines	0.3491579791499599	0.19484294421003281
Vertical and Horizontal Line Count	Total number of lines that are vertical or horizontal	0.42004811547714516	0.1949367088607595
Circle to Path Ratio	Ratio of the number of circle elements to the number of paths	0.32991178829190054	0.19540553211439288
Circle to Line Ratio	Ratio of the number of circle elements to the number of lines	0.3931034482758621	0.1945616502578528

Table 17: Table showing unused single feature descriptions and classification accuracy results for rect and line elements, as well as element count ratios.

.4 Datastore Query Results

This section contains the outputs of all the complex queries discussed in Section 0.7.3.

fraction	type
0.53571	sunburst
0.43478	parallel_coordinates
0.10000	graph
0.04286	bubble
0.03125	heat_map
0.02381	area
0.01847	geographic_map
0.00593	bar
0.00446	donut
0.00379	scatter
0.00369	line
0.00000	box
0.00000	chord
0.00000	contour
0.00000	filled_line
0.00000	hexabin
0.00000	pie
0.00000	radial
0.00000	sankey
0.00000	stream graph
0.00000	treemap
0.00000	voronoi
0.0000	waffle
0.0000	word_cloud

Table 18: Fractions of charts with hidden elements (left column) per chart type (right column)

avg	type
377.0000000000000000	bubble
214.7333333333333333	sunburst
47.9166666666666667	scatter
28.8000000000000000	parallel_coordinates
23.4285714285714286	geographic_map
13.2000000000000000	line
11.5000000000000000	graph
6.1111111111111111	bar
1.00000000000000000000	heat_map
1.00000000000000000000	area
1.00000000000000000000	donut

Table 19: Average number of hidden elements per chart with at least one hidden element (left column) for each chart type (right column)

avg	type
259.3241379310344828	choropleth
128.3181818181818182	hexabin
127.0588235294117647	heat_map
94.0536004331348132	other
68.3333333333333333	treemap
44.8470588235294118	scatter_geo
35.2571428571428571	voronoi
32.5724637681159420	contour
28.7647058823529412	sunburst
27.5000000000000000	word_cloud
22.2727272727272727	sankey
20.9743589743589744	bubble
20.1666666666666667	graph
18.0874316939890710	geographic_map
15.5980861244019139	filled_line
15.5226458321094989	scatter
12.7307692307692308	radial
10.9687500000000000	chord
10.8669724770642202	donut
10.0400000000000000	waffle
9.9090909090909091	pie
9.0782301810339194	line
7.8565400843881857	box
7.6725123903926801	bar
7.4375000000000000	stream_graph
6.2162162162162162	area
2.4285714285714286	parallel_coordinates

Table 20: Average number of distinct colors per chart (left column) with at least one color for each chart type (right column).

avg	type
262	geographic_map
115	other
89	line
79	scatter
69	bar
38	bubble
17	area
15	parallel_coordinates
14	voronoi
12	graph
6	donut
5	chord
5	box
5	radial
2	hexabin
1	sankey
1	stream_graph
1	filled_line

Table 21: Counts of chart per type (left column) with color specified externally (right column).

avg	type
1166.8048780487804878	bubble
368.9838709677419355	graph
308.0051020408163265	geographic_map
210.0857142857142857	voronoi
91.1000000000000000	radial
14.9117647058823529	chord
13.7221366204417052	other
8.8609244961395769	scatter
1.4504811791935680	line
0.97183098591549295775	box
0.76190476190476190476	area
0.63636363636363636364	hexabin
0.62500000000000000000	waffle
0.25000000000000000000	sunburst
0.15178571428571428571	donut
0.05699745547073791349	bar
0.00000000000000000000	word_cloud
0.00000000000000000000	contour
0.00000000000000000000	filled_line
0.00000000000000000000	stream_graph
0.00000000000000000000	treemap
0.00000000000000000000	choropleth
0.00000000000000000000	scatter_geo
0.00000000000000000000	parallel_coordinates
0.00000000000000000000	sankey
0.00000000000000000000	pie
0.00000000000000000000	heat_map

Table 22: Average number of circle SVG elements per chart (left column) for each chart type (right column).

Bibliography

[1] 2014. JavaScript InfoVis Toolkit. (Jan. 2014). http://philogb.github.io/jit/

[2] 2015. dygraphs. (June 2015). http://dygraphs.com/

[3] 2015. Springy - A force directed graph layout algorithm in JavaScript. (July 2015). http://getspringy.com/

[4] 2016. dimple - A simple charting API for d3 data visualisations. (March 2016). http://dimplejs.org/

[5] 2016. Sigma js. (Feb. 2016). http://sigmajs.org/

[6] Eytan Adar, Mira Dontcheva, James Fogarty, and Daniel S. Weld. 2008. Zoetrope: Interacting with the Ephemeral Web. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. ACM, New York, NY, USA, 239–248. DOI:http://dx.doi.org/10.1145/1449715.1449756

[7] Dmitry Baranovskiy. 2016. Raphael–JavaScript Library. (July 2016). http://dmitrybaranovskiy.github.io/raphael/

[8] M.A. Borkin, A.A. Vo, Z. Bylinskii, P. Isola, S. Sunkavalli, A. Oliva, and H. Pfister. 2013. What Makes a Visualization Memorable? *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (Dec. 2013), 2306–2315. DOI:http://dx.doi.org/10.1109/TVCG.2013.234

[9] Michael Bostock. 2016. Popular Blocks - bl.ocks.org. (Sept. 2016). http://bl.ocks.org/

[10] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. DOI:http://dx.doi.org/10.1109/TVCG.2011.185

[11] Kurt Bollacker et al. 2016. Common Crawl. (2016). http://commoncrawl.org/

[12] Patrikalakis et al. 2009. Definition of Bezier Curve and its properties. (2009). http://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node12.html

[13] Firefox. 2016. Paths. (2016). https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths

[14] Mozilla Foundation. 2016. Download Firefox – Free Web Browser – Mozilla. (2016). https://www.mozilla.org/firefox/

[15] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2 (March 2015), 7:1–7:35. DOI:http://dx.doi.org/10.1145/2699751

[16] W3C Working Group. 2011. Scalable Vector Graphics (SVG) 1.1 (Second Edition). (Aug. 2011). https://www.w3.org/TR/2011/REC-SVG11-20110816/

[17] Jonathan Harper and Maneesh Agrawala. 2014. Deconstructing and Restyling D3 Visualizations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 253–262. DOI:http://dx.doi.org/10.1145/2642918.2647411

[18] Lane Harrison, Katharina Reinecke, and Remco Chang. 2015. Infographic Aesthetics: Designing for the First Impression. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1187–1190. DOI:http://dx.doi.org/10.1145/2702123.2702545

[19] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. 2000. WebBase: A Repository of Web Pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 277–293. http://dl.acm.org/citation.cfm?id=347319.346288

[20] Weihua Huang and Chew Lim Tan. 2007. A System for Understanding Imaged Infographics and Its Applications. In *Proceedings of the 2007 ACM Symposium on Document Engineering (DocEng '07)*. ACM, New York, NY, USA, 9–18. DOI:http://dx.doi.org/10.1145/1284420.1284427

[21] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering* 9, 3 (2007), 90–95. DOI:http://dx.doi.org/10.1109/MCSE.2007.55

[22] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092. DOI:http://dx.doi.org/10.1145/2470654.2466420

[23] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 2197–2206. DOI:http://dx.doi.org/10.1145/1978942.1979262

[24] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[25] Plotly. 2016. Plotly | Make charts and dashboards online. (2016). https://plot.ly/

[26] V. S. N. Prasad, B. Siddiquie, J. Golbeck, and L. S. Davis. 2007. Classifying Computer Generated Charts. In *2007 International Workshop on Content-Based Multimedia Indexing*. 85–92. DOI:http://dx.doi.org/10.1109/CBMI.2007.385396

[27] Selenium Project. 2016. Selenium - Web Browser Automation. (2016). http://www.seleniumhq.org/

[28] Katharina Reinecke, Tom Yeh, Luke Miratrix, Rahmatri Mardiko, Yuechen Zhao, Jenny Liu, and Krzysztof Z. Gajos. 2013. Predicting Users' First Impressions of Website Aesthetics with a Quantification of Perceived Visual Complexity and Colorfulness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 2049–2058. DOI:http://dx.doi.org/10.1145/2470654.2481281

[29] Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R. Klemmer. 2011. D.Tour: Style-based Exploration of Design Example Galleries. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 165–174. DOI:http://dx.doi.org/10.1145/2047196.2047216

[30] Babak Saleh, Mira Dontcheva, Aaron Hertzmann, and Zhicheng Liu. 2015. Learning Style Similarity for Searching Infographics. In *Proceedings of the 41st Graphics Interface Conference (GI '15)*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 59–64. http://dl.acm.org/citation.cfm?id=2788890.2788902

[31] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. 2011. ReVision: Automated Classification, Analysis and Redesign of Chart Images. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 393–402. DOI:http://dx.doi.org/10.1145/2047196.2047247

[32] Mingyan Shao and Robert P. Futrelle. 2006. Recognition and Classification of Figures in PDF Documents. In *Proceedings of the 6th International Conference on Graphics Recognition: Ten Years Review and Future Perspectives (GREC'05)*. Springer-Verlag, Berlin, Heidelberg, 231–242. DOI:http://dx.doi.org/10.1007/11767978_21

[33] Tableau Software. 2016. Tableau Public. (2016). https://public.tableau.com/

[34] Fernanda B. Viegas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. 2007. ManyEyes: A Site for Visualization at Internet Scale. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (Nov. 2007), 1121–1128. DOI:http://dx.doi.org/10.1109/TVCG.2007.70577

[35] Hadley Wickham. 2009. *ggplot2: Elegant Graphics for Data Analysis*. Springer New York, New York, NY. http://link.springer.com/10.1007/978-0-387-98141-3

[36] Yan Ping Zhou and Chew Lim Tan. 2000. Hough technique for bar charts detection and recognition in document images. In *2000 International Conference on Image Processing, 2000. Proceedings*, Vol. 2. 605–608 vol.2. DOI:http://dx.doi.org/10.1109/ICIP.2000.899506