ASSISTING DATA EXPLORATION VIA IN-SITU ADAPTIVE VISUALIZATIONS

BY

JAEWOO KIM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Aditya Parameswaran

ABSTRACT

Visual analytics has been widely used by data scientists to shed light on complex problems. Despite the prevalence of many visual analytics tools that empower human decision making with data-driven insights, challenges still exist that hinder users from genuinely capitalizing on insights from visualizations. The two biggest challenges we identify are the lack of task support and disconnected workflow. Visual analytics tools lack task support because they do not actively suggest insights to the users, requiring users to pick each individual step during exploration manually. These tools also suffer from disconnected workflows by keeping interactive exploration via dashboards separate from data preparation and cleaning tools like computational notebooks.

To address these challenges, we introduce Lux, a visualization recommendation library that automatically generates useful insights for data exploration, and seamlessly integrates into a user's data exploration workflow by augmenting the Pandas library. In this thesis, we document the design decisions made and the implementation details of Lux as well as how users can easily unlock intelligent analytical capabilities by adding our library to their code. Furthermore, we share how predecessor visual analytics tools such as Zenvisage that we contributed to guided the development of Lux.

Keywords: Data analysis, Visualizations, Recommendations, Scientific Applications.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I want to express my deepest gratitude to my advisor, Professor Aditya Parameswaran, who has advised me throughout my entire research career at the university. He encourages me to collaborate with others to tackle exciting research problems and provides insightful advice when I am struggling. Under his guidance, I was able to achieve greater heights, both as an academic and as a person. It was my honor to witness and benefit from his outstanding leadership and expertise.

I want to thank Doris Lee, who taught me all the necessary skills to become a researcher in the data visualization field. She also played a crucial role in this thesis because we collaborated from the beginning to build Lux. I will always be grateful for her support and mentorship.

I express gratitude to the Computer Science Department, the College of Engineering, and the University of Illinois for providing all the necessary resources that I need to finish my studies and research.

Finally, I am grateful to my beloved family. They have always believed in me and rooted for my success. I will always be indebted to their love and kind consideration.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

In recent years, organizations can accredit their success to actionable decisions made using data-driven insights. Whether it is forming business strategies from user behavioral data or finding support for scientific hypothesis through experimental data, deriving value from data has been crucial to these successes. Obtaining such valuable insights requires data scientists to undergo the process of data exploration, where they manually search for patterns and anomalies in datasets. This process becomes tedious and unwieldy, especially when the dataset is large and complex. Not surprisingly, the number of visual analytics tools have also boomed in recent years due to a higher demand for automation and assistance during data exploration.

Traditional visual analytics systems fall into two major categories. They are either visualization support libraries such as ggplot2 [1] in R and matplotlib [2] in Python, or stand-alone data visualization platforms such as Tableau [3] and Spotfire [4]. These two forms of data exploration fall short in many aspects. In both cases, users must explicitly specify the visualizations of interest, which is based on the assumption that users already know what they want. Also, it is hard to generate and analyze collections of visualizations since these systems usually operate on a single visualization at a time. This task of examining multiple visualizations becomes overwhelming and error-prone as the number of visualizations scales with the size and complexity of the dataset. Moving towards a solution for these shortcomings, we focus on developing another flavor of visual analytics tools, namely visualization recommendation systems (VRSs).

In contrast to traditional visual analytics tools, the goal of VRSs is to proactively suggest potentially interesting patterns or insights about the dataset. Since the system delivers recommendations automatically, users can learn about their datasets without substantial effort or background knowledge, and it is easy to iterate through the recommended visualizations thanks to a system-computed ranking based on well-defined metrics. However, based on our own experience from developing such systems, we found two major hurdles for people to adopt VRSs in practice. One limitation of VRSs is the lack of features that support a variety of analytical tasks that are useful to data scientists. For example, many existing VRSs such as Zenvisage [5], Voyager [6], and SEEDB [7], are all different in terms of their recommendation mechanisms and cannot be considered as a comprehensive solution for a variety of use cases. Such limited task support for users means that recommendations from VRSs only work for a small audience.

Furthermore, a significant difficulty stems from the gap that exists in the current landscape of analytics tools. On one end, data scientists use computational notebooks like Jupyter or R studio to write code that performs operations such as data cleaning, data preparation, or modeling. On the other end, many VRSs exist as standalone web or desktop-based applications to support interactive analytics. The chore of importing and exporting data every time from one system to another to perform data analytics slows down experimentation and disrupts the workflow of data scientists. To address these challenges, we built Lux, a novel VRS, which represents our first attempt at tackling all the aforementioned challenges. To fix the problem of limited task support, Lux features a multi-purpose analytics framework that is extensible to a variety of use cases. To bridge the gap between current tools, we are bringing the power of visualization recommendations directly into Jupyter notebooks, a popular platform among data scientists for interactive computing. We believe that Lux is an important first step towards building VRSs that support the broader data science community.

In this dissertation, we break down our research contributions into multiple milestones.

1. During the development of Zenvisage, a VRS developed by our research group, we discovered various adoption challenges. We collaborated with participants over a year to build an extensive set of features to address some of these challenges. We share the barriers faced while developing and using Zenvisage, and how this experience guided the development of Lux.

2. We defined a set of core features for Lux that resolves challenges faced by traditional VRSs. In particular, we designed various types of recommendations that support more use cases compared to existing tools. We also take a modular approach in designing these features to encourage extensions that support new use cases.

3. We iterated on a prototype design based on the features defined. With the goal of creating a seamless workflow for data exploration, Lux was designed as a library that augments a popular data science ecosystem. We designed Lux as an extension to Pandas dataframes [8] supporting analytic capabilities to display visualizations and visual recommendations, which are then rendered in a Jupyter notebook.

The rest of this thesis is organized as follows:

- Chapter 2 gives context on related work on VRSs. We spend considerable time on our work on Zenvisage and how it motivated the design of Lux;

- Chapter 3 explains the core capabilities of Lux;

- Chapter 4 documents the development process of Lux from design stages to prototype implementation;
- Chapter 5 discusses future work pertaining to Lux;
- Chapter 6 concludes this thesis.

CHAPTER 2: RELATED WORK

2.1 THE CURRENT LANDSCAPE OF VISUAL ANALYTICS TOOLS

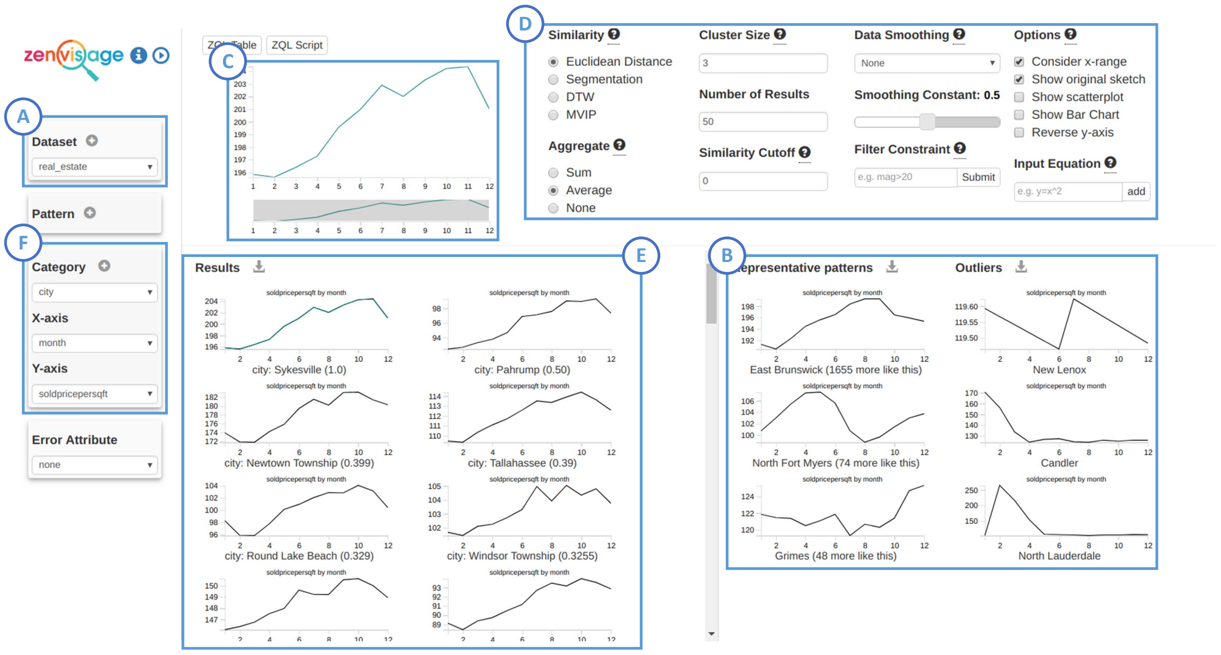
A number of tools have been recently developed to help people visualize and understand their data in intuitive ways. These tools, such as Spotfire [4], Polaris [9], and Tableau [3] are useful for analyzing a single visualization. For instance, Tableau integrates Show Me [10], a set of user interface defaults that automatically constructs a single visualization based on graphical design principles. However, a limitation of these systems is that they usually do not operate on multiple visualizations at a time. Since users often do not know what they are looking for, this flaw is critical. A solution to this problem is to recommend a collection of visualizations that may contain information that is potentially interesting to data scientists. For example, Voyager [6] is a system that recommends charts based on statistical and perceptual measures. SeeDB [7] shows views that highlight the difference between two sets of data. Profiler [11] automatically detects anomalies and suggests visualizations that provide valuable context. However, we believe that each attempt is fairly constrained and lacks flexibility. In particular, each system is limited to a single or a few types of recommendations, and the user has no control over which regions of the dataset or which visualization collections to explore. Furthermore, all of the VRS examples described in this chapter lack integration with the rest of the data science ecosystems to ease the burden of context switching between systems during exploration.

2.2 VISUAL QUERY SYSTEMS

Visual Query Systems (VQSs) are a subset of VRSs that allow users to specify a desired pattern via some high-level specification language or interface, with the system returning visual recommendations that match the specified pattern [5]. These systems usually include intuitive querying mechanisms like a sketchpad in their interface. Until the development of Lux, our research focused on VQSs because of their promise as data exploration tools. We felt that having an expressive querying mechanism that puts the user in control was the most important factor in aiding adoption.

2.3 ZENVISAGE

Zenvisage is a VQS developed by our group to address the challenge of exploratory analysis.



Dataset: real_estate
Category: city
X-axis: month
Y-axis: soldpricepersqft
Error Attribute: none

Similarity: [✓ Euclidean Distance] Segmentation DTW MVIP
Aggregate: Sum [✓ Average] None
Cluster Size: 3
Number of Results: 50
Similarity Cutoff: 0
Data Smoothing: None
Smoothing Constant: 0.5
Options: [✓ Consider x-range] [✓ Show original sketch] Show scatterplot Show Bar Chart Reverse y-axis

Figure 2.1: Zenvisage interface [12]

As shown in Figure 2.1, Zenvisage accepts various types of user input for generating visualizations. When a query is submitted in the frontend, Zenvisage computes the scores of candidate visualizations based on the query to identify matches. Then, visual recommendations are ranked and displayed in the output pane ordered by the scores. Based on Figure 2.1, let us study how the user interface features multiple interactive components. First, the user uploads a dataset using Panel A. Zenvisage then populates x, y, and z axis options based on the dataset schema (Panel F). Once attributes are specified, representative patterns and outliers are shown in Panel B. These particular visual recommendations can be generated in the absence of a query aiding a bottom-up exploration approach [13]. That is, the system automatically recommends visualizations from the data with the aim of provoking further data-driven inquiries. In contrast, users can control the outcome of their queries in a more fine-grained manner with a top-down approach, where users draw a pattern by hand or drag these patterns to the sketching canvas (Panel C). Here, the user plays a more active role in generating recommendations because they provide a high-level specification for how desired patterns should look like via the visual query. In Zenvisage, a large portion of the interface is dedicated to the top-down approach to support complex queries. For example, within Panel D, many options exist such as different comparison metrics for query processing like dynamic time warping (DTW) [14], Earth Mover's distance [15], and interactive smoothing. Finally, Panel E shows the ranked results of the query, listed from the highest to the lowest score.

Through a year-long user-centered development process of Zenvisage, we made incremental improvements to the system and received feedback from domain experts. Our work concludes that while VQSs support the capability of working with expressive visual queries and multiple visualizations, and did receive enthusiasm, interest, and initial usage from domain experts, VQSs such as Zenvisage are often not used for a sustained period of time due to several adoption challenges.

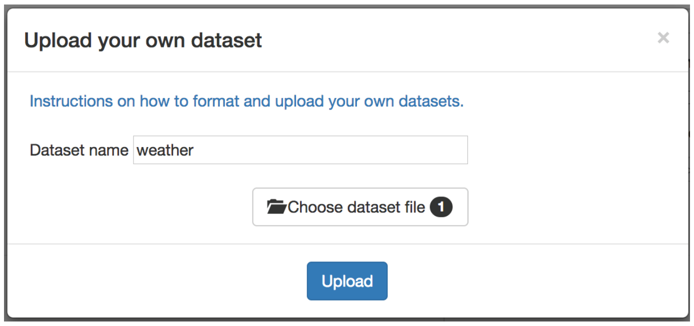
The most visible challenges were akin to those faced in systems mentioned above. VQSs still lack integration with the user's typical flow of analysis, leading to difficulties in context switching between two different systems. For example, our astronomy users could move on to VQSs only after all data preparation and processing operations were performed in a coding environment like a computational notebook or an IDE. In addition, the analytical capabilities of the system are confined to what has been implemented and released by the VQS developers, so there is no flexibility and room for extension by the users. Such a rigidity in the feature set poses challenges for supporting a variety of user tasks.

Finally, we share the results of a user study [13] conducted with domain experts. According to the study, a limitation of VQSs is the focus on top-down approaches for exploration, which are less relevant compared to bottom-up approaches. Our study shows that bottom-up approaches correspond to roughly 70% of the operations performed in the study, much larger than top-down approaches. The reason for this discrepancy is the necessity for preconceived knowledge in a top-down approach. That is, data scientists often do not have a clear idea of what desired patterns look like "in theory" to form an effective query. In contrast, bottom-up approaches require no prior knowledge and provide users with initial ideas to guide their search.

Next, we present two specific features of Zenvisage to provide context on the issues mentioned above.

2.3.1 Automatic Dataset Upload

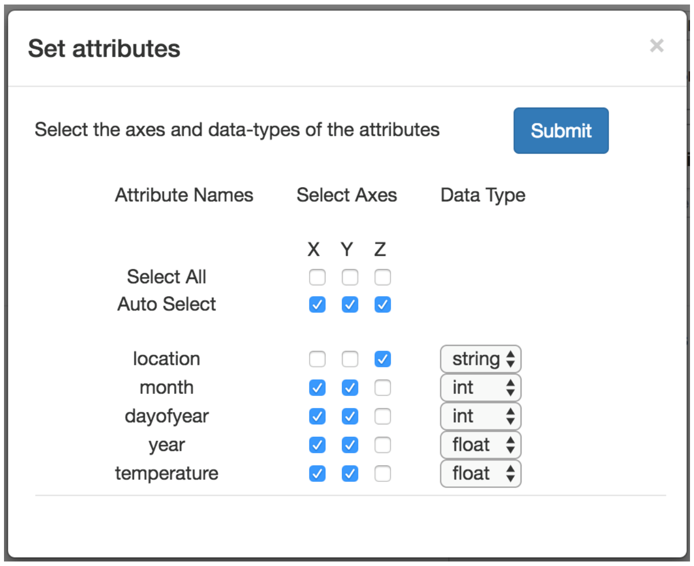
In an earlier version of Zenvisage, a bottleneck caused by context switching occurred when users uploaded their datasets along with a data type schema file. The burden of maintaining a separate schema file was onerous from the user's perspective. To this end, we developed a dataset upload feature that automatically generates the schema to support a smooth transition between systems. With automatic dataset upload, users can click on the plus icon in Panel A of Figure 2.1 to upload a dataset. A dialog shown in Figure 2.2 will pop up to allow users to select the dataset file.


Upload your own dataset

Instructions on how to format and upload your own datasets.

Dataset name: weather

Figure 2.2: Dataset upload modal [12]


Set attributes

Select the axes and data-types of the attributes

Attribute Names	X	Y	Z	Data Type
Select All	☐	☐	☐	
Auto Select	✓	✓	✓	
location	☐	☐	✓	string
month	✓	✓	☐	int
dayofyear	✓	✓	☐	int
year	✓	✓	☐	float
temperature	✓	✓	☐	float

Figure 2.3: Automatic dataset schema detection [12]

The user can also enter a custom name for the dataset. Once the dataset is uploaded, Zenvisage will automatically extract data types and axis channels for each attribute. This process uses multiple heuristics to determine the correct type and channel. The results of automatic detection are displayed in a new dialog that is shown in Figure 2.3. The dialog includes checkboxes and dropdown menus, which are filled out based on detected values. This feature allows the data uploading process to scale with the number of attributes in the uploaded dataset because users do not have to manually select an option for each attribute. The users still have the option to change settings for each individual attribute. Nevertheless, we observed that users still faced difficulties due to a division in their workflow because the use of Zenvisage assumes that data processing and cleaning is complete. Users were required to clean the data and save it in a CSV format before uploading the dataset to Zenvisage for further inspection.

2.3.2 Sketch Queries

While developing Zenvisage, we put significant effort into supporting the top-down search of visualizations via visual queries. The two types of top-down search are invoked by sketch queries and scatter queries. A sketch query is used for searching for desired patterns in time series data, while scatter queries are suited for finding scatter plots with high data point density in specified areas. For both types of search, Zenvisage receives input from a sketch board where users can draw visual queries. For a sketch based query, users draw a time series trend line that is compared to candidate visualizations based on similarity. Each candidate is then ranked by its similarity score and displayed to the user. A concrete example is illustrated in Figure 2.4.

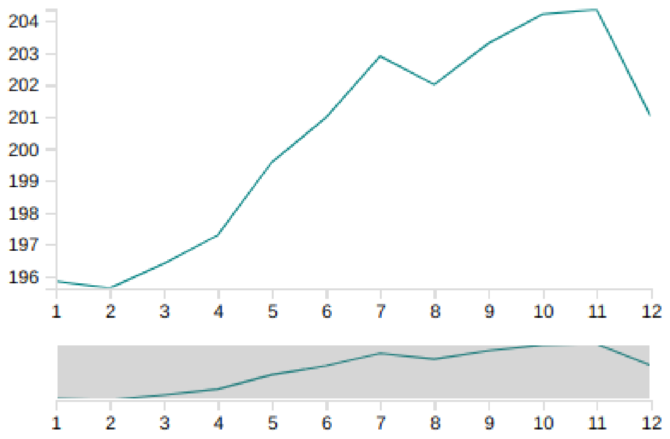


Figure 2.4: Time series sketch query for similarity search [12]

During our collaboration with material scientists, we learned that they prefer to work with scatter plots instead of line charts since they were interested in the behavior of individual solvents unaggregated. To support their analysis tasks, we developed a scatter querying functionality to search for interesting scatter plots. For a scatter query, the sketch board shows a scatter plot that contains the aggregation of points across all possible bivariate charts. The data points are then binned into hexagons, where a darker color signals a higher density of data points. The users can draw a polygon on the sketch board, and Zenvisage will rank candidate charts based on the scatter plots with the most points within the polygon in the sketch board. An example query is shown in Figure 2.5. While we expected these visual queries to assist data scientists with their search for interesting visualizations, such queries were seldom used due to the challenges of top-down queries alluded to earlier. In addition, we learned the importance of building flexible analytics modules because supporting only time series and scatter plot queries limited the number of use cases for Zenvisage.

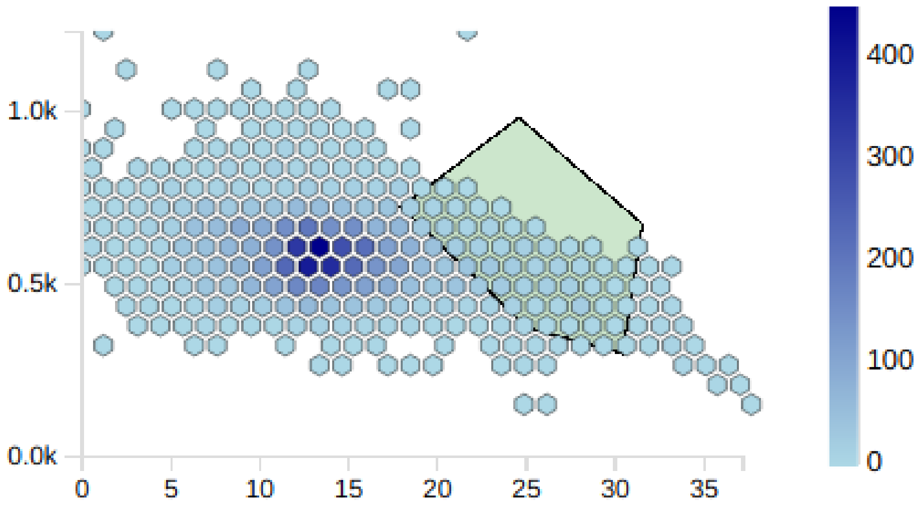


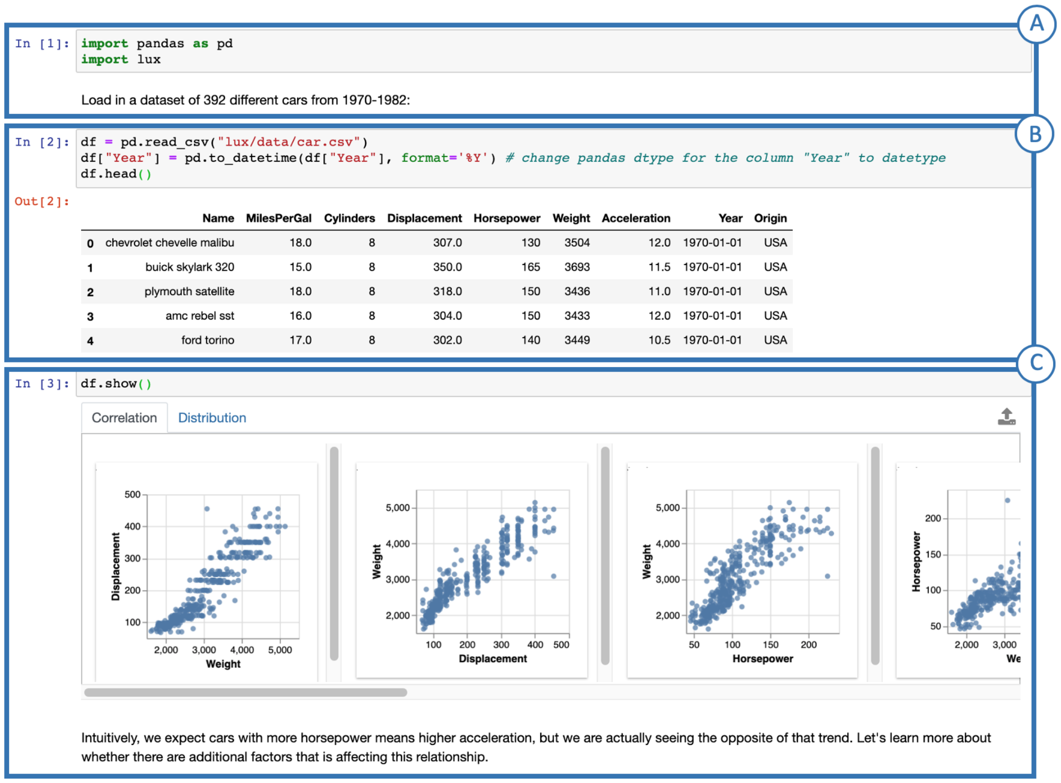
Figure 2.5: Scatter plot polygon query [12]

CHAPTER 3: LUX CAPABILITIES

In this chapter, we introduce the Lux interface, a Jupyter widget [16] that displays visual recommendations targeted at extending the capabilities described in the previous chapter and fixing the limitations thereof. Then, we describe features for generating visualization collections and custom analytics modules.

3.1 LUX WIDGET INTERFACE

Jupyter Notebooks, an open-source web application for creating documents that contain live code and visualizations, is our chosen medium for interacting with users. We leverage the ubiquity of Jupyter in the data science community to serve a large audience, as well as to provide a seamless interface with data scientists' existing workflows. To describe how the interface operates, we showcase a demonstration. Figure 3.1, 3.2 and 3.3 illustrates how data scientists can explore the Cars dataset using Lux within Jupyter. This comprehensive demonstration will be our reference for all Lux features moving forward.



```
In [1]: import pandas as pd
        import lux
```

Load in a dataset of 392 different cars from 1970-1982:

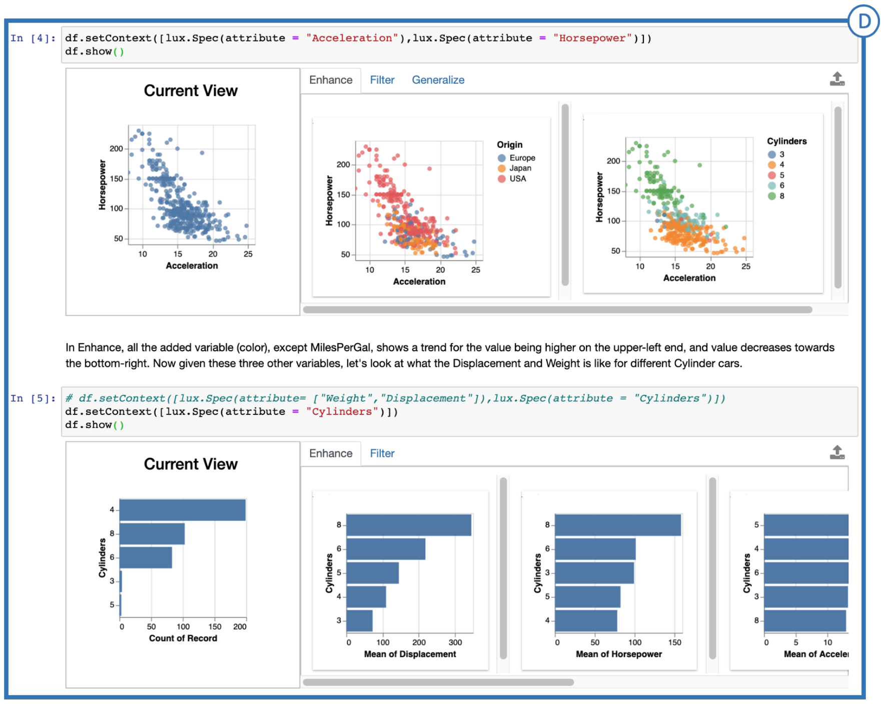
```
In [2]: df = pd.read_csv("lux/data/car.csv")
        df["Year"] = pd.to_datetime(df["Year"], format='%Y') # change pandas dtype for the column "Year" to datetype
        df.head()
Out[2]:
```

	Name	MilesPerGal	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
0	chevrolet chevelle malibu	18.0	8	307.0	130	3504	12.0	1970-01-01	USA
1	buick skylark 320	15.0	8	350.0	165	3693	11.5	1970-01-01	USA
2	plymouth satellite	18.0	8	318.0	150	3436	11.0	1970-01-01	USA
3	amc rebel sst	16.0	8	304.0	150	3433	12.0	1970-01-01	USA
4	ford torino	17.0	8	302.0	140	3449	10.5	1970-01-01	USA

```
In [3]: df.show()
```

Intuitively, we expect cars with more horsepower means higher acceleration, but we are actually seeing the opposite of that trend. Let's learn more about whether there are additional factors that is affecting this relationship.

Figure 3.1: Lux setup, data Ingestion, and data overview actions

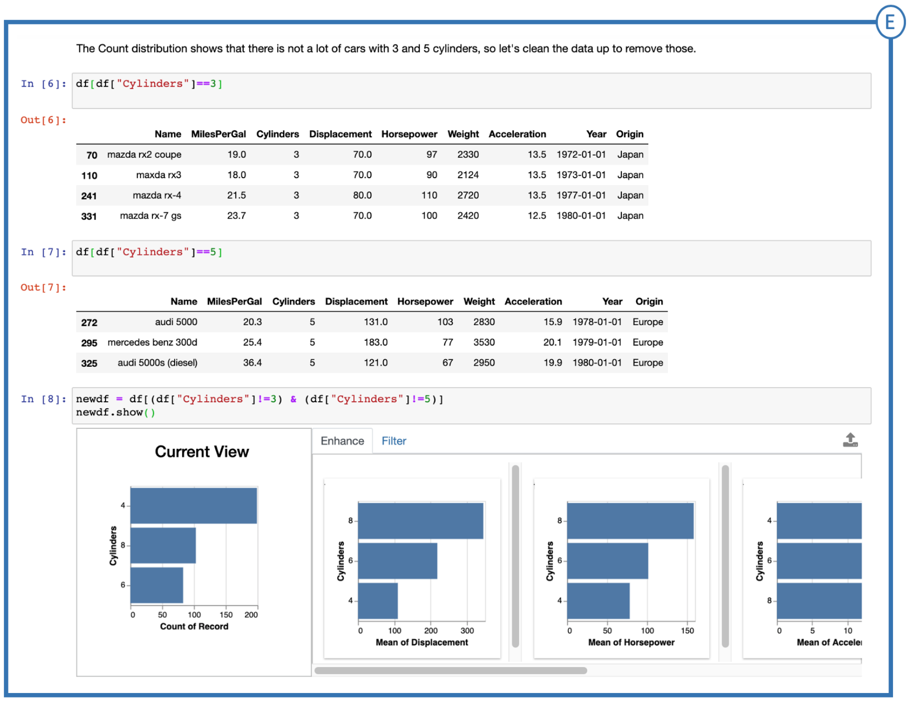


```
In [4]: df.setContext([lux.Spec(attribute = "Acceleration"),lux.Spec(attribute = "Horsepower")])
        df.show()
```

In Enhance, all the added variable (color), except MilesPerGal, shows a trend for the value being higher on the upper-left end, and value decreases towards the bottom-right. Now given these three other variables, let's look at what the Displacement and Weight is like for different Cylinder cars.

```
In [5]: # df.setContext([lux.Spec(attribute= ["Weight","Displacement"]),lux.Spec(attribute = "Cylinders")])
        df.setContext([lux.Spec(attribute = "Cylinders")])
        df.show()
```

Figure 3.2: Actions with specifications



The Count distribution shows that there is not a lot of cars with 3 and 5 cylinders, so let's clean the data up to remove those.

```
In [6]: df[df["Cylinders"]==3]
```

Out[6]:

	Name	MilesPerGal	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
70	mazda rx2 coupe	19.0	3	70.0	97	2330	13.5	1972-01-01	Japan
110	maxda rx3	18.0	3	70.0	90	2124	13.5	1973-01-01	Japan
241	mazda rx-4	21.5	3	80.0	110	2720	13.5	1977-01-01	Japan
331	mazda rx-7 gs	23.7	3	70.0	100	2420	12.5	1980-01-01	Japan

```
In [7]: df[df["Cylinders"]==5]
```

Out[7]:

	Name	MilesPerGal	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
272	audi 5000	20.3	5	131.0	103	2830	15.9	1978-01-01	Europe
295	mercedes benz 300d	25.4	5	183.0	77	3530	20.1	1979-01-01	Europe
325	audi 5000s (diesel)	36.4	5	121.0	67	2950	19.9	1980-01-01	Europe

```
In [8]: newdf = df[(df["Cylinders"]!=3) & (df["Cylinders"]!=5)]
        newdf.show()
```

Figure 3.3: Visual recommendations after data cleaning

In Jupyter Notebooks, code cells allow data scientists to enter and run Python code. A typical workflow includes data ingestion, data cleaning, analysis, and visualization. An example of data ingestion is illustrated in panel B (code line 2, denoted as In[2]), where the data scientist loads the Cars dataset before starting analysis.

Sometimes, running commands in code cells generate data to standard output. The standard output streams are displayed in the output area such as panel B (code line 2, denoted as Out[2]). For these output panels, Jupyter also supports the Jupyter widget framework which allows developers to build displays that can be customized for rich representations of Python objects. For Lux, we take advantage of such custom widget displays to visualize recommendations in the output cells. Panel C (code line 3 denoted as In[3]) is an example of visual recommendations rendered in Jupyter using custom rendering logic.

To summarize, Lux receives user input in code cells, generates helpful visualizations in the background based on the input, and displays the visualizations as recommendations in the output cells.

Overall, Lux strives to be a one-stop solution for a diverse set of visual data exploration needs. To achieve this, Lux is deeply integrated into the common tasks of data scientists in Jupyter. Analysts can start using Lux with a simple import statement to our library. Lux also exposes its API with language constructs that are simple and unobtrusive to the users existing code. Such seamless integration is made possible because Lux is built as an extension to Pandas [17], a popular Python library that enables users to work with relational-like data through a data structure called *dataframes*. Pandas dataframes are widely adopted by data scientists for data loading, wrangling, cleaning, and analysis. Lux extends the notion of a dataframe by augmenting them with visualizations based on the current state of the data and optional inputs. Specifically, if the user is unclear about what to visualize, Lux visualizes all univariate and bivariate visualizations from the data to give an overview of the dataset. However, if the user chooses to dig deeper into a specific set of visualizations, the augmented dataframes from Lux can receive specification inputs to generate recommendations that target the user's interest. When rendered as visualizations, these recommendations are ranked by metrics that capture their likelihood to be potentially interesting to the user. Armed with the knowledge of how the Lux interface works at a high level, we explain each step that the analyst takes in our Cars demonstration.

- Figure 3.1 A: The user imports Lux to augment Pandas dataframes with automatic visualization capabilities. After the import, all dataframes include Lux capabilities to generate visual recommendations.

- Figure 3.1 B: The user reads data from a CSV file into a dataframe object. The initial lines of the dataframe are printed to validate that the data read is successful.

- Figure 3.1 C: The user is not ready to dive into detailed analysis yet. For now, the user simply calls the "show" function to request visualizations that give an overview of the dataset. The outputs are visualized into two tabs that represent different statistics.
- Figure 3.2 D: Based on the information mined from visualizations in step C, the analyst decides on a specific area of interest for further exploration. This area of interest is expressed as input specifications to the "setContext" call that sets context for what the user may be interested in. After running "setContext", the output of "show" differs from that of step C. In contrast to step C, the output has a *current view* which represents the user's specification input. The generated visualizations also differ because the recommendations are now based on the current view.
- Figure 3.2 E: User repeats step D, but also performs data inspection and data cleaning with Pandas operations before visualization. The procedures highlight that all existing Pandas related operations are unaltered by Lux and proceed as before. Furthermore, we observe that Pandas and Lux work together to provide continuous visualization capabilities.

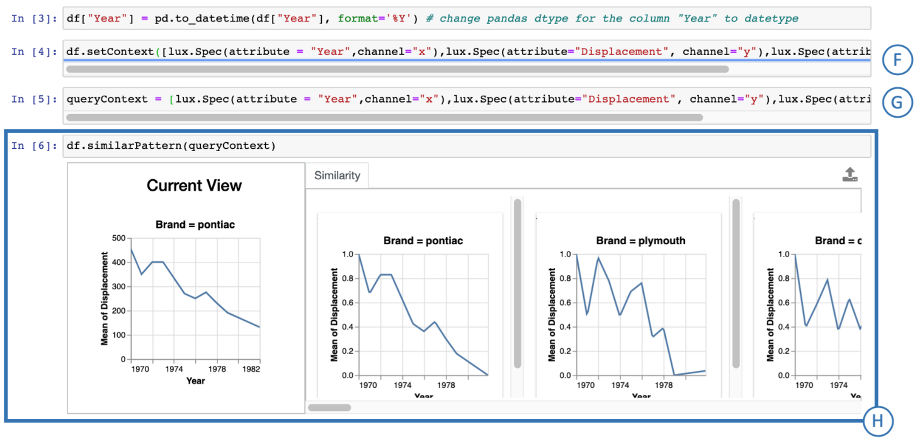
3.2 SEARCH SPACE ENUMERATION

Similar to existing mixed-initiative interfaces for visualization recommendations [18] [6] [19], Lux allows users to specify the visualization sets that they intend to explore. The freedom to generate an arbitrary group of visualizations is useful across various scenarios. In a simple use case, data scientists can browse through visualization collections to find interesting relationships within the data. For example, an analyst who wants to explore the relationships of the "Miles per Gallon" attribute with respect to other attributes can use the command below to generate a collection of scatter plots that meet the specifications.

```
ldf.setContext(lux.Spec(attribute = "MilesPerGallon"),lux.Spec("?"))
```

Here, the question mark denotes a wild card attribute that signals the system to generate a visualization collection with all possible attributes that can replace the wildcard.

In an advanced use case, users can specify desired patterns so that the system automatically traverses all candidate visualizations to find matching results. We showcase this application by re-implementing the similarity search functionality of Zenvisage in Lux. We can adapt our Cars dataset demonstration to run a similarity search for car brands that match the trend line of “Displacement” for the brand “Pontiac”.



```
In [3]: df["Year"] = pd.to_datetime(df["Year"], format='%Y') # change pandas dtype for the column "Year" to datetype
In [4]: df.setContext([lux.Spec(attribute = "Year",channel="x"),lux.Spec(attribute="Displacement", channel="y"),lux.Spec(attrib
In [5]: queryContext = [lux.Spec(attribute = "Year",channel="x"),lux.Spec(attribute="Displacement", channel="y"),lux.Spec(attri
In [6]: df.similarPattern(queryContext)
```

Figure 3.4: Similarity search functionality originally implemented in Zenvisage. The example search identifies car brands with similar displacement trends to Pontiac.

In Figure 3.4 we outline the sequence of commands required to run the similarity search.

- Figure 3.4 F: The dataframe receives input specifications to define a search space.
- Figure 3.4 G: The dataframe calls a similarity search with a query object. The query object represents a single visualization that contains a pattern of interest.
- Figure 3.4 H: The result set ranks visualizations by their similarity to the query visualization.

Similarity search shows how visualization enumeration is useful for sophisticated analytics. Also, the feature highlights Lux as a versatile tool that can be customized to support a diverse set of tasks.

3.3 AUTOMATIC ENCODING

Lux is built on the foundation that visual exploration should be accessible, even for those who do not have expertise in data visualization. In order to support this, Lux users can visualize and explore anything they specify without having to worry about how the visualizations should look like. Similar to existing encoding-based visualization recommendation systems [10], Lux uses a set of heuristics to automatically determine the appropriate visual encoding for a given visualization. For example, it takes only a few lines of code in Lux to plot a bar chart of “Average Horsepower” by “Origin”. This is because Lux automatically determines that a bar chart should be visualized given that “Horsepower” is a quantitative variable and “Origin” is a categorical variable. In contrast, other visualization libraries such as Plotly [20] and matplotlib [2] require additional specification details to plot the same charts. While BI tools like Tableau [3] often support similar features, the level of automation Lux brings to users does not exist in other visualization libraries.

3.4 LUX QUERY LANGUAGE SYNTAX

As seen in our demonstration, data scientists steer their exploration by interacting with the dataframe *context*, i.e., a specification of the attribute or values that they are interested in. Lux supports a simple syntax for specifying the context so that users can explore all parts of the dataset. A basic usage of Lux syntax is specifying the context with one specification. Specifying only “Horsepower” means that the user is interested in one attribute, and the visual recommendations will be generated based on “Horsepower” alone.

```
ldf.setContext(["Horsepower"])
```

In the previous example, specifying the attribute name (“Horesepower”) was sufficient because Lux automatically populates underspecified arguments for generating visualizations. In contrast, users can provide explicit specifications by providing Lux Spec objects as arguments for the context. The next example shows the user forcing the “MilesPerGal” attribute to be the x axis of the output visualizations by providing a Spec object.

```
ldf.setContext(["Horsepower", lux.Spec("MilesPerGal",channel="x")])
```

For categorical attributes, users can apply a filter to the output visualizations by specifying a value for the attribute. To apply the filter, the user inputs a string which is the attribute name, followed by an equal sign and the filter value.

```
ldf.setContext(["Horsepower", "Origin=USA"])
```

Lux allows multiple attributes and values to be specified at once so that users can express interest in visualizations with more than one attribute or filter. For example, the bar sign denotes a logical OR which can be used to specify multiple attributes or filters.

```
ldf.setContext(["Horsepower", "Origin=USA|Japan"])
```

```
ldf.setContext(["Horsepower|MilesPerGal|Weight", "Origin=USA"])
```

For those who are more familiar with Python syntax, multiple attributes can also be specified as a list of strings.

```
ldf.setContext([["Horsepower", "MilesPerGal", "Weight"], "Origin=USA"])
```

Finally, since it may be difficult for users to specify all possible attributes or values at once, we support question marks as wild card queries. The wildcard signals Lux to automatically populate the context with all possible unique attributes or filter values.

```
ldf.setContext(["Horsepower", "Origin=?"])
```

```
ldf.setContext([lux.Spec("?")])
```

3.5 ANALYTICS ACTION MODULES

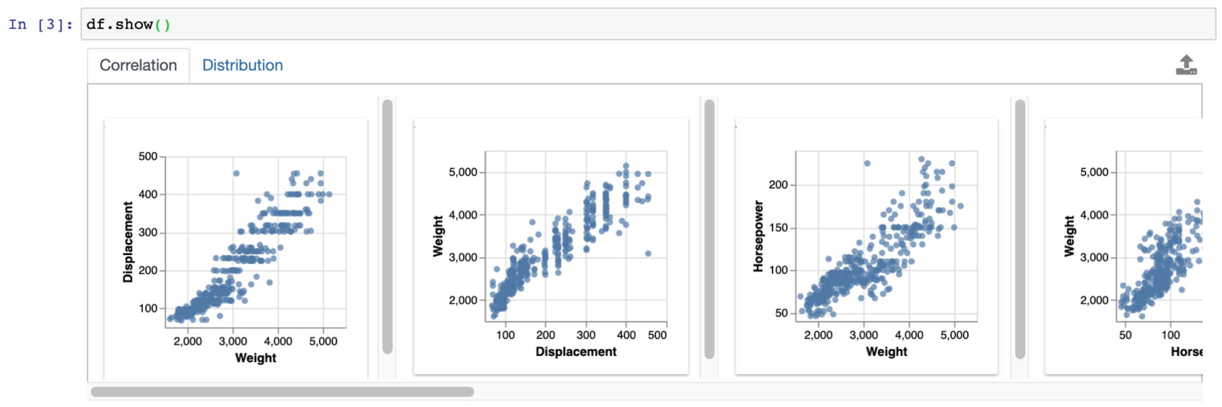
Working off of a system that allows users to specify a group visualizations, we can perform analytics on top of the selected displays. We define such operations on visualizations as *actions*, where each action represents different analytical intents or user goals. The goal of organizing analytics into discrete user actions is to build a system that can flexibly support a plethora of tasks.

Lux currently supports a set of most popular operations based on existing research in the visualization recommendation literature. The action classes include Correlation, Distribution, Enhance, Filter, and Generalize. Next, we describe each action class with supporting examples.

3.5.1 Actions without Input Specifications

We depicted in the Cars demonstration that the easiest way to learn about a dataset is by initializing a dataframe and calling the “show” function. This simple two-step process is significant for a few reasons. First and foremost, data scientists who want an overview of the dataset can use Lux before closer inspection via subsequent analysis. Second, it allows beginner users to interact with Lux without the overhead of learning new syntax. When a user does not provide any specification, we generate all visual recommendations under action classes that can be computed without any user input. Such actions are intended to provide useful information despite minimal effort from the user. Currently, we have implemented Correlation and Distribution for these types of actions.

3.5.2 Correlation

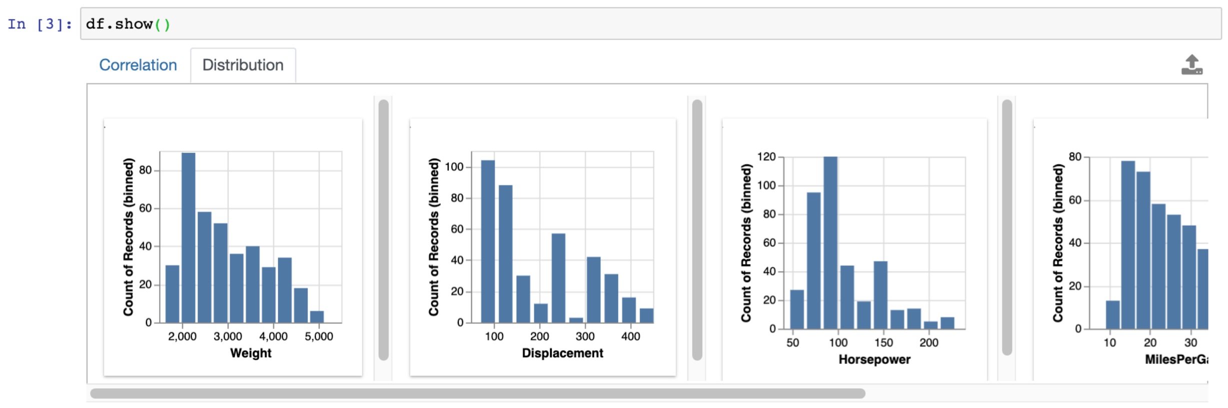


```
In [3]: df.show()
```

Figure 3.5: Example of Correlation action

Displayed in Figure 3.5, the Correlation action scores visualizations based on how correlated two quantitative variables are. When the action is invoked, Lux generates bivariate visualizations that represent all pairwise relationships in the dataset. Then, each visualization is scored by the Spearman's correlation to show the visualizations in ranked order. This action serves as a launchpad for data scientists trying to understand the relationships between attributes in their dataset.

3.5.3 Distribution



```
In [3]: df.show()
```

Correlation
Distribution
Count of Records (binned)
Weight
Count of Records (binned)
Displacement
Count of Records (binned)
Horsepower
Count of Records (binned)
MilesPerGa

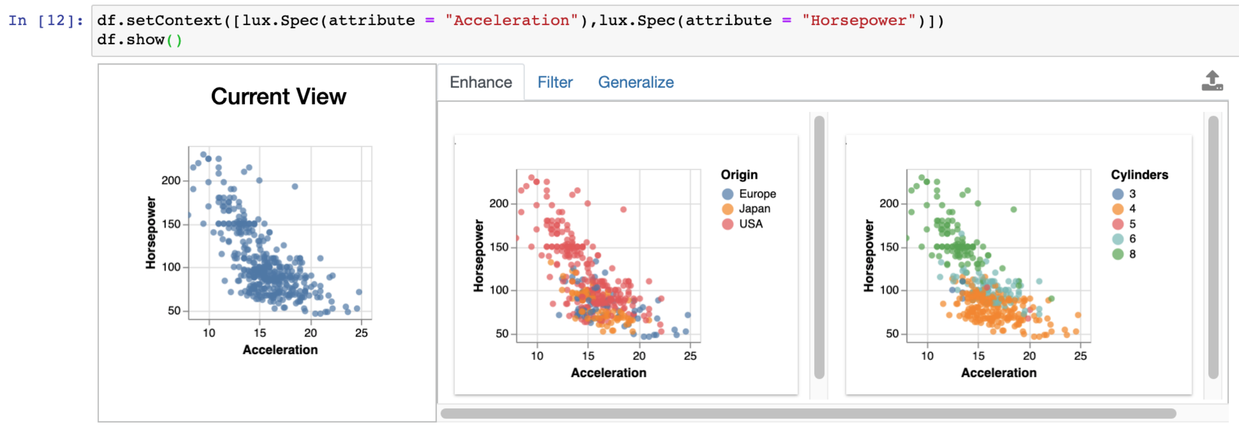
Figure 3.6: Example of Distribution action

Another aspect of the dataset that Lux can visualize is the data distribution. Depicted in Figure 3.6, the Distribution action provides an overview of the data by generating univariate count distributions of different attributes in the dataset. The distributions are ranked by a skewness coefficient to identify histograms that deviate from a normal distribution. In general, skewed distributions will have more weight in the left or right tail of the distribution. The Distribution action is useful when little is known about the dataset because it finds visualizations with interesting distributions.

3.5.4 Actions for data navigation

Visualizing relevant data is essential when exploring for insights. Unfortunately, data scientists often overlook parts of the data that may hold valuable information when the number of attributes in the dataset increases. Figuring out which exact attributes and values to visualize is difficult when certain attributes are left out, or when visualizations contain too much noise to hinder the viewers' focus. Hence, Lux implements actions that allow users to navigate through the data by dynamically adjusting their context.

3.5.5 Enhance

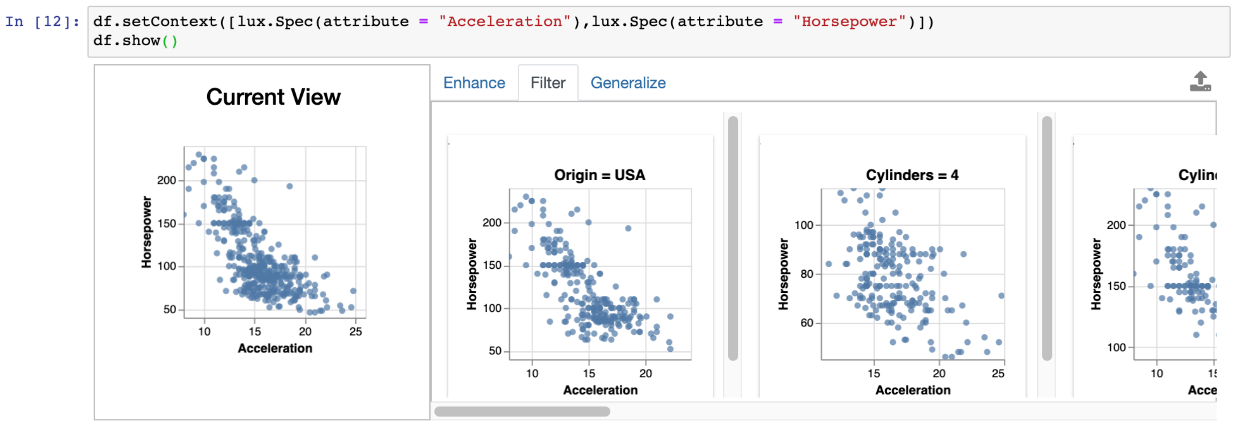


```
In [12]: df.setContext([lux.Spec(attribute = "Acceleration"),lux.Spec(attribute = "Horsepower")])
         df.show()
```

Figure 3.7: Example of Enhance action

Sometimes, including additional attributes in a visualization can show hidden information. Lux can reveal such details with the Enhance action, which takes in a set of visualizations and generates all possible views that include one additional attribute. Data scientists can use the Enhance action when they want to see the effects of an extra variable in their visualizations. For example, in Figure 3.7, the Enhance action shows the relationship between "Horsepower"and "Acceleration" broken down by either "Origin" or by "Cylinders".

3.5.6 Filter

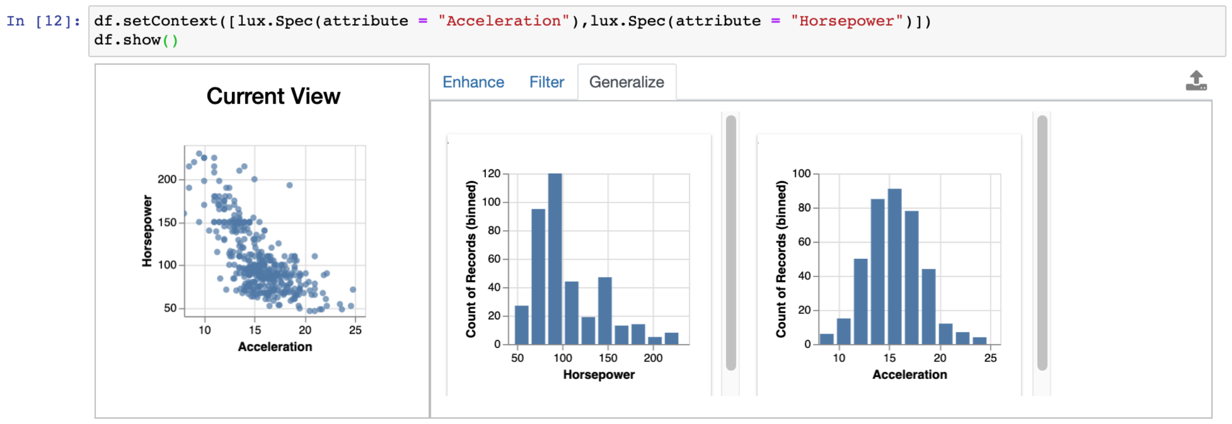


```
In [12]: df.setContext([lux.Spec(attribute = "Acceleration"),lux.Spec(attribute = "Horsepower")])
         df.show()
```

Figure 3.8: Example of Filter action

The Filter action iterates over all possible values of a categorical variable and generates visualizations where each categorical value filters the data in various ways. The goal is to visualize different data subsets that exist in the dataset. Example visualizations with applied filters are illustrated in Figure 3.8.

3.5.7 Generalize



```
In [12]: df.setContext([lux.Spec(attribute = "Acceleration"),lux.Spec(attribute = "Horsepower")])
         df.show()
```

Figure 3.9: Example of Generalize action

The opposite of an Enhance action is the Generalize action. In the Generalize action, we remove an attribute to observe a general trend. The Generalize action is helpful when users want to step back and see an overall trend of their data.

CHAPTER 4: LUX DESIGN AND IMPLEMENTATION

Up to this section, we described all user-facing aspects of Lux. Next, we discuss how Lux works under the hood by explaining important design choices and implementation details for building our system.

4.1 LUX ARCHITECTURE

The Lux architecture is composed of modules that have distinct responsibilities. The architecture can be described in layers: the user interface layer, the user input validation and parsing layer, the query processing layer, the data execution layer, and finally the analytics layer. From a software engineering standpoint, the principle behind our design is to take advantage of the extensibility of loosely coupled modules. We hope the modular architecture supports easy integration of new optimizations to any part of the existing code.

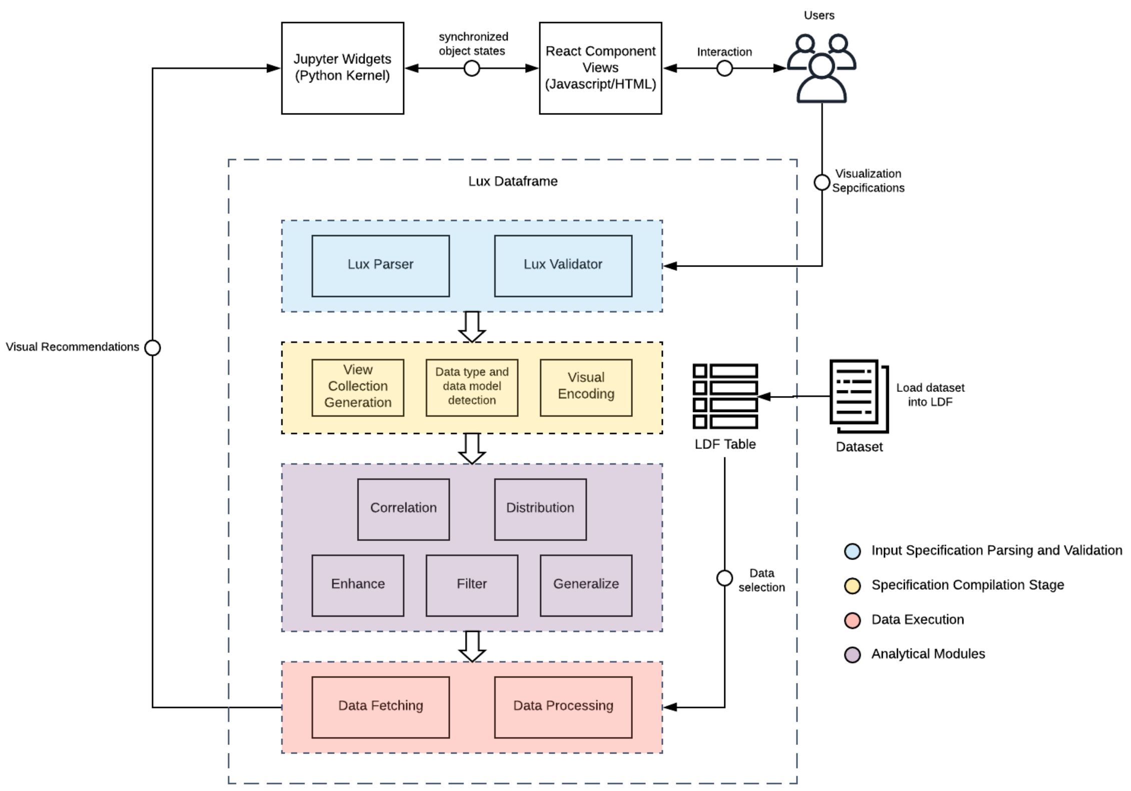

Jupyter Widgets (Python Kernel)
synchronized object states
React Component Views (Javascript/HTML)
Interaction
Users
Visualization Sepcifications
Lux Dataframe
Lux Parser
Lux Validator
Visual Recommendations
View Collection Generation
Data type and data model detection
Visual Encoding
LDF Table
Load dataset into LDF
Dataset
Correlation
Distribution
Enhance
Filter
Generalize
Data selection
Data Fetching
Data Processing
Input Specification Parsing and Validation
Specification Compilation Stage
Data Execution
Analytical Modules

Figure 4.1: Architecture diagram that shows interactions between different components in the Lux ecosystem.

4.1.1 User Interface Layer

As explained earlier, we output visualizations to Jupyter via custom widgets. These widgets act as a framework for creating custom HTML representations of Python objects within Jupyter. The benefit of displaying the output through widgets is that we can make visualizations interactive, a crucial component in building VRSs. Developing these widgets requires two steps.

The first step is to set up a communication layer between widget objects in the Python kernel (backend) and the Javascript objects represented in the browser (frontend). Establishing this connection allows the synchronization of visualization information between the Lux backend and the Jupyter frontend. Data about visualizations can now flow from Lux to Jupyter because the Python backend generates visualizations that are available immediately to Jupyter via the widgets. Information can also transmit in the opposite direction because the Python objects update in real-time based on the changes made to their Javascript counterparts [16].

The second step is extending the Jupyter interface so that widgets can display visualizations and receive user input. Precisely, we must add HTML elements and Javascript logic that can take care of displaying interactive visualizations. The React framework [21], a well-suited tool for building interactive user interfaces, was chosen for implementing these frontend elements. The advantage of React is its component-based structure. In React, different frontend elements are organized into encapsulated components that manage their own state. Based on changes in the state, a component responds by re-rendering to keep the interface up to date. Therefore, React's component based state management allows Lux visualizations to re-render whenever recommendations should be updated based on changes in the context.

4.1.2 Lux Data Structures

We made progress in developing the Lux backend over two iterations of system design and implementation. The first design was a proof-of-principle prototype, where we validated our hypothesis that the system can assist user tasks with Lux capabilities. Our current design is a functional prototype that supports an accessible API and extensible analytics modules. Over the two development cycles, we designed data structures that are necessary abstractions for each stage in the system pipeline. We introduce these essential building blocks to provide background information before going over the rest of the system.

4.1.3 The Lux Dataframe

Dataframes have become incredibly popular data models that are well suited for analytics. Its success can be attributed to multiple features including induced schemas, label-based indexing, and matrix-like operations. To benefit from the convenience of Pandas dataframes, Lux is designed with a focus on a tight integration with Pandas. To this end, we follow the instructions from the Pandas documentation [8] for extending Pandas data structures, which recommends subclassing. We define the central piece to Lux's data model as the Lux Dataframe (LDF), a subclassed Pandas dataframe that supports all dataframe operations while housing other variables and functions for generating visual recommendations.

There are strong motivations for choosing this design. First, we can preserve Pandas dataframe functionalities with subclassing, meaning all code written using Pandas dataframes will also work with Lux. Second, Lux syntax becomes elegant because all API endpoints can be directly called from the dataframe instead of an external Lux object.

4.1.4 Spec/Context

The Spec object represents a single unit of user specification. These specifications can be attributes that designate columns or filter values that specify rows in the dataset. The LDF stores these objects in a list named the Context, which holds all current specifications for generating recommendations. An essential job of the LDF is to maintain the Specs within the Context, so that generated visual recommendations are up to date with the user's input.

4.1.5 View/ViewCollection

Since Lux maintains sets of visualizations, we require a data structure that encapsulates each visualization and its properties so that we can score, rank and display them later. Hence, we define a View object for each visualization as a representation of all information required for data fetching and rendering. The LDF stores multiple Views in a View Collection, which is a list that represents a set of visualizations to display to the user. Since data fetching for a View is an expensive operation in VRSs [7], we designed the View to be decoupled from the data, so that the View can be easily modified or transferred during query processing stages.

4.1.6 System overview

Based on established definitions of the data structures used in Lux, we overview the system with a focus on each module. At a high level, the following sections describe the life cycle of how Lux interprets the user's analytical intent, fetches the relevant data, and finally performs analytics to generate visualizations.

4.1.7 The Lux Parser and Validator: Standardizing input specification

Before any processing happens, Lux interprets user inputs to transform strings into Spec objects for the Context. All syntax rules are applied to parse user input in this stage. In addition, input validation catches inconsistencies between the Specs and the dataset. With this feature, data scientists can discover mistakes early on in their exploration and make corrections. For example, if the input is a filter specification where the attribute "Origin" is equal to "USA", the validation stage checks whether the value "USA" exists for the attribute "Origin" in the dataset.

4.1.8 The Lux Compiler: Creating a full specification

Lux allows users to provide the bare minimum in terms of input specifications. Therefore, Spec objects often require additional processing before they are used for creating Views. Underspecified information for Specs within the Context are inferred during the compilation stage. The transformation of these Specs into Views is a three-step process.

1. **View Collection generation.** The system generates list of Views for visualization. These Views are created from Specs in the Context that are fully or partially specified. In the fully defined case, there is no ambiguity in which attributes the user wants to visualize. For partially specified instances, the system locates any Spec objects that include wildcard characters that are denoted by a question mark. These wildcard Specs are further processed to enumerate all candidate Views that hold explicit Specs. Ultimately, Lux creates a list of Views that correspond to each visualization that will be displayed in the frontend.

2. **Infer data type and data model information.** The system auto-fills missing details for each View. Each View holds Specs that correspond to the attributes for a visualization. For each of the attributes, we populate the Specs with corresponding data type information. These bits of information are necessary for encoding data into the correct visual elements.

3. **Visual encoding.** The final step in the compilation is an automatic encoding process that determines visualization mappings. The system automatically infers type, marks,

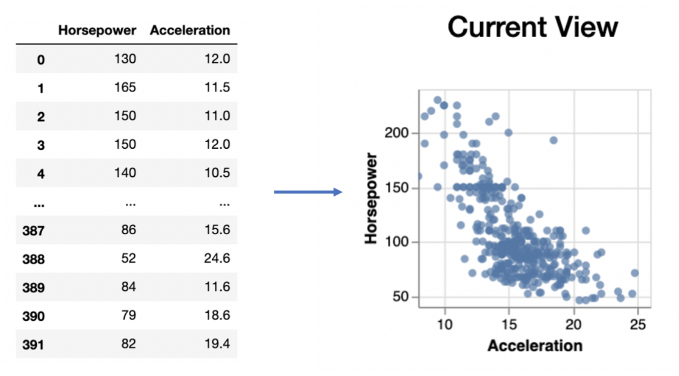
channels and additional details that can be left underspecified in the input specifications. We have implemented a set of visualization encoding rules that automatically determines marks and channels of each visualization based on data properties determined in step 2, as shown in table 4.1. This mechanism of automatic presentation has already seen success in commercial BI software, such as Tableau [3]. Therefore, encoding rules in Lux draw inspiration from the Show Me rules introduced by Mackinlay [10].

Number of Dimensions	Number of Measures	Mark Type
0	1	Histogram
1 (ordinal)	0,1	Line chart
1 (categorical)	0,1	Bar chart
2 (ordinal)	0,1	Line chart
2 (categorical)	0,1	Line chart
0	2	Scatter plot
1	2	Scatter plot
0	3	Scatter plot

Table 4.1: Encoding rules for automatically inferring visualization mappings.

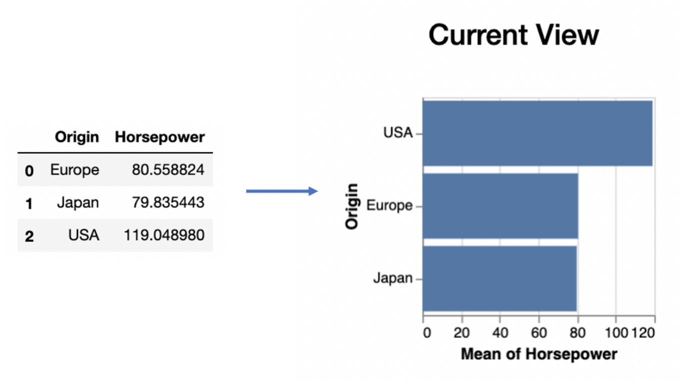
4.1.9 The Lux Execution Engine: Data fetching and processing

With all View properties defined, the final missing piece for visualization is the data. The data executor populates each View with a subset of the dataframe based on View specifications. The data executor invokes dataframe operations supported by the Pandas library to execute the selection and filter operations. After data fetching, each View data may require additional processing based on its chart type. For the scatter plot case, selection is sufficient for visualization. If the View is a bar or line chart, Lux performs data aggregation on an axis. Finally, if the View is a histogram, Lux computes appropriate bins and counts of data points. To show output of the data execution engine, we provide example visualizations and their underlying data in Figures 4.2, 4.3, and 4.4.



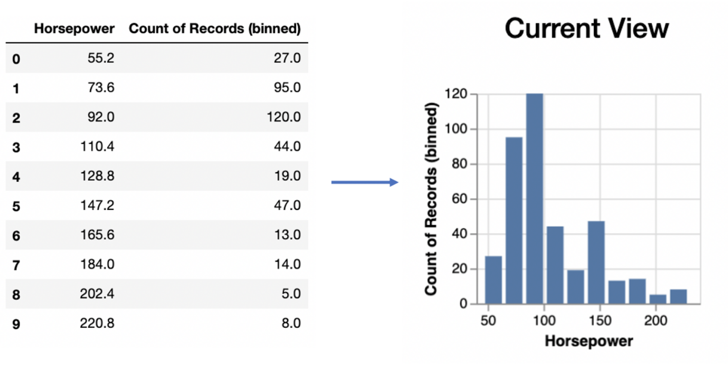
	Horsepower	Acceleration
0	130	12.0
1	165	11.5
2	150	11.0
3	150	12.0
4	140	10.5
...	...	...
387	86	15.6
388	52	24.6
389	84	11.6
390	79	18.6
391	82	19.4

Figure 4.2: Scatter plot data generated by dataframe selection operations for the context query *[lux.Spec(attribute = "Acceleration"),lux.Spec(attribute = "Horsepower")]*



	Origin	Horsepower
0	Europe	80.558824
1	Japan	79.835443
2	USA	119.048980

Figure 4.3: Bar chart data generated by dataframe selection and aggregation operations for the context query *[lux.Spec(attribute = "Horsepower"),lux.Spec(attribute = "Origin")]*



	Horsepower	Count of Records (binned)
0	55.2	27.0
1	73.6	95.0
2	92.0	120.0
3	110.4	44.0
4	128.8	19.0
5	147.2	47.0
6	165.6	13.0
7	184.0	14.0
8	202.4	5.0
9	220.8	8.0

Figure 4.4: Histogram data generated by dataframe selection and binning operations for the context query *[lux.Spec(attribute = "Horsepower")]*

CHAPTER 5: FUTURE WORK

Building our most recent prototype allowed us to establish the foundations of the VRS we envisioned. While many of our goals were brought to fruition, multiple options exist for improving our current system to reach a level of maturity that is on par with the expectations of data scientists. Here, we share the paths in our future roadmap that we believe can take Lux to the next level.

5.1 USER INTERACTIONS

A critical feature of a successful VRS is an interactive interface. While users should be able to access all Lux capabilities programmatically, unlocking interactive queries speeds up and simplifies the user's experience. It may also flatten the learning curve for users who are not proficient with Lux syntax. For instance, Zenvisage had a dedicated user interface for interactive queries that provided an intuitive user experience. Based on our success from Zenvisage, we plan to build interactive components for Lux.

Currently, the user interface of Lux only supports programmatic input and output display of visualizations. However, Lux was designed with an expectation from day one to support an interactive user interface in the future. For example, it was described that Jupyter widgets allow two-way communication between objects in the Python kernel and Javascript frontend. Using this capability, we can build widgets that update states of objects in the backend according to changes made to the Javascript in the frontend.

Another consideration is deciding on which types of interactive queries to support. We can start experimenting with existing query types from Zenvisage, such as drag-and-drop queries or sketch queries. Another approach is to design new kinds of queries based on the user's needs. In either case, building an effective interactive interface will require thorough research and design.

5.2 MODULE EXTENSION CAPABILITIES

Throughout this thesis, we emphasized Lux to be an extensible tool that is flexible in terms of supporting different types of analytics. We believe the process of adding new analytical modules to Lux should be extensible and accessible. We consider two initiatives for improving Lux extensibility. First, a dedicated documentation for users who aim to customize Lux should exist to onboard beginners. This documentation is vital to encourage and streamline the process of extending Lux. The second initiative is to build Lux as a framework that allows users to add new analytics capabilities easily. We believe that providing a systematic way of creating custom features enables Lux to scale with the number of different user workflows.

5.3 SCALABILITY

For Lux to generate insightful visualizations, it must handle large amounts of data. Uncharted territory for Lux is a definitive plan on how it will scale to bigger datasets. The challenge to consider is that the interactive nature of the interface must be preserved while scaling. We have discussed possible solutions such as finding optimizations in our data execution model or using external libraries like Modin [22] to speed up our dataframe operations. However, it is not clear which method is best suited for scaling up.

5.4 USER STUDIES

The ultimate goal of Lux is to become integral to the success of data scientists. After the Lux project reaches a certain level of maturity, we believe it is necessary to conduct a user study targeting the data science community. Gathering usage data and user feedback will allow us to fine-tune the tool and plan our next steps.

CHAPTER 6: CONCLUSION

In recent years, the impact of data analytics tools have increased in tandem with the rise of big data. Among these tools, data scientists have favored Jupyter and Pandas dataframes for a variety of tasks such as data loading, wrangling, cleaning, and analysis. However, we believe that these tools can offer more with additional capabilities for data exploration. This thesis outlines our work in integrating adaptive visualizations into Jupyter and Pandas to assist data exploration.

Developing effective data exploration systems requires a deep understanding of challenges that users face. To this end, we studied Zenvisage to identify critical user challenges. Based on the lessons learned, we designed Lux, a novel system that augments Pandas dataframes via in-situ visualizations. New features in Lux such as interactive widget displays, dataframe integration, and flexible analytics modules highlight our attempts to address issues in traditional systems. Hence, we believe that Lux brings the data science community one step closer to unlocking powerful insights with the help of automatic visual recommendations.

REFERENCES

[1] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York, 2016. [Online]. Available: http://ggplot2.org

[2] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, p. 90, 2007.

[3] "Tableau," Mar. 2019. [Online]. Available: https://www.tableau.com

[4] C. Ahlberg, "Spotfire: an information exploration environment," *ACM SIGMOD Record*, vol. 25, no. 4, pp. 25–29, 1996.

[5] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, "Effortless data exploration with zenvisage: an expressive and interactive visual analytics system," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 457–468, 2016.

[6] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager: Exploratory analysis via faceted browsing of visualization recommendations," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 649–658, 2016.

[7] M. Vartak, S. Madden, A. Parameswaran, and N. Polyzotis, "Seedb: supporting visual analytics with data-driven recommendations," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, p. 2015, 2015.

[8] "Pandas api reference," Mar. 2020. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/index.html

[9] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: a system for query, analysis, and visualization of multidimensional databases," *Communications of the ACM*, vol. 51, no. 11, pp. 75–84, 2008.

[10] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, pp. 1137–1144, 2007.

[11] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer, "Profiler: Integrated statistical analysis and visualization for data quality assessment," in *Proceedings of the International Working Conference on Advanced Visual Interfaces.* ACM, 2012, pp. 547–554.

[12] R. Wang, "Enabling effective visual data exploration for solvent discovery in material science," M.S. thesis, University of Illinois at Urbana Champaign, Urbana, 2019.

[13] D. J.-L. Lee, J. Lee, T. Siddiqui, J. Kim, K. Karahalios, and A. Parameswaran, "You can't always sketch what you want: Understanding sensemaking in visual query systems." *arXiv preprint arXiv:1710.00763*, 2017.

[14] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[15] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International journal of computer vision*, vol. 40, no. 2, pp. 99–121, 2000.

[16] "Ipython widgets reference," Mar. 2020. [Online]. Available: ipywidgets.readthedocs.io/

[17] W. McKinney, "Data structures for statistical computing in python," *Proceedings of the 9th Python in Science Conference*, 2010.

[18] T. Siddiqui, J. Lee, A. Kim, E. Xue, X. Yu, S. Zou, L. Guo, C. Liu, C. Wang, K. Karahalios et al., "Fast-forwarding to desired visualizations with zenvisage." in *CIDR*, 2017.

[19] C. H. Kevin Hu, Diana Orghian, "Dive: A mixed-initiative system supporting integrated data exploration workflows," *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2018.

[20] "Plotly documentation," Mar. 2020. [Online]. Available: https://dash.plotly.com/

[21] "React documentation," Mar. 2020. [Online]. Available: https://reactjs.org

[22] D. J.-L. Lee, J. Lee, T. Siddiqui, J. Kim, K. Karahalios, and A. Parameswaran, "Towards scalable dataframe systems," *arXiv preprint arXiv:1710.00763*, 2017.