ST_Visions — A Python library for Visualizing Spatio-temporal Datasets (Technical Report)

Andreas Tritsarolis
Data Science Lab. (datastories.org)
University of Piraeus
andrewt@unipi.gr

Christos Doulkeridis
Data Science Lab. (datastories.org)
University of Piraeus
cdoulk@unipi.gr

Yannis Theodoridis
Data Science Lab. (datastories.org)
University of Piraeus
ytheod@unipi.gr

Nikos Pelekis
Data Science Lab. (datastories.org)
University of Piraeus
npelekis@unipi.gr

Contents

List of Figures

Chapter 1

Introduction

Nowadays, we live in what could be regarded as the era of 'data renaissance'. Data flows from various sources in huge quantities. Thus, their analysis is evolving into a complex task due to their heterogenity, high complexity and massive volume. The most common (and representative) example is mobility data generated and collected by GPS sensors installed in vehicles, such as cars, vessels and aircrafts.

In order to be exploited in the decision-making process, the data needs to be properly processed for several data mining purposes, such as fishing pressure and traffic control in both air and land as well. Naturally, this process requires integrating human perception and visualization so as to determine the correct course of action in order to reach some meaningful conclusions.

In this technical report, we address the problem of data visualization, and present a utility called "*ST_Visions*" (**S**patio-**T**emporal **Vis**ualizat**ions**), able to interactively visualize spatio-temporal data in a quick-and-easy way. This utility is written entirely in Python 3 and (mainly) makes use of the Bokeh [1] library[1] for the interactive visualization part.

1.1 Overview

Figure 1.1 illustrates the block diagram of an *ST_Visions* instance. At a higher level, the block diagram is separated into five distinct phases, namely, "Loading Data", "Creating Canvas", "Drawing Geometry", (optional) "Filtering Geometry Data", and "Rendering Canvas".

The first phase, is about loading data from either, a (Geo)Pandas (Geo)DataFrame, or a CSV file, or a PostGIS database and allocating them to the class' respective attribute. Afterwards, the second phase, is concerned with the creation of the *bokeh.plotting.figure instance* (that will be referred to as the *Canvas*) as well as the *bokeh.models.ColumnDataSource* (that will be referred to as the *CDS*).

After creating the basic structures of our instance (Data, Canvas, and CDS), the third phase uses the latter two in order to (primarily) draw the dataset and (optionally) create a colormap, given either a categorical or a numerical attribute. At the fourth phase, the user may choose to add one (or more) filters in order to add another layer of interaction. Finally, regardless of whether a filter is created or not, the canvas is ready to be rendered, thus reaching the final phase of our instance.

1.2 Setup Instructions

In order to use *ST_Visions* in your project, download all necessary modules in your directory of choice via pip or conda, install the class' dependencies, as the following commands suggest:

```
# Using pip/virtualenv
pip install -r requirements.txt

# Using conda
conda install -r requirements.txt
```

[1] Overview and Installation instructions can be found at: bokeh.org

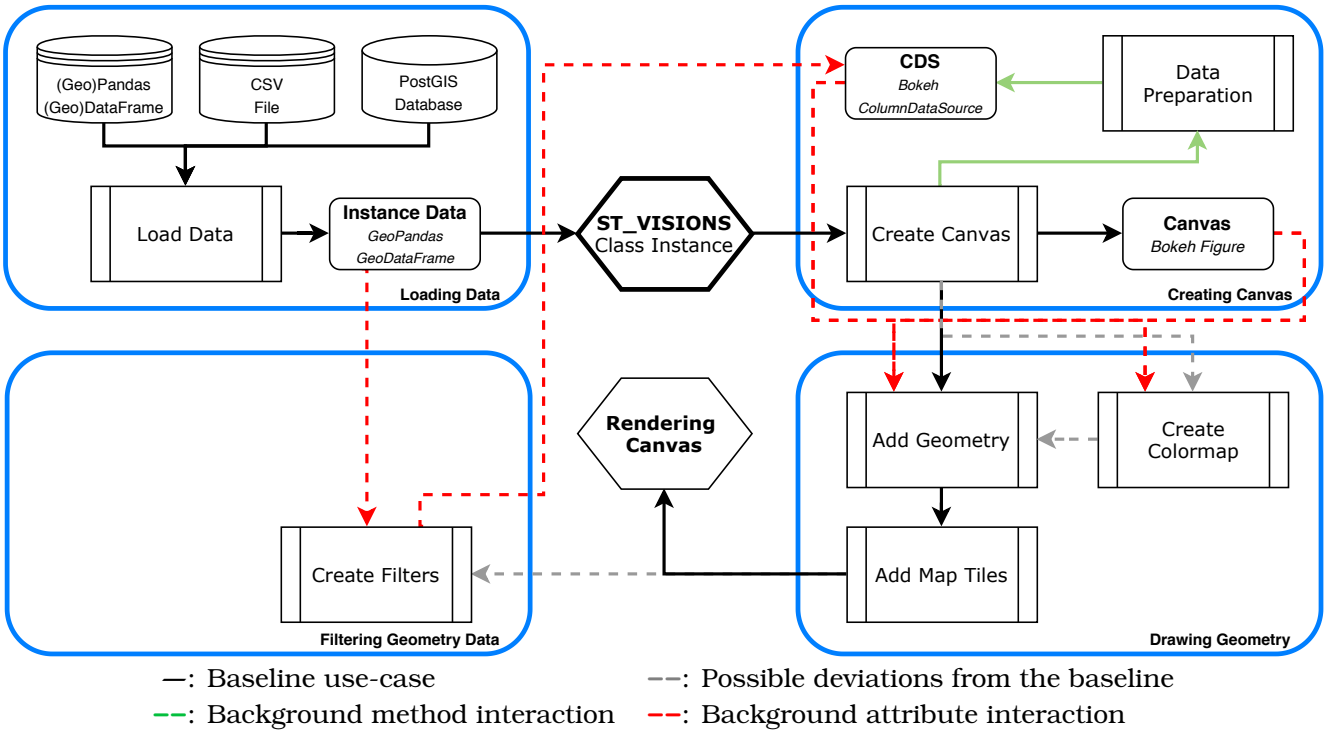


Figure 1.1: Block diagram of an *ST_Visions* instance.

Afterwards, in order to gain access to *ST_Visions*, you must append the class' directory to the script's scope and then import the class, as the following example suggests. The first two lines add the path of the class to the script's scope, while the 4$^{\text{th}}$ line gives access to *ST_Visions*, and the 5$^{\text{th}}$ provides access to an ensemble of baseline use-case methods for fast, yet highly interactive figures.

```
1  import sys
2  sys.path.append(<PATH_TO_ST_VISIONS_PARENT_DIRECTORY>)
3
4  import st_visions.st_visualizer as viz
5  import st_visions.express as viz_express
6  import st_visions.geom_helper as viz_helper
7  import st_visions.callbacks as viz_callbacks
```

The datasets that we will use for this demonstration lie in the field of mobility data, and more specifically the maritime and urban transportation domain. For the former domain, we use the popular open "Brest" Dataset [2]; and for the latter we use the – similarly – famous "GeoLife" Dataset [3, 4, 5]. Both datasets contain information regarding their location (i.e., longitude, latitude) as well as several other related data, such as, the objects' (users', respectively) identifier and the type of vehicle they use.

The rest of the report is structured as follows: Chapter 2 presents the required steps in order to draw a single dataset. Chapters 3 and 4 present the available filtering (and colorization, respectively) methods as well as the required steps in order to add them to the Canvas. Finally, at Chapter 5 we finalize the report by demonstrating the capabilities of *ST_Visions* through some advanced yet real-life data analytics scenarios.

Chapter 2

Visualizing a Single Dataset

In this chapter, we utilize *ST_Visions* in order to get an interactive visualization of a mobility dataset in a quick-and-easy way. For demonstration purposes, we use the "Brest" Dataset, presented at Chapter 1.

2.1 Loading Data

After importing our modules, we create an instance of our class to contain our dataset and prepare our canvas. To load a dataset to our instance there are two approaches, either from a CSV file or from a (Geo)Pandas (Geo)DataFrame.

2.1.1 Loading Data given a CSV file

The following lines of code load and prepare a subset (specifically, the first 30000 rows) of the Brest dataset from a CSV file.

```
1 st_viz = viz.st_visualizer(limit=500)
2 st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
```

The first line creates an *ST_Visions* instance, called "st_viz" with its attributes, "limit" and "proj" (explicitly) set to 500 records, and (implicitly) set to "epsg:3857", respectively. More specifically:

- "limit" determines the number of records that will be visualized. The number of points to rendered is correlated to the performance of the CPU. According to experiments conducted in a single node with 8 CPU cores, 16 GB of RAM and 256 GB of HDD, provided by okeanos-knossos[1], this parameter is default at 30,000 data-points.
- "allow_complex_geometries" Choose to plot either the polygons' exterior (False) or along with its inner voids/holes (True). The value of this parameter defaults to False.
- "proj" determines the Coordinate Reference System (CRS) in which the data will be transformed in order to be properly visualized, in case map tiles[2] are added (defaults to "epsg:3857").

At the second line of code, we use "get_data_csv()" method in order to load a CSV file, with its parameters, "filepath" set to the path of the source file, "sp_columns", i.e., the list of columns that contain the spatial coordinates (implicitly) set to ['lon', 'lat']; and "crs"(implicitly) set to "epsg:4326".

Moreover, using the "**kwargs" argument, more parameters (related to "read_csv") can be specified[3]. For instance, the "nrows" parameter is one of them, specifying the number of rows to be fetched from the CSV file[4].

In a nutshell, "get_data_csv()" works as follows:

[1] An IAAS service for the Greek Research and Academic Community `https://okeanos-knossos.grnet.gr/home/`

[2] Map tiles will be further discussed in the following paragraphs

[3] To learn more about Pandas' "read_csv" method, visit: pandas.pydata.org.

[4] "nrows", is not related, and thus, not necessary equal to "st_viz.limit"

1. Read the CSV file using Pandas' "read_csv" method, according to the "filepath", and "**kwargs" parameters. The latter contains all other arguments related to the aforementioned method;
2. Convert the produced DataFrame to a GeoDataFrame, by adding a geometry column that contains the locations' Point geometry with its coordinates and (initial) projection set according to the "sp_columns" and "crs" parameter, respectively;
3. Project the newly created GeoDataFrame to the CRS necessary for proper visualization, according to the "st_viz.proj" attribute[5]; and
4. Finally, add the resulted DataFrame to the "data" attibute of the *ST_Visions* instance.

2.1.2 Loading Data given a (Geo)DataFrame

Apart from "get_data_csv", one can also use the "set_data" method, in order to directly load a (Geo)Pandas (Geo)DataFrame, as the following lines of code suggest:

```
1 import pandas as pd
2
3 brest_points = pd.read_csv('./data/csv/ais_brest_2015-2016.csv', nrows=30000)
4
5 st_viz = viz.st_visualizer(limit=500)
6 st_viz.set_data(brest_points)
```

At the last line of code, we use "set_data()" method in order to load a DataFrame object, with its parameters, "sp_columns" (i.e., the list of columns that contain the spatial coordinates) and "crs" (implicitly) set to ['lon', 'lat'] and "epsg:4326", respectively. In a nutshell, the aforementioned method works as follows:

1. Given a DataFrame instance and the — ordered pair — of its spatial coordinates, convert it to a GeoDataFrame instance by adding a geometry column that contains the locations' Point geometry with its coordinates and projection set according to the "sp_columns" and "crs" parameters, respectively; However, if "data" is already a GeoDataFrame instance, the above step is not necessary;
2. Project the (newly) created GeoDataFrame to the CRS necessary for proper plotting, according to the "st_viz.proj" attribute[5]; and
3. Finally, add the resulted DataFrame to the "data" attibute of the *ST_Visions* instance.

A necessary clarification at this point is that both CSV files and Pandas DataFrames alike must contain – or implicitly refer, in the sence of spatial columns, to – Point Geometries. Datasets that contain Polygon (or LineString) geometry, must already be a GeoDataFrame instance before calling "set_data" method so as to be properly loaded into the *ST_Visions* instance.

From this point onward we have two choices (depending on the use-case) as far as visualization is concerned; either use the automated methods at the "st_visions.express" module or utilize the instance's methods for a more personalized result.

2.2 Automating the Process using "st_visions.express"

After our dataset is loaded, its time we visualize it on the map. In order for that to be done without much hassle, we can make use of the "st_visions.express" module. Depending on the dataset's geometry type there are three alternative methods at the aforementioned module, namely:

1. plot_points_on_map()
2. plot_polygons_on_map()

[5]To learn more regarding projection management in GeoPandas visit: GeoPandas documentation.

3. plot_lines_on_map()

For (Multi)Point, (Multi)Polygon, and (Multi)Line geometries, respectively. In the sections that will follow, we use data from – or certain variations of – the Brest dataset in order to demonstrate the inputs and outputs of each respective method.

2.2.1 Visualizing (Multi)Point Geometry

In order to effectively visualize (multi)point geometries, we make use of the 30,000 first records of brest dataset and the “plot_points_on_map()” method, as the following code suggests:

```
st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)

tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)
```

This method accepts the most representative arguments for each stage of the visualization process (which will be clarified in the following sections), namely:

1. tools: Any extra tools to be embedded into a Bokeh figure (as a list of the tools’ names as strings – default: None). The basic tools that will be embedded in the figure are “Pan”, “Box Zoom”, “Wheel Zoom”, “Save” and “Reset”[6]
2. map_provider: The name of the map tile provider (default: “CARTODBPOSITRON”)
3. glyph_type: The type of the Glyph that will be used when rendering the data (default: "circle")
4. size: The size of the Glyph (default: 9)
5. color: The primary color of the Glyph (default: “royalblue”)
6. alpha: The overall alpha of the Glyph (default: 0.7)
7. fill_alpha: The inner area alpha of the Glyph (default: 0.7)
8. muted_alpha: The alpha of the Glyph when disabled from the legend (default: 0)
9. legend_label: The label in which the polygons will be represented at the Canvas’ legend (default: “Object GPS Locations”).
10. sizing_mode: How the component should size itself. (default: “scale_width”; allowed values: “fixed”, “stretch_width”, “stretch_height”, “stretch_both”, “scale_width”, “scale_height”, “scale_both”).

In conclusion, the complete code that vizualises the Brest dataset at a Jupyter Notebook, using the “plot_points_on_map” method, is shown below, with its results illustrated at Figure 2.1:

```
import st_visions.st_visualizer as viz
import st_visions.express as viz_express

st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)

tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)

st_viz.show_figures(notebook=True)
```

[6] To learn more about Bokeh figure tools visit: Bokeh Documentation

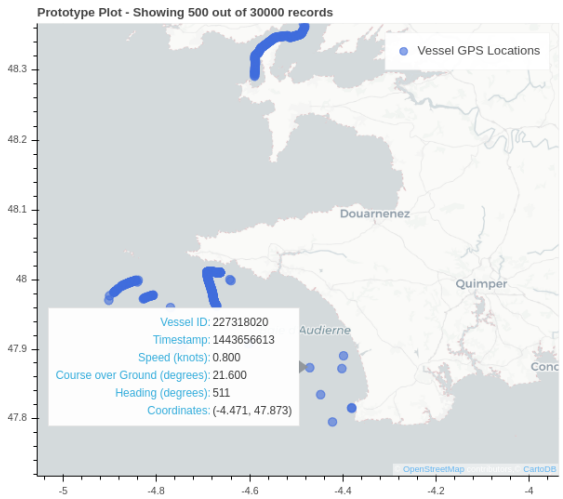


Figure 2.1: Creating an Interactive Figure using Automated Scenarios – a Demonstration using Brest Dataset

2.2.2 Visualizing (Multi)Polygon Geometry

In order to effectively visualize (multi)polygon geometries, the "plot_polygons_on_map()" method will be used, as the following code suggests:

```
st_viz = viz.st_visualizer(limit=500)
st_viz.set_data(brest_ports)

tooltips = [('Port ID','@gml_id'), ('Libelle Po.','@libelle_po'), ('Insee Comm.','@insee_comm'), ('
    Location','(@por_x, @por_y)')]
viz_express.plot_polygons_on_map(st_viz, tools=['hover'], tooltips=tooltips)
```

This method works similar to "plot_points_on_map()", as presented at Chapter 2.2.1, with the only difference being that the call to "add_glyph" is now replaced by "add_polygon". As the aforementioned method, this too accepts the most representative arguments for each stage of the visualization process, namely:

1. tools: Any extra tools to be embedded into a Bokeh figure (as a list of the tools' names as strings – default: None). The basic tools that will be embedded in the figure are "Pan", "Box Zoom", "Wheel Zoom", "Save" and "Reset"[7]
2. map_provider: The name of the map tile provider (default: "CARTODBPOSITRON")
3. polygon_type: The type of the (Multi)Polygon (default: "patches")
4. fill_color: The fill color of the (Multi)Polygon (default: "royalblue")
5. line_color: The line color of the (Multi)Polygon (default: "royalblue")
6. alpha: The overall alpha of the (Multi)Polygon (default: 0.7)
7. fill_alpha: The inner area alpha of the (Multi)Polygon (default: 0.7)
8. muted_alpha: The alpha of the (Multi)Polygon when disabled from the legend (default: 0)
9. legend_label: The label in which the polygons will be represented at the Canvas' legend (default: "Polygon Locations").

[7]To learn more about Bokeh figure tools visit: Bokeh Documentation

10. sizing_mode: How the component should size itself. (default: “scale_width”; allowed values: “fixed”, “stretch_width”, “stretch_height”, “stretch_both”, “scale_width”, “scale_height”, “scale_both”).

In conclusion, the complete code that vizualises the Ports of the Brest dataset at a Jupyter Notebook, using the “plot_polygons_on_map” method, is shown below, with its results illustrated at Figure 2.2:

```
import pandas as pd
import st_visions.st_visualizer as viz
import st_visions.express as viz_express

brest_ports = pd.read_pickle('data/pkl/ports_raw.pkl')

st_viz = viz.st_visualizer(limit=500)
st_viz.set_data(brest_ports)

tooltips = [('Port ID','@gml_id'), ('Libelle Po.','@libelle_po'), ('Insee Comm.','@insee_comm'), ('
    Location','(@por_x, @por_y)')]
viz_express.plot_polygons_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)

st_viz.show_figures(notebook=True)
```

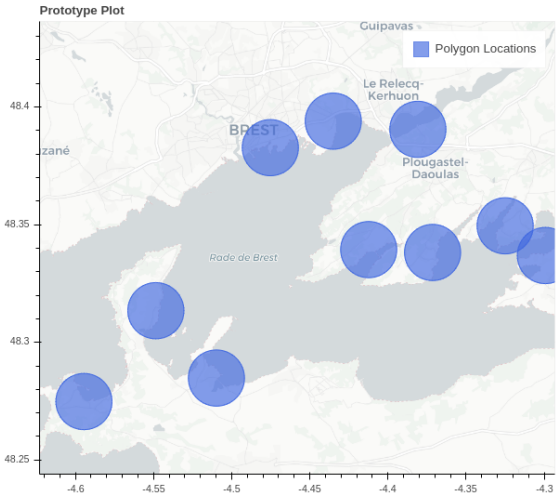


Figure 2.2: Creating an Interactive Figure using Automated Scenarios – a Demonstration using the Ports of the Brest Dataset

2.2.3 Visualizing (Multi)LineString Geometry

In order to effectively visualize (multi)linestring geometries, the “plot_lines_on_map()” method will be used, as the following code suggests:

```
st_viz = viz.st_visualizer(limit=500)
st_viz.set_data(brest_trajectories)

tooltips = [('Port ID','@gml_id'), ('Libelle Po.','@libelle_po'), ('Insee Comm.','@insee_comm'), ('
    Location','(@por_x, @por_y)')]
viz_express.plot_lines_on_map(st_viz, tools=['hover'], tooltips=tooltips)
```

This method works similar to the methods presented at Chapters 2.2.1 and 2.2.2, with the only difference being that the call to “add_polygon” (or “add_glyph”, respectively) is now replaced by “add_line”. Like the previous methods, it accepts the most representative arguments for each stage of the visualization process (which will be clarified in the following paragraphs), namely:

1. tools: Any extra tools to be embedded into a Bokeh figure (as a list of the tools' names as strings – default: None). The basic tools that will be embedded in the figure are "Pan", "Box Zoom", "Wheel Zoom", "Save" and "Reset"[8]
2. map_provider: The name of the map tile provider (default: "CARTODBPOSITRON")
3. line_type: The type of line (default: "patches")
4. line_color: The primary color of the line (i.e., the color to use to stroke lines with – default: "royalblue")
5. line_width: The line stroke width (in units of pixels – default: 5)
6. alpha: The overall alpha of the line (default: 0.7)
7. muted_alpha: The alpha of the line when disabled from the legend (default: 0)
8. legend_label: The label in which the lines will be represented at the Canvas' legend (default: "Polygon Locations").
9. sizing_mode: How the component should size itself. (default: "scale_width"; allowed values: "fixed", "stretch_width", "stretch_height", "stretch_both", "scale_width", "scale_height", "scale_both").

In conclusion, the complete code that vizualises the Trajectories of the Brest dataset at a Jupyter Notebook, using the "plot_lines_on_map" method, is shown below, with its results illustrated at Figure 2.3:

```
1  import pandas as pd
2  import st_visions.st_visualizer as viz
3  import st_visions.express as viz_express
4
5  brest_trajectories = pd.read_pickle('data/pkl/brest_trajectories.pkl')
6
7  st_viz = viz.st_visualizer(limit=500)
8  st_viz.set_data(brest_trajectories)
9
10 tooltips = [('Port ID','@gml_id'), ('Libelle Po.','@libelle_po'), ('Insee Comm.','@insee_comm'), ('
       Location','(@por_x, @por_y)')]
11 viz_express.plot_lines_on_map(st_viz, tools=['hover'], tooltips=tooltips)
12
13 st_viz.show_figures(notebook=True)
```

A necessary clarification at this point is that, in addition to the arguments of the previously mentioned methods (i.e., plot_points_on_map, plot_polygons_on_map, and plot_lines_on_map), the user can define several more via the **kwargs argument in order to further customize his Canvas. The "tooltips" parameter is one of those extra parameters, indicating the features that will be shown when the cursor hovers over a point, polygon, or line, respectively.

[8]To learn more about Bokeh figure tools visit: Bokeh Documentation

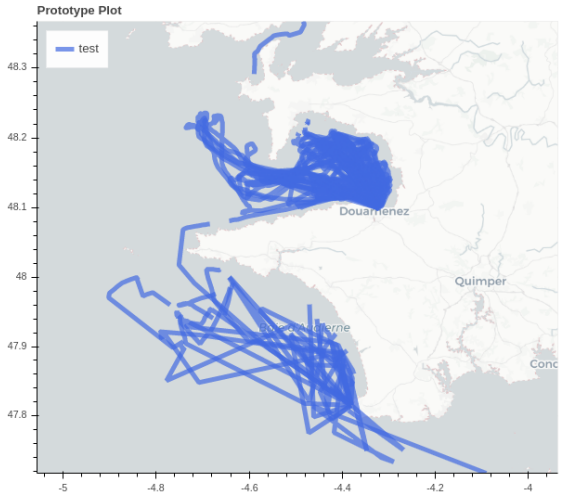


Figure 2.3: Creating an Interactive Figure using Automated Scenarios – a Demonstration using the data-points of the Brest Dataset in the form of LineStrings, one for each of the vessels' trajectories.

2.3 The Inner Functionality of "st_visions.express" methods

Having visualized our dataset using the automated methods of the "st_visualizer.express" module its time to delve into their inner functionality.

2.3.1 Creating a Canvas

Let's return to the stage that we loaded our dataset. From this point, its time to create our canvas, i.e., the backbone in which our figure will eventually be build. In order to do that, the "create_canvas" method must be used, as the following line of code suggests:

```
1 st_viz.create_canvas(title='Prototype Plot', sizing_mode='scale_width', plot_height=540)
```

This method provides several customization parameters, namely:

- title: The title of the figure (required);
- x_range, y_range: The coordinates' range, i.e., where will the resulted figure will focus to (defaults to *None*); and
- suffix: The suffix for the coordinates' names once they have been projected to the target CRS, according to "st_viz.proj" (defaults to "_merc")

Except for the above arguments, the user can provide several others (via the "**kwargs" parameter), related to Bokeh's "figure" object[9] in order to get more customization levels. For instance, sizing_mode and plot_height (plot_width, respectively) are some of these "extra" parameters, which control the aspect ratio and figure size, respectively.

In summary, "create_canvas" operates according to the following steps:

1. In case $st\_viz.limit \leq len(st\_viz.data)$, suffix given title with the following statement: " – showing X out of Y records.';
2. Get the Minimum Bounding Box (MBB) of the loaded data. In case either x_range or y_range are not given (i.e. the value is *None*), the respective axes' range is set according to the MBB;
3. Instantiate a Bokeh figure (i.e., the canvas), with all the parameters we gave at "create_canvas" (including the ones given at "**kwargs").

[9]To learn more about Bokeh's "figure" arguments visit: docs.bokeh.org

4. Instantiate a Bokeh ColumnDataSource (CDS). In a nutshell, it serves as a communication bridge between a Pandas DataFrame and our canvas, able to provide data in order to get interactive plots. Its usefulness will be further clarified in the following (and more advanced) use-cases; and

5. Finally, set the CDS, and figure as public attributes ("source" and "figure", respectively) and the column suffix as a private attribute ("__suffix") of the st_viz instance.

2.3.2 Vizualizing Geometries

Having created our main canvas it is time to visualize the dataset. Depending on the dataset's geometry type there are three alternative methods for geometry visualization, namely:

1. add_glyph()

2. add_polygon()

3. add_line()

For (Multi)Point, (Multi)Polygon, and (Multi)Line geometries, respectively. In the sections that will follow, we clarify each method, in terms of both documentation and use.

Visualizing (Multi)Point Geometries

Using the "add_glyph" method, we can use the CDS produced at the previous step in order to plot the data in our figure, as the following code suggests:

```
circular_glyph = st_viz.add_glyph(glyph_type='circle', size=10, color='royalblue', alpha=0.7,
    fill_alpha=0.5, muted_alpha=0, legend_label=f'Vessel GPS Locations')
```

Like "create_canvas", so too this method provides several customization options, namely:

- glyph_type: The type of Glyph to be vizualized. (default value: "circle"; allowed values: "asterisk", "cross", "circle", "circle_x", "circle_cross", "dash", "diamond", "diamond_cross", "hex", "inverted_triangle", "triangle")

- size: The glyph's size (in screen units).

- color: The glyph's color[10].

- sec_color: The glyph's secondary color. It is visible only when a selection tool (e.g. Box/Lasso Selection) is active and a certain glyph is outside the selected area.

- alpha: Measures the glyphs' opacity (0: transparent, 1: opaque)

- muted_alpha: The glyphs' opacity when deactivated[11].

Visualizing (Multi)Polygon Geometries

Using the "add_polygon" method, we can use the CDS produced at the previous step in order to plot the data in our figure, as the following code suggests:

```
polygons = st_viz.add_polygon(polygon_type='patches', legend_label="Brest Ports' Location")
```

Using the aforementioned method, we can use the CDS produced at the previous step in order to plot the data in our figure. As "create_canvas", so too this method provides several customization options, namely:

[10] To learn more about the available colors visit: Bokeh Documentation

[11] Glyph selection/deactivation will be discussed in the following paragraphs.

- polygon_type: The type of Polygon to be visualized. (default value: “patches”; allowed values: “patches”, “multi_polygons”). “patches” is useful for simple polygons (i.e., shapely.geometry.Polygon instances), while “multi_polygons” is useful for complex geometries, such as polygons with holes (i.e., shapely.geometry.MultiPolygon instances).
- line_width: The polygons’ border line width (in screen units – default: 1)
- line_color: The polygons’ border line color[12] (default: “royalblue”)
- fill_color: The polygons’ area color[12] (default: “royalblue”)
- fill_alpha: Measures the polygons’ area opacity (0: transparent, 1: opaque – default: 0.5)
- muted_alpha: The polygons’ overall opacity when deactivated[13] (default: 0).

Visualizing (Multi)LineString Geometries

```
lines = st_viz.add_line(line_type='multi_line', line_color="royalblue", line_width=5, alpha=0.7,
    muted_alpha=0, legend_label='Brest Trajectories')
```

With the above method, we can use the CDS produced at the previous step in order to plot the data in our figure. As “create_canvas”, so too this method provides several customization options, namely:

- line_type: The type of Polygon to be visualized. (default value: “multi_line”; allowed values: “hline_stack”, “line”, “multi_line”, “step”, “vline_stack”).
- line_width: The lines’ width (in screen units – default: 1)
- line_color: The lines’ line color[14] (default: “royalblue”)
- alpha: Measures the lines’ area opacity (0: transparent, 1: opaque – default: 0.5)
- muted_alpha: The lines’ overall opacity when deactivated[15] (default: 0).

A necessary clarification at this point is that in addition to the previous methods’ parameters, by using the “**kwargs” parameter even more arguments can be set, in order to get a more personalized result. For instance, legend_label is one of these “extra” parameters, that represents the drawn elements (i.e., glyphs, polygons, or linestrings) at the Canvas’ legend.

2.3.3 Adding Map Tiles

After that step, we can plot the figure as-is, but the result will be an empty figure with just the glyphs introduced, as Figure 2.4 indicates. So, in order to start making some meaningful observations, we need to add a map tile. This can be done by simply calling the “add_map_tile” method. The following line of code sets (explicitly) a map tile from “CartoDB”, and (implicitly) at retina resolution.

```
st_viz.add_map_tile('CARTODBPOSITRON')
```

More specifically, the aforementioned method provides two parameters:

- provider: The name of the map tile’s provider. Figure 2.5 provides a sample of the available map tiles. (default value: “CARTODBPOSITRON”; allowed values: “CARTODBPOSITRON”, “STAMEN_TERRAIN”, “STAMEN_TONER”, “STAMEN_TONER_LABELS”, “STAMEN_TONER_BACKGROUND”)

[12] To learn more about the available colors visit: Bokeh Documentation
[13] Polygon deactivation will be discussed in the following paragraphs.
[14] To learn more about the available colors visit: Bokeh Documentation
[15] Polygon deactivation will be discussed in the following paragraphs.

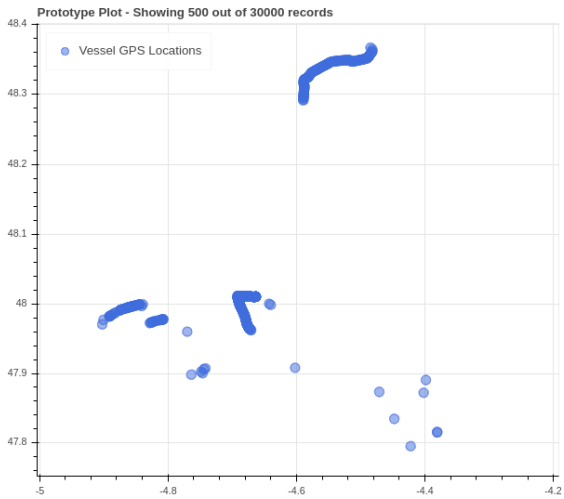


Figure 2.4: Creating an interactive Figure I – Visualizing the available data points on the canvas

- retina: Determines the resolution of the map tile. If *False* the tile will be downloaded in standard resolution, otherwise it will be downloaded at retina resolution, useful for displays with high pixel density.
- level: The z-order of the map tiles (default value: "underlay"). 'underlay' means that the map tiles will be always at the back of the plot (i.e., z-order=0).

After adding a map tile to Figure 2.4, the end result is shown at Figure 2.6a. Now panning and/or zooming the newly created Figure, may reveal some interesting patterns or behaviours.

2.3.4 Canvas Interactivity

Up to this point, our Canvas is pretty much ready to be drawn, but it still lacks another layer of interaction, for instance, the ability to know for each datapoint its features (e.g. speed, heading, vehicle type, etc.).

Adding a Hover Tooltip Utility

So, in order to access these features on-demand we will add a hover tooltip that will show for each datapoint, its respective features, as the following code suggests.

```
1 features = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over
      Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
2 st_viz.add_hover_tooltips(features)
```

At first, we need to specify the necessary features in order for them to be shown properly in our canvas. To specify them we need to set a list of tuples of two elements, the first being the title and the second one being the name (suffixed by "@") of the feature, respectively. Then, by calling the "add_hover_tooltips" method, they are integrated in the canvas. Figure 2.6b illustrates the result of Figure 2.6a after adding our hover tooltips.

Aside the (mandatory) "feature" argument, the user can achieve higher customization levels by using the **kwargs parameter, in order to pass even more arguments to the aforementioned function.

Adding a Lasso/Box Select Utility

Except Hover Tooltips, the user can also add other utilities to the canvas, such as lasso or box select. The following line of code adds a lasso select tool to the created canvas. Note however that as of the

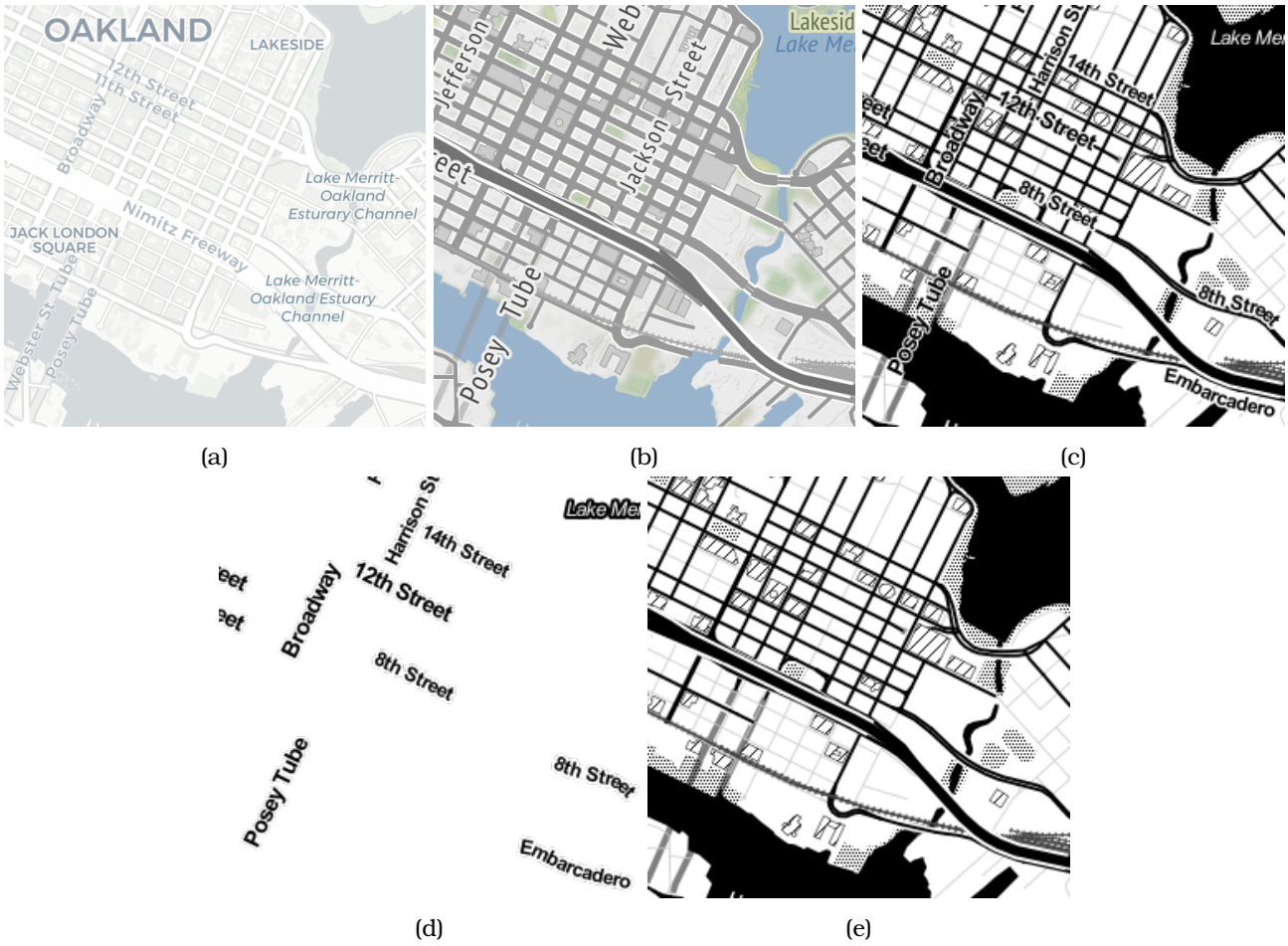


Figure 2.5: Sample of the available map tiles (a) CartoDB; (b) Stamen Terrain; (c) Stamen Toner; (d) Stamen Toner (labels only); and (e) Stamen Toner (background only) at retina resolution (whenever available).

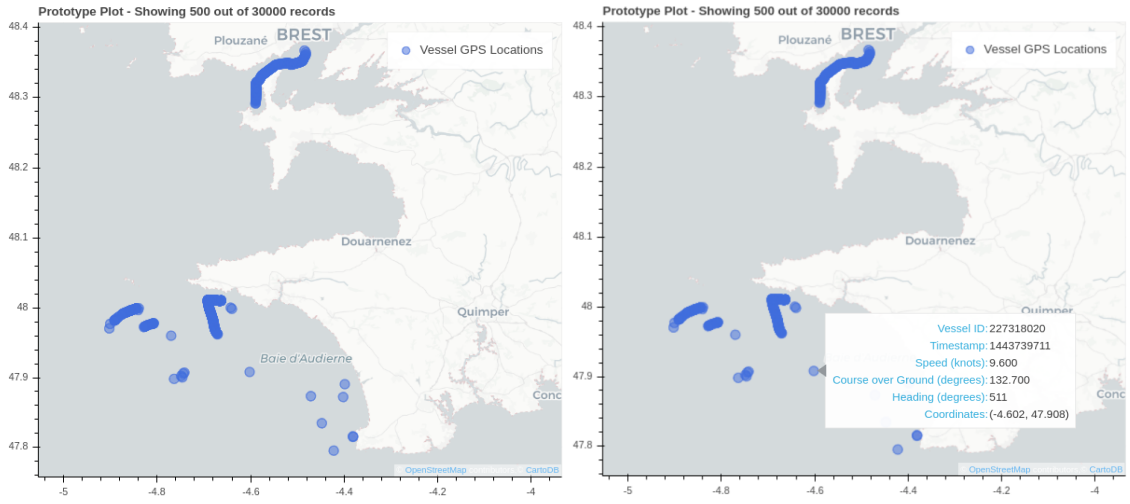
period this report is being written, Bokeh does not support (Multi)Polygon as well as (Multi)LineString element selection.

```
1 st_viz.add_lasso_select()
```

Up to this point our canvas is ready to be rendered to the user, who will now be able to get a visual summary of the nature of the dataset at hand. But there are even more customization options available, in order to further enrich the canvas' possibilities. The following lines of code do the following customization actions:

- Set the location of the legend from the top right to the top left;
- Change the click policy of the legend to "mute", in order to make its items not only clickable, but also when clicked, the referenced objects will dissapear (that's where the "muted_alpha" argument of "st_viz.add_glyph" method is useful); and
- Set the active item of the figures' toolbar to wheel zoom tool (WheelZoomTool) for instant pan and zoom.

```
1 st_viz.figure.legend.location = "top_left"
2 st_viz.figure.legend.click_policy = "mute"
3 st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)
```



(a) (b)

Figure 2.6: Creating an interactive Figure II – (a) Adding a Map Tile to the created Canvas (Provider: CartoDB, Resolution: Retina); (b) Adding a Hover Tooltip to the Canvas' displayed datapoints

2.3.5 Canvas Rendering

Finally, in order to show our figure, we will use the "show_figures" method. It essentially creates a local Bokeh server in which the rendered Canvas can be observed. If the "notebook" parameter is set to True, the Canvas will be drawn at a Jupyter Notebook cell at its default IP address (i.e., http://localhost:8888). The following line of code draws the (final) canvas at a Jupyter Notebook cell.

```
st_viz.show_figures(notebook=True)
```

To create visualizations via Python script, set "notebook" parameter to **False** and to execute it write in a Terminal instance:

```
[python -m] bokeh serve --show <PATH_TO_SCRIPT> --allow-websocket-origin=<COMPUTER_IP_ADDRESS>:<PORT>
```

A necessary clarification at this point is that "show_figures" is not only able to draw a single canvas (such as the one created at this chapter) but also render an ensemble of canvases in a grid, in whichever layout the user specifies. The parameters of that method will be explained in the following chapters. For the current use-case, the only parameters that are concerned are "notebook" and "notebook_url" (in the case that $notebook = True$).

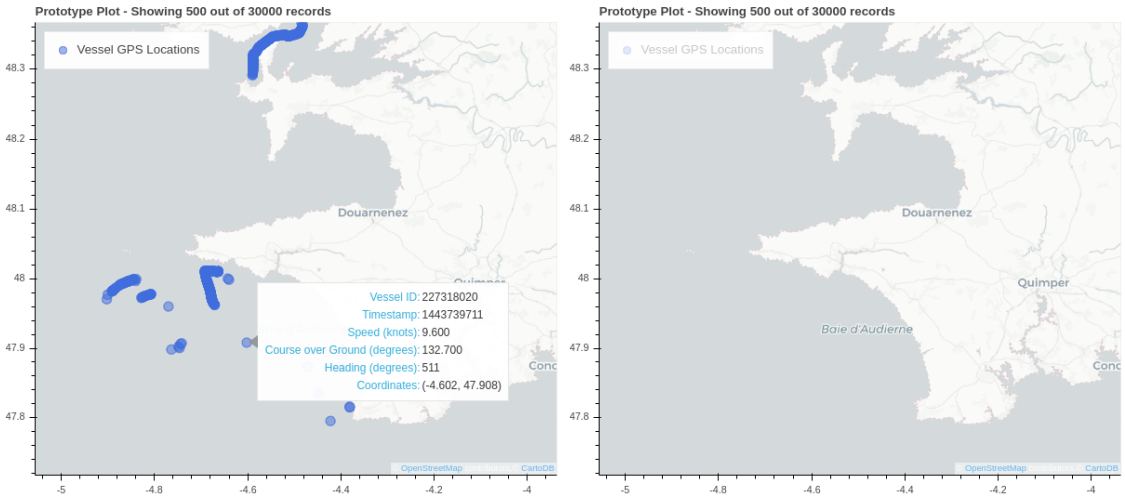
2.3.6 Complete Code

In conclusion, the complete codes that produces an interactive visualization of the Brest dataset (and its variations) at a Jupyter Notebook is shown below, with the results illustrated at their respective Figures.

Visualizing (Multi)Point Geometry

```
import st_visions.st_visualizer as viz

st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)

st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)

st_viz.add_map_tile('CARTODBPOSITRON')
_ = st_viz.add_glyph(glyph_type='circle', size=10, color='royalblue', alpha=0.7, fill_alpha=0.5,
    muted_alpha=0, legend_label=f'Vessel GPS Locations')

```

```
tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over
    Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
st_viz.add_hover_tooltips(tooltips)
st_viz.add_lasso_select()

st_viz.figure.legend.location = "top_left"
st_viz.figure.legend.click_policy = "mute"
st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)

st_viz.show_figures(notebook=True)
```



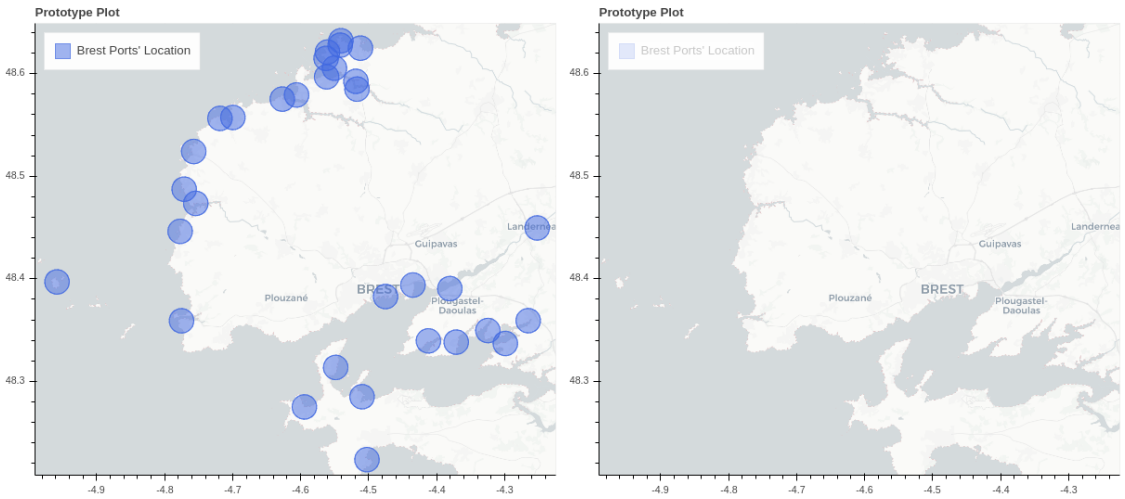
(a) (b)

Figure 2.7: Creating an interactive Figure – (a) Final Result; (b) Demonstrating interactive legend by muting an item's respective datapoints on click

Visualizing (Multi)Polygon Geometry

```
import pandas
import st_visions.st_visualizer as viz

brest_ports = pd.read_pickle('data/pkl/ports_raw.pkl')

st_viz = viz.st_visualizer(limit=500)
st_viz.set_data(brest_ports)

st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)

st_viz.add_map_tile('CARTODBPOSITRON')
plgn = st_viz.add_polygon(polygon_type='patches', legend_label="Brest Ports' Location")

tooltips = [('Port ID','@gml_id'), ('Libelle Po.','@libelle_po'), ('Insee Comm.','@insee_comm'), ('
    Location','(@por_x, @por_y)')]
st_viz.add_hover_tooltips(tooltips)

st_viz.figure.legend.location = "top_left"
st_viz.figure.legend.click_policy = "mute"
st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)

st_viz.show_figures(notebook=True)
```

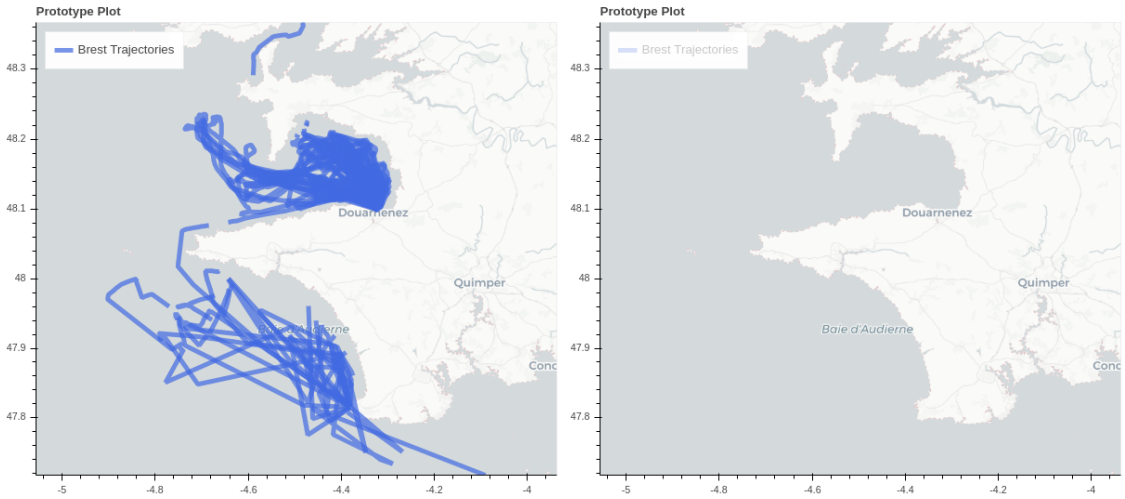
Visualizing (Multi)LineString Geometry



(a) (b)

Figure 2.8: Creating an interactive Figure – (a) Final Result; (b) Demonstrating interactive legend by muting a respective item's polygons on click

```
1  import pandas
2  import st_visions.st_visualizer as viz
3  import st_visions.geom_helper as viz_helper
4
5  # Prepare Data
6  df = pd.read_csv('data/csv/ais_brest_2015-2016.csv', nrows=30000)
7  df = viz_helper.getGeoDataFrame_v2(df, coordinate_columns=['lon', 'lat'])
8  df.sort_values('ts', inplace=True)
9
10 df_trajectories = viz_helper.create_linestring_from_points(df, column_handlers=['mmsi', 'trip_id'])
11
12 # Prepare & Render the Canvas
13 st_viz = viz.st_visualizer(limit=500)
14 st_viz.set_data(df_trajectories)
15
16 st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540, tools="pan,
       box_zoom,lasso_select,wheel_zoom,previewsave,reset")
17
18 st_viz.add_map_tile('CARTODBPOSITRON')
19 _ = st_viz.add_line(line_type='multi_line', line_color="royalblue", line_width=5, alpha=0.7,
       muted_alpha=0, legend_label='Brest Trajectories')
20
21 tooltips = [('Vessel ID', '@mmsi'), ('Trajectory/Trip ID', '@trip_id')]
22 st_viz.add_hover_tooltips(tooltips)
23
24 st_viz.figure.legend.location = "top_left"
25 st_viz.figure.legend.click_policy = "mute"
26 st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)
27
28 st_viz.show_figures(notebook=True)
```



(a) (b)

Figure 2.9: Creating an interactive Figure – (a) Final Result; (b) Demonstrating interactive legend by muting a respective item's polygons on click

Chapter 3

Data Filtering

In this chapter, we utilize *ST_Visions* in order to interactively filter the values of a mobility dataset in a quick-and-easy way. For demonstration purposes, we use the "Brest" Dataset (for Temporal and Numerical Filtering) and the "GeoLife" Dataset for Categorical Filtering, both presented at Chapter 1.

A necessary clarification regarding the filters' operation is that they operate directly on the loaded (at the instance) data. The "st_viz.limit" parameter is applied after the filtering operation and before the rendering process in order to alleviate any issues stemming from overwhelming data volume.

3.1 Temporal Filtering

Given an *ST_Visions* instance with properly defined data and geometry method we demonstrate our class' capability for interactive data filtering. In this section, we filter our dataset in the temporal axis via the "add_temporal_filter" method[1].

Temporal filtering occurs within a user-defined temporal horizon via a DateRangeSlider[2]. Incorporating it and can be done using a single line of code, as the following code block suggests:

```
st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)
...
st_viz.add_temporal_filter(temporal_name='ts', temporal_unit='s', step_ms=500, title='Temporal Horizon
    ', height_policy='min', callback_policy='value_throttled')
```

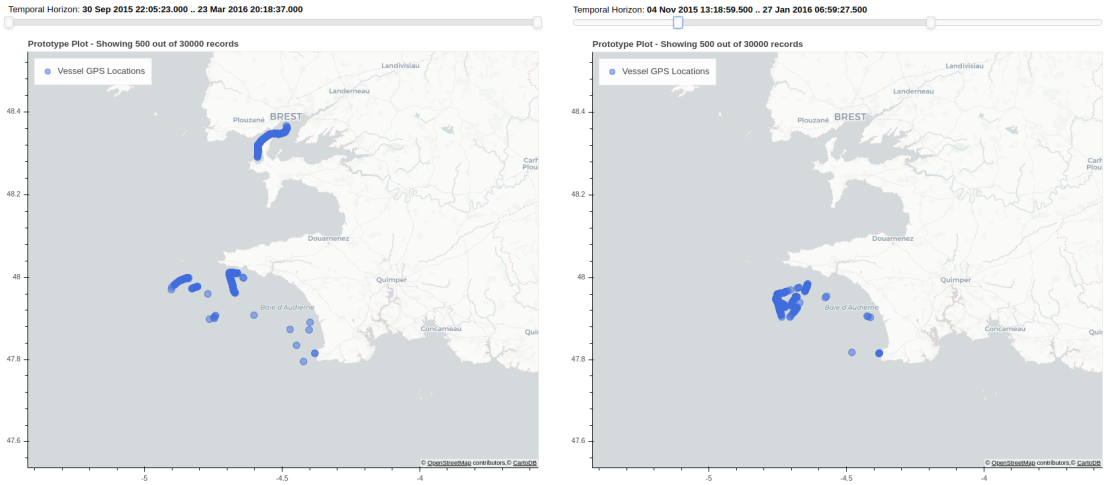
The aforementioned method supports a wide spectrum of parameters, namely:

- temporal_name: The column of the data that contains the temporal information
- temporal_unit: The temporal axis' unit (default: 's')
- step_ms: The step (in ms) in which the temporal horizon will be advanced (default: 3600000 – 1 hr.)
- title: The filter's title (default: 'Temporal Horizon')
- height_policy: The widget's height policy (default value: 'min'; allowed values: 'auto', 'min', 'max', 'fit', 'fixed')
- callback_policy: The widget's callback policy. (default value: 'value_throttled'; allowed values: 'value', 'value_throttled'). In a nutshell this parameter controls the vidget's behaviour. Setting it to 'value' will update the figure on-the-fly, while setting it to 'value_throttled' will update the figure after-the-fact.

[1] To avoid any code repetitions, we continue from the complete code provided at either Section 2.2 or Section 2.3.6.
[2] To learn more about DateRangeSlider visit: Bokeh Documentation

- callback_class: callbacks.BokehFilters (default: None). Allows custom callback methods to be set[3].

In addition to the above arguments, the user can specify even more (related to the widget's definition) via the **kwargs parameter. Figure3.1a illustrates the given dataset at the entirety of its temporal horizon. This is the default behaviour, as produced by the class' output, and its as if the filter is non-existent. Its effect however, is visible when the temporal window is adjusted, as shown by Figure 3.1b.



(a) (b)

Figure 3.1: Temporal Filtering at (a) the dataset's whole horizon (default); and (b) at a used-defined time window. (Notice that the number of points always stay constant, and equal to the "limit" parameter of our instance.)

3.2 Categorical Filtering

In this section, we interactively filter our dataset according to its categorical attribute(s) via the "add_categorical_filter" method[1].

Categorical filtering occurs within a user-defined feature value selection via a DropDown Menu[4]. Incorporating it can be done using a single line of code, as the following code block suggests:

```
st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/geolife_trips_cleaned_v2_china_subset.csv', nrows=30000)
st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)
...
st_viz.add_categorical_filter(title='Category', categorical_name='mmsi', options=None, value=None,
    height_policy='min')
```

The aforementioned method supports a wide spectrum of parameters, namely:

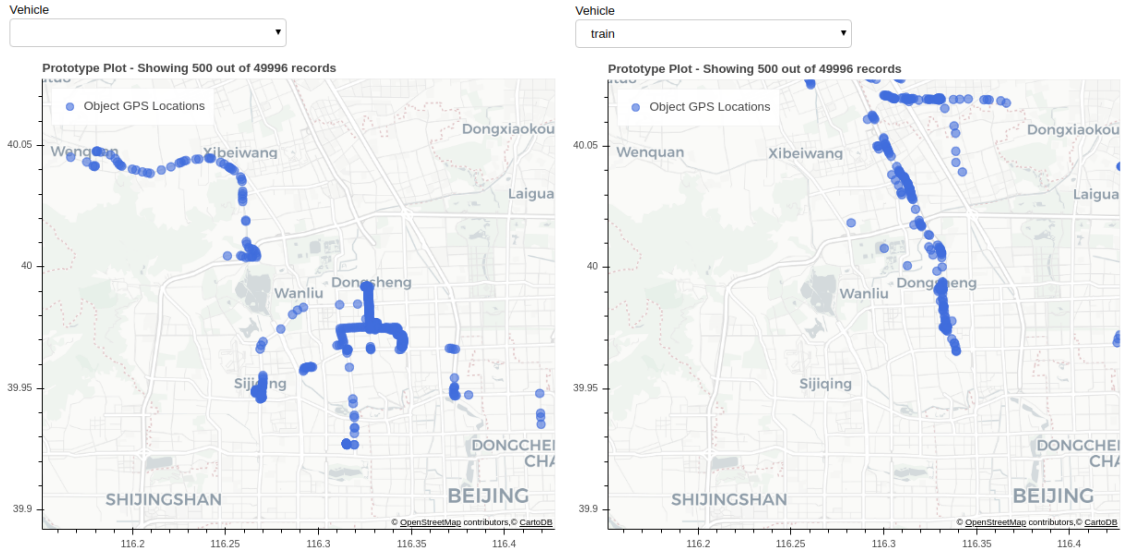
- title: The filter's title (default: 'Category')
- categorical_name: The column of the data that contains the categorical information
- options: The list of tuples of feature values in the form of [($feature\_value_1$, $feature\_label_1$) ... ($feature\_value_n$, $feature\_label_n$)] that will determine the filter's available options (default: None; they will be automatically determined by the respective column's values in the form of [($feature\_value_1$, $feature\_value_1$) ... ($feature\_value_n$, $feature\_value_n$)])

[3]More regarding custom callbacks will be discussed at Section 3.5
[4]To learn more about DropDown Menu visit: Bokeh Documentation

- value: The initial value of the filter (default: None)
- height_policy: The widget's height policy (default value: 'min'; allowed values: 'auto', 'min', 'max', 'fit', 'fixed')
- callback_class: callbacks.BokehFilters (default: None) Allows custom callback methods to be set[5].

In addition to the above arguments, the user can specify even more (related to the widget's definition) via the **kwargs parameter. Note however, that the (categorical) values of the given feature must be in String format[6]

Figure 3.2 summarizes the above discussion. More specifically, Figure 3.2a illustrates the dataset at the entirety of its categorical values. This is the default behaviour, as produced by the class' output, and it's as if the filter is non-existent. Its effect however, is visible when a certain value is specified, as shown by Figure 3.2b



(a) (b)

Figure 3.2: Categorical Filtering at (a) all the dataset's categories (default); and (b) at a certain category. (Notice that the number of points always don't exceed in cardinality, the "limit" parameter of our instance.)

3.3 Numerical Filtering

In this section, we interactively filter our dataset according to its numerical attribute(s) via the "add_numerical_filter" method[1].

Numerical filtering occurs within a user-defined feature via a Slider[7]. Incorporating it can be done using a single line of code, as the following code block suggests:

```
1 st_viz = viz.st_visualizer(limit=500)
2 st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
3 st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)
4 ...
5 st_viz.add_numerical_filter(title='Speed (knots)', filter_mode='>=', numeric_name='speed', step=1,
      callback_policy='value')
```

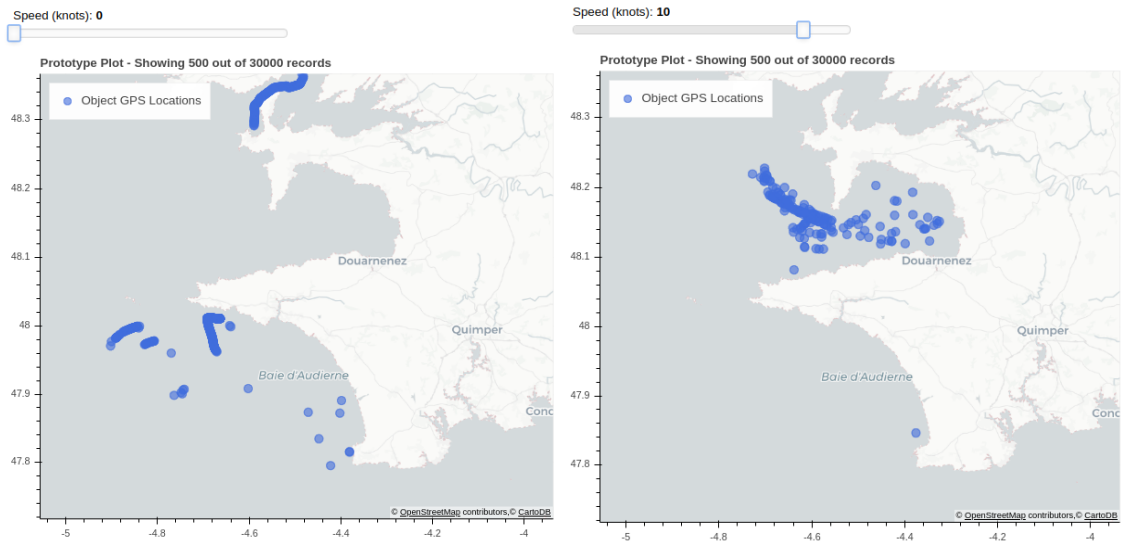
The aforementioned method supports a wide spectrum of parameters, namely:

[5]More regarding custom callbacks will be discussed at Section 3.5
[6]So far Bokeh accepts categorical values in String format. Workarounds for other data types are WIP.
[7]To learn more about Slider visit: Bokeh Documentation

- filter_mode: Determines the filter's behaviour, given a value (default value: '>='; allowed values: '==', '!=', '<', '<=', '>', '>=', 'range')
- title: The filter's title (default: 'Value')
- numeric_name: The column of the data that contains the numerical information
- value: The initial value of the filter (default: None – will be the minimum value of the dataset's given numerical feature)
- step: The filter's numerical step (default: 50)
- height_policy: The widget's height policy (default value: 'min'; allowed values: 'auto', 'min', 'max', 'fit', 'fixed')
- callback_policy: The widget's callback policy. (default value: 'value_throttled'; allowed values: 'value', 'value_throttled'). In a nutshell this parameter controls the vidget's behaviour. Setting it to 'value' will update the figure on-the-fly, while setting it to 'value_throttled' will update the figure after-the-fact.
- callback_class: callbacks.BokehFilters (default: None) Allows custom callback methods to be set[8].

In addition to the above arguments, the user can specify even more (related to the widget's definition) via the **kwargs parameter. Note however, that the (categorical) values of the given feature must be in String format[9]

Figure 3.3 summarizes the above discussion. More specifically, Figure3.3a illustrates the dataset at the entirety of its numerical spectrum. This is the default behaviour, as produced by the class' output, and it's as if the filter is non-existent. Its effect however, is visible when the filter's value is changed, as shown by Figure 3.3b



(a) (b)

Figure 3.3: Numerical Filtering at (a) all the dataset's entire spectrum (default); and (b) at a certain value. (Notice that the number of points always don't exceed in cardinality, the "limit" parameter of our instance.)

[8]More regarding custom callbacks will be discussed at Section 3.5

[9]So far Bokeh accepts categorical values in String format. Workarounds for other data types are WIP.

3.4 Complete Code

In this section we provide the complete code for each use-case of the previous chapters. The codes that follow, have been executed as Python Scripts (via a Bokeh local server) and accessed via Google Chrome.

3.4.1 Temporal Filtering

```
import st_visions.st_visualizer as viz
import st_visions.express as viz_express

st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)

tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)

st_viz.add_temporal_filter(width_policy='fit')

st_viz.figure.legend.location = "top_left"
st_viz.figure.legend.click_policy = "mute"
st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(WheelZoomTool)

st_viz.show_figures(notebook=False)
```

3.4.2 Categorical Filtering

```
import st_visions.st_visualizer as viz
import st_visions.express as viz_express

st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv('./data/csv/geolife_trips_cleaned_v2_china_subset.csv', sp_columns=['lon', 'lat'])

tooltips = [('User ID','@user_id'), ('Vehicle','@label'), ('Location','(@lon, @lat, @alt)'), ('Timezone', '@rec_dt')]
viz_express.plot_points_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)


st_viz.add_categorical_filter(title='Vehicle', categorical_name='label')
# st_viz.add_numerical_filter(filter_mode='>=')

st_viz.figure.legend.location = "top_left"
st_viz.figure.legend.click_policy = "mute"
st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(bkhm.WheelZoomTool)

st_viz.show_figures(notebook=False)
```

3.4.3 Numerical Filtering

```
import st_visions.st_visualizer as viz
import st_visions.express as viz_express

st_viz = st_visualizer(limit=500)
st_viz.get_data_csv('./data/csv/ais_brest_2015-2016.csv', sp_columns=['lon', 'lat'], nrows=30000)

tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz, tools=['hover,lasso_select'], tooltips=tooltips)

st_viz.add_numerical_filter(title='Speed (knots)', filter_mode='>=', numeric_name='speed', step=1, callback_policy='value')

```

```
12 st_viz.figure.legend.location = "top_left"
13 st_viz.figure.legend.click_policy = "mute"
14 st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(WheelZoomTool)
15
16 st_viz.show_figures(notebook=False)
```

3.5 A Note on Multiple Filter Interaction

Bokeh is without a doubt, a powerful library for interactive visualizations. While that may hold true, when multiple filters (i.e. widgets such as, Sliders, DropDown menus, etc.) are introduced it cannot automatically synchronize them, thus resulting in chaos, if they are not controlled properly. A baseline workaround for that problem, is using a shared callback for all introduced widgets. While this solution is quite useful, it cannot be generalized for multiple dynamically introduced widgets. Thus, in order to alleviate this issue, we propose a solution that can effectively account for multiple widgets while not having any significant loss in performance.

Algorithm 1: FILTER CALLBACK. The core structure of the filters' callback.

Input: Callback Policy $attr$, Old Value old, New Value new

1 $\_\_iterative\_filter\_data(filter.id)$
2 $filtered\_data = \_\_get\_data()$
3 $filtered\_data = widget\_filter\_data(filtered\_data, new)$
4 $\_\_callback\_prepare\_data(filtered\_data, filter.id == this.lock)$

Algorithm 1 describes the core structure of a filter's callback method in order to co-exist harmonously with the previously introduced widgets. More specifically, given a change in the value of a filter, its respective callback method must:

1. Iterate (via the "widgets" attribute of the "*ST_Visions*" instance) all other introduced widgets and execute their callbacks in order to filter the data (line 1);

2. Fetch the newly filtered data and apply the widget's respective filter method (lines 2–3);

3. Finally, pass the data to the CDS and visualize them on the instance's canvas (line 4).

Algorithm 2, which is in charge for the first step of Algorithm 1, iteratively traverses the instance's introduced widgets and triggers their respective callback methods, in order to apply their respective filter to the dataset at hand. To avoid any deadlocks (i.e. recursive calls to the same callback function), a (private) lock attribute is introduced, which (if None) is assigned with the widgets identifier[10].

After our data are properly (i.e. according to specification) filtered, they are passed as input to Algorithm 3. This algorithm is in charge of preparing the input data for output (i.e. passing them to the instance's CDS). Because the aforementioned method exists in each callback method (as part of its core structure), in order to avoid flickering, as well as performance throttling phenomena, rendering to the canvas is done *only* by the widget the lock is assigned to. Finally, our data are drawn, and the lock is released (i.e. reverted to *None*).

All the above algorithms are integrated into the "BokehFilters" class located at the callbacks.py module, allowing for custom synchronized callbacks. In order to create a custom callback, after inheriting the "BokehFilters" class, one must implement, as baseline, the "callback" method, according to Algorithm 1.

Of course there will be scenarios that need more personalized code. Thus, beyond the "callback" method, all inherited methods can be customized in order to account for every possible use-case. Section 5 covers such scenarios, and further analyze the inner workings of the "BokehFilters" class.

[10] According to Bokeh Documentation each model/widget/filter is assigned with its own unique identifier.

Algorithm 2: __CALLBACK_FILTER_DATA. Data Filtering Pipeline.

Input: Widget Identifier id

if $this.lock = None$ **then**
  $this.lock \leftarrow id$
  foreach $widget \in this.widgets$ **do**
    if $widget.id \neq id$ **then**
      $widget.trigger\_callback(attr = \text{“}value\text{”}, old = None, new = widget.value)$
    end
  end

Algorithm 3: __CALLBACK_PREPARE_DATA. Data Filtering Pipeline.

Input: Filtered Dataset $data$, Output Flag $flag$

$this.CDS\_data \leftarrow data$
/* Q: $Am\ I\ Ready\ to\ Render?$ */
if $flag$ **then**
  $this.renderToCanvas(this.CDS\_data)$
  $this.lock \leftarrow None$ // $Releasing\ Lock...$
  $this.CDS\_data \leftarrow None$ // $Emptying\ Intermediate\ Storage...$

Chapter 4

Data Colorization

In addition to data filtering (c.f. Chapter 3), our library supports data colorization as well. In the following sections, we will present the two colorization methodologies, for categorical and numerical data, respectively, as well as the methods that implement them. For demonstration purposes, we use the "Brest" Dataset, presented at Chapter 1.

4.1 Categorical Data Colorization

Given a properly instantiated *ST_Visions* instance, we demonstrate our class' capability for categorical color mapping. In this section, we colorize the geometry data of our dataset, via the "add_categorical_colormap" method[1].

The categorical colormap is created via a user-defined categorical handler (i.e. a column from the input DataFrame). Creating and incorporating it can be done using a single line of code, as the following code block suggests:

```
1 st_viz = viz.st_visualizer(limit=500)
2 st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
3 st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)
4 ...
5 st_viz.add_categorical_colormap(palette=bokeh_palettes.Category10_3, categorical_name='mmsi')
6 st_viz.add_glyph(glyph_type='circle', size=10, color=st_viz.cmap, alpha=0.8, fill_alpha=0.7,
      muted_alpha=0, legend_group=f'mmsi')
7 ...
8 st_viz.show_figures(notebook=False)
```

The aforementioned method supports a wide spectrum of parameters, namely:

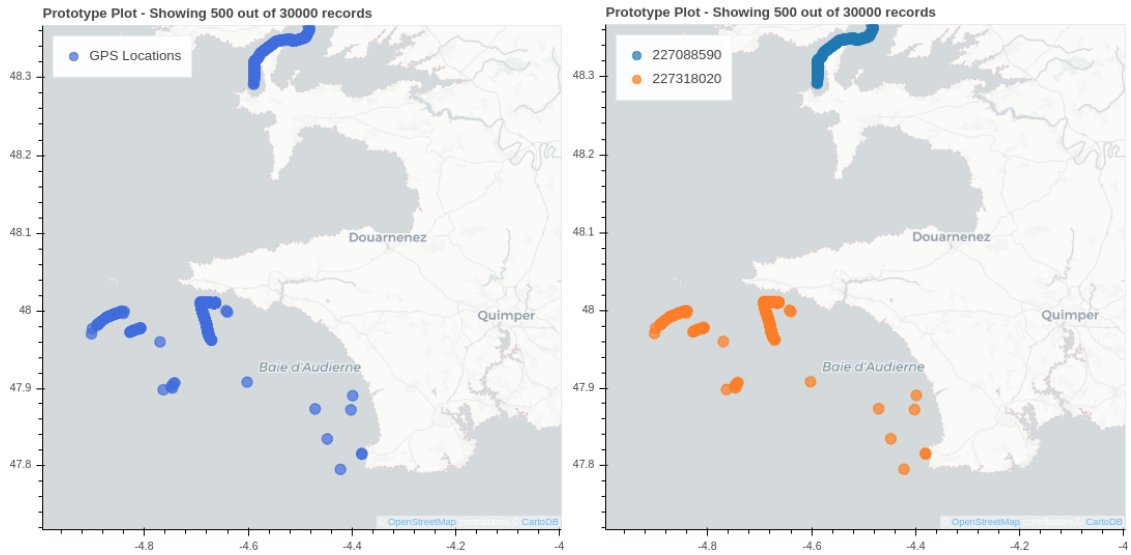
- palette: The color palette which will be used for mapping the categories to their respective colors. The palette can be defined, either as a String that contains its name[2], or as a tuple that contain the hexadecimal value of each color. The latter option can be of use in cases which a huge color palette is required.

- categorical_name: The column name of the given DataFrame that contains the categorical information.

In addition to the above arguments, the user can specify even more (related to the colormap creation) via the **kwargs parameter. Figure4.1 illustrates the given dataset (a) without; and (b) with colormap. The color palette used for demonstration purposes is the "Categorical10" given as a tuple 3 of its colors in their respective hexadecimal values.

[1] To avoid any code repetitions, we continue from the complete code provided at either Section 2.2 or Section 2.3.6

[2] To learn more about the available palettes, visit Bokeh Documentation



(a) (b)

Figure 4.1: Data Colorization (a) before; and (b) after Categorical Color Mapping (Categorical Handler: "mmsi").

4.1.1 A few notes towards Categorical Data Colorization

Due to limitations from the Bokeh library, the categories must be provided in String format. In the previous example, the column "mmsi" that contains the vessels' identifier is parsed as a String prior to the colormap creation.

The (newly created) colormap is not automatically integrated. In order to integrate it to the canvas, it must be passed to the glyph/polygon/line either:

- via the "color" parameter of the add_glyph (or add_polygon, add_line method, respectively); or
- via the renderer.glyph.fill_color (and renderer.glyph.line_color, should the user want to change its value as well) attribute of our instance

Finally, in order to create the categories' legend (such as the one at Figure4.1b) during the glyph creation instead of the "legend_label" parameter, the "legend_group" parameter must be used, as suggested by the previous code block.

4.2 Numerical Data Colorization

Given a properly instantiated *ST_Visions* instance, we demonstrate our class' capability for numerical color mapping. In this section, we colorize the geometry data of our dataset, via the "add_numerical_colormap" method[1].

The numerical colormap is created via a user-defined numerical handler (i.e. a column from the input DataFrame). Creating and incorporating it can be done using a single line of code, as the following code block suggests:

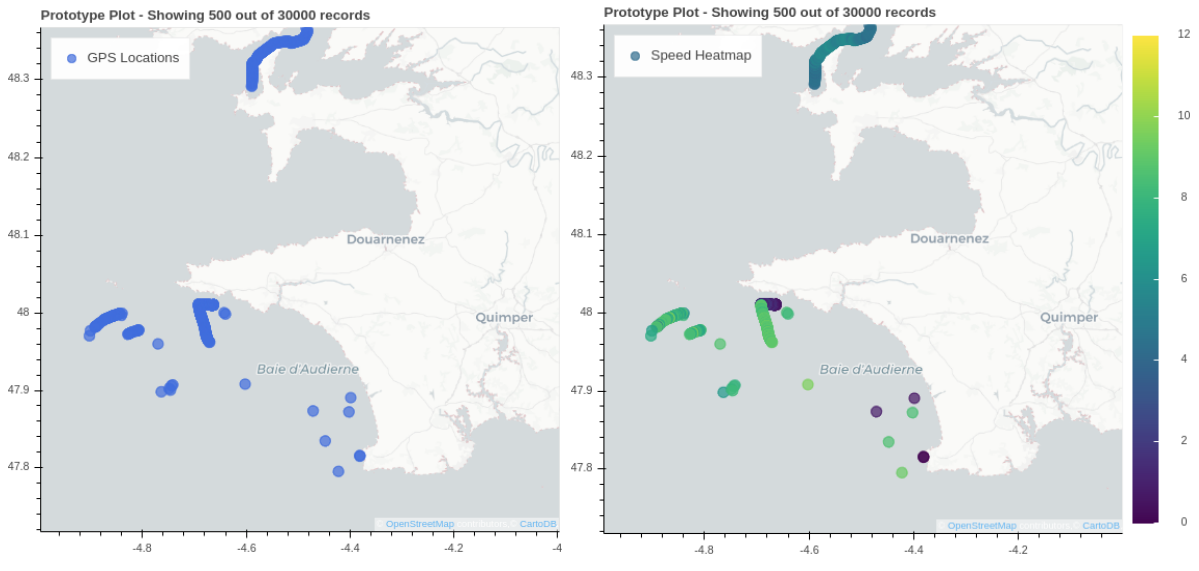
```
1 st_viz = viz.st_visualizer(limit=500)
2 st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
3 st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540)
4 ...
5 st_viz.add_numerical_colormap('Viridis256', 'speed', colorbar=True, cb_orientation='vertical',
      cb_location='right', label_standoff=12, border_line_color=None, location=(0,0))
6 st_viz.add_glyph(glyph_type='circle', size=10, color=st_viz.cmap, alpha=0.8, fill_alpha=0.7,
      muted_alpha=0, legend_label=f'GPS Locations (Speed Heatmap)')
```

```
...
st_viz.show_figures(notebook=False)
```

The aforementioned method supports a wide spectrum of parameters, namely:

- palette: The color palette which will be used for mapping the numeric ranges to their respective colors. The palette can be defined, either as a String that contains its name[3], or as a tuple that contain the hexadecimal value of each color. The latter option can be useful in cases which the default color palettes are not enough to properly visualize the loaded dataset.
- numeric_name: The column name of the given DataFrame that contains the numerical information.
- colorbar: Choose to either draw a colorbar (*True*), or not (*False*) (default: *True*)
- cb_orientation: The orientation of the colorbar (default value: "vertical"; allowed values: "vertical", "horizontal")
- cb_location: The location of the colorbar related to the canvas (default value: "right"; allowed values: "above", "below", "left", "right")
- label_standoff: The distance (in pixels) to separate the tick labels from the colorbar (default: 12).
- border_line_color: The border color of the colorbar (default: *None*)
- location: The location (anchor) of the colorbar, relative to the canvas (default: *(0,0)*)

In addition to the above arguments, the user can specify even more (related to the colormap creation) via the **kwargs parameter. Figure4.2 illustrates the given dataset (a) without; and (b) with colormap. The color palette used for demonstration purposes is "Viridis256".



(a) (b)

Figure 4.2: Data Colorization (a) before; and (b) after Numerical Color Mapping (Numerical Handler: "speed").

[3]To learn more about the available palettes, visit Bokeh Documentation

4.2.1 A few notes towards Numerical Data Colorization

Due to limitations from the Bokeh library, the categories must be provided in numeric (Integer, Float, etc.) format. In the previous example, the column "speed" that contains the vessels' momentary speed is parsed as a Float prior to the colormap creation.

The (newly created) colormap is not automatically integrated. In order to integrate it to the canvas, it must be passed to the glyph/polygon/line either:

- via the "color" parameter of the add_glyph (or add_polygon, add_line method, respectively); or
- via the renderer.glyph.fill_color (and renderer.glyph.line_color, if the user want to change its value as well) attribute of our instance

4.3 Complete Code

In this section we provide the complete code for each use-case of the previous chapters. The codes that follow, have been executed within a (local) Jupyter Notebook cell.

4.3.1 Categorical Data Colorization

```
import st_visions.st_visualizer as viz

st_viz = viz.st_visualizer(limit=500)

st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)
st_viz.data.loc[:, 'mmsi'] = st_viz.data.mmsi.astype(str)

st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_width=540,
    plot_height=540, tools="pan,box_zoom,lasso_select,wheel_zoom,previewsave,reset")
st_viz.add_map_tile('CARTODBPOSITRON')

st_viz.add_categorical_colormap(palette=bokeh_palettes.Category10_3, categorical_name='mmsi')
circ = st_viz.add_glyph(glyph_type='circle', size=10, color=st_viz.cmap, alpha=0.8, fill_alpha
    =0.7, muted_alpha=0, legend_group=f'mmsi')

tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course
    over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon
    , @lat)')]
st_viz.add_hover_tooltips(tooltips)
st_viz.add_lasso_select()

st_viz.figure.legend.location = "top_left"
st_viz.figure.legend.click_policy = "mute"
st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)

st_viz.show_figures(notebook=True)
```

4.3.2 Numerical Data Colorization

```
import st_visions.st_visualizer as viz

st_viz = viz.st_visualizer(limit=500)
st_viz.get_data_csv(filepath='./data/csv/ais_brest_2015-2016.csv', nrows=30000)

st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540,
    tools="pan,box_zoom,lasso_select,wheel_zoom,previewsave,reset")
st_viz.add_map_tile('CARTODBPOSITRON')

st_viz.add_numerical_colormap('Viridis256', 'speed', colorbar=True, cb_orientation='vertical',
     cb_location='right', label_standoff=12, border_line_color=None, location=(0,0))
```

```
10          circ = st_viz.add_glyph(glyph_type='circle', size=10, color=st_viz.cmap, alpha=0.8, fill_alpha
                =0.7, muted_alpha=0, legend_label=f'Speed Heatmap')
11
12          tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@ts'), ('Speed (knots)','@speed'),('Course
                over Ground (degrees)','@course'), ('Heading (degrees)','@heading'), ('Coordinates','(@lon
                , @lat)')]
13          st_viz.add_hover_tooltips(tooltips)
14          st_viz.add_lasso_select()
15
16          st_viz.figure.legend.location = "top_left"
17          st_viz.figure.legend.click_policy = "mute"
18          st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(st_visualizer.WheelZoomTool)
19
20          st_viz.show_figures(notebook=True)
```

Chapter 5

Advanced Use-Cases

In this section, we utilize the power provided by Bokeh combined with the abstraction provided by ST_Visions, in complex, yet everyday use-cases ranging from choropleth maps to online dashboards.

5.1 Choropleth Maps

Choropleth maps are one of the most common visualization types in the field of Mobility Data Analytics (MDA). They provide an easy way to visualize how a measurement varies across a geographic area or show the level of variability within a region. A heat map or isarithmic map is similar but does not use a priori geographic areas.

They are the most common type of thematic map because published statistical data (from government or other sources) is generally aggregated into well-known geographic units, such as countries, states, provinces, and counties, and thus they are relatively easy to create using GIS, spreadsheets, or other software tools[6].

5.1.1 Simple Choropleth Map

In *ST_Visions*, creating a choropleth map is quite easy. More specifically, given a dataset consisted of (Multi)Polygons, along with their respective population, the choropleth map can be created by:

- Loading the Data (e.g., by using the set_data method);
- Create a numeric colormap based on the polygons' respective population; and
- Add the polygons to the CDS, along with the aforementioned colormap.

The below code block loads the first 50,000 rows from the GeoLife dataset, creates the spatial grid based on the dataset's spatial coverage, populates the choropleth map, and finally makes use of *ST_Visions* to visualize it, with Figure 5.1 illustrating the resulted plot.

```
1     # Importing Libraries
2     import pandas as pd
3
4     import st_visions.st_visualizer as viz
5     import st_visions.express as viz_express
6     import st_visions.geom_helper as viz_helper
7     import st_visions.callbacks as viz_callbacks
8
9
10    # Loading GeoLife Dataset
11    gdf = pd.read_csv('./data/csv/geolife_trips_cleaned_v2_china_subset.csv', nrows=50000)
12    gdf = viz_helper.getGeoDataFrame_v2(gdf, crs='epsg:4326')
13
14
```

```
15      # Creating Choropleth (Grid) Geometry
16      bbox = np.array(gdf.total_bounds)
17
18      p1 = shapely.geometry.Point(bbox[0], bbox[3])
19      p2 = shapely.geometry.Point(bbox[2], bbox[3])
20      p3 = shapely.geometry.Point(bbox[2], bbox[1])
21      p4 = shapely.geometry.Point(bbox[0], bbox[1])
22
23      np1 = (p1.coords.xy[0][0], p1.coords.xy[1][0])
24      np2 = (p2.coords.xy[0][0], p2.coords.xy[1][0])
25      np3 = (p3.coords.xy[0][0], p3.coords.xy[1][0])
26      np4 = (p4.coords.xy[0][0], p4.coords.xy[1][0])
27
28      spatial_coverage = gpd.GeoDataFrame(gpd.GeoSeries(shapely.geometry.Polygon([np1, np2, np3, np4])),
            columns=['geom'], geometry='geom', crs='epsg:4326')
29      spatial_coverage_cut = viz_helper.quadrat_cut_geometry(spatial_coverage, 0.7)
30      spatial_coverage_cut = gpd.GeoDataFrame(np.array(list(spatial_coverage_cut)).reshape(-1,1),
            geometry=0, crs='epsg:4326')
31
32
33      # Classifying Area Proximity (i.e., Populate the Choropleth Map)
34      cnt = viz_helper.classify_area_proximity(gdf.copy(), spatial_coverage_cut, compensate=True,
            verbose=True).area_id.value_counts()
35
36      spatial_coverage_cut.loc[cnt.index, 'count'] = cnt.values
37      spatial_coverage_cut.rename({0:'geom'}, axis=1, inplace=True)
38      spatial_coverage_cut.set_geometry('geom', inplace=True)
39
40
41      # Visualize the Choropleth Map
42      st_viz = viz.st_visualizer(limit=len(spatial_coverage_cut))
43      st_viz.set_data(spatial_coverage_cut.dropna())
44
45      st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540, tools="
            pan,box_zoom,lasso_select,wheel_zoom,previewsave,reset")
46      st_viz.add_map_tile('CARTODBPOSITRON')
47
48      st_viz.add_numerical_colormap('Viridis256', 'count', colorbar=True, cb_orientation='vertical',
            cb_location='right', label_standoff=12, border_line_color=None, location=(0,0))
49      st_viz.add_polygon(fill_color=st_viz.cmap, line_color=st_viz.cmap, fill_alpha=0.6, muted_alpha=0,
            legend_label=f'GPS Locations (Choropleth Map)')
50
51      st_viz.figure.legend.location = "top_left"
52      st_viz.figure.legend.click_policy = "mute"
53      st_viz.figure.toolbar.active_scroll = st_viz.figure.select_one(bkhm.WheelZoomTool)
54
55      st_viz.show_figures(notebook=False)
```

5.1.2 Interactive Choropleth Map

While the (simple) choropleth map, may cover the majority of the use-cases, we can go one step further and use the filters of Section 3 in order to create an interactive choropleth map. To achieve that however, two *ST_Visions* instances must be created; one for the choropleth plot (main instance); and another for filtering and maintaining the original data (secondary instance).

Given a baseline geometry dataset (i.e. a two-column dataframe; one column for the polygon geometries, and another for their respective numeric attribute) the main instance will (mainly) be consisted of the CDS for visualzing the polygon geometries on the map, as well as a numerical colormap, as suggested by the following code block[1]:

[1]To avoid unnecessary code repetition, we ommited lines 1–37 from the code of Section 5.1.1.

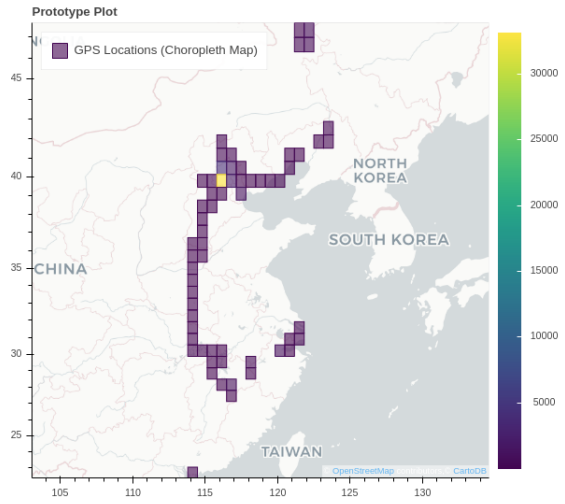


Figure 5.1: Creating a Simple Choropleth Map – a Demonstration using the GeoLife Dataset.

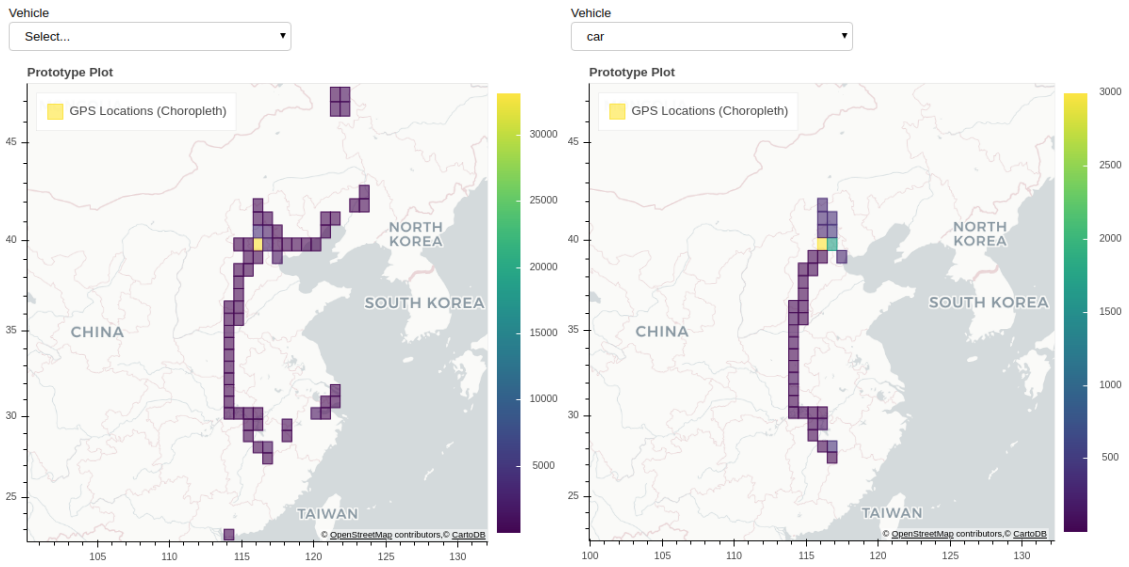
```
    st_viz = viz.st_visualizer(limit=len(spatial_coverage_cut))
    st_viz.set_data(spatial_coverage_cut)

    st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540, tools="
        pan,box_zoom,lasso_select,wheel_zoom,previewsave,reset")
    st_viz.add_map_tile('CARTODBPOSITRON')

    st_viz.add_numerical_colormap('Viridis256', 'count', colorbar=True, cb_orientation='vertical',
        cb_location='right', label_standoff=12, border_line_color=None, location=(0,0), nan_color=
        bokeh_colors.RGB(1,1,1,0))
    st_viz.add_polygon(fill_color=st_viz.cmap, line_color=st_viz.cmap, fill_alpha=0.6, muted_alpha=0,
        legend_label=f'GPS Locations (Choropleth)')
```

The next instance (that will contain the raw GPS points) will be linked to the main instance by sharing the latter's Canvas (i.e. Bokeh Figure — bokeh.plotting.figure). But in order to dynamically change the numeric attribute of the choropleth map instance, a custom callback must be created. Thus we inherit the BokehFilters abstract class from "callbacks.py" module, and implement the abstract method "callback" as well as change the "callback_prepate_data", in order to account for preparing the numeric attribute of the choropleth map. The following code block creates the aforementioned ST_Visions instance and Callback class, and uses it for a categorical filter based on the vehicle type.

```
    data_points = viz.st_visualizer(limit=len(gdf))
    data_points.set_data(gdf)
    data_points.set_figure(st_viz.figure)


    categorical_name='label'

    class Callback(callbacks.BokehFilters):
        def __init__(self, vsn_instance, widget):
            super().__init__(vsn_instance, widget)


        def callback_prepare_data(self, new_pts, ready_for_output):
            self.vsn_instance.canvas_data = new_pts

            if ready_for_output:
                cnt = viz_helper.classify_area_proximity(self.vsn_instance.canvas_data, st_viz.data,
                    compensate=True, verbose=True).area_id.value_counts()
                st_viz.canvas_data = st_viz.data.loc[cnt.index].copy()
                st_viz.canvas_data.loc[:, 'count'] = cnt.values

```

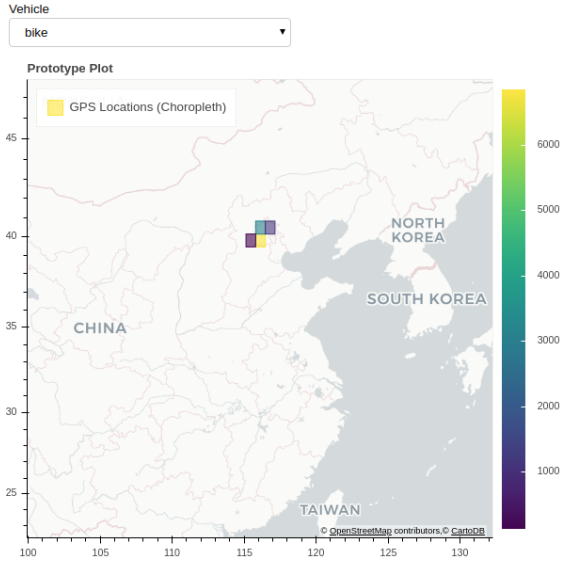
```
                st_viz.canvas_data = st_viz.prepare_data(st_viz.canvas_data)

                low, high = st_viz.canvas_data[st_viz.cmap['field']].agg([np.min, np.max])
                st_viz.cmap['transform'].low = 0 if low == high else low
                st_viz.cmap['transform'].high = high

                st_viz.source.data = st_viz.canvas_data.drop(st_viz.canvas_data.geometry.name, axis=1)
                    .to_dict(orient="list")

                # print ('Releasing Lock...')
                st_viz.canvas_data = None
                self.vsn_instance.canvas_data = None
                self.vsn_instance.aquire_canvas_data = None


        def callback(self, attr, old, new):
            self.callback_filter_data()

            cat_value = self.widget.value
            new_pts = self.get_data()

            # print (cat_value, categorical_name)
            if cat_value:
                new_pts = new_pts.loc[new_pts[categorical_name] == cat_value].copy()

            self.callback_prepare_data(new_pts, self.widget.id==self.vsn_instance.aquire_canvas_data)


    data_points.add_categorical_filter(title='Vehicle', categorical_name=categorical_name,
        height_policy='min', callback_class=Callback)
```

Finally, the below code block renders the interactive choropleth map, with its results illustrated at Figure 5.2.



(a) (b)

Figure 5.2: Interactve Choropleth Map on (a) All; and (b) “Car” vehicle types.



(c)

Figure 5.2: Interactve Choropleth Map on (c) “Bike” vehicle types. (cont.)

5.2 Visualizing Multiple Datasets at the same Canvas

In order to visualize more datasets at the same canvas, we need as many *ST_Visions* instances as the datasets we need to render, because each instance will be loaded with its respective (Geo)DataFrame. Moreover, in order for all the datasets to be visualized at the same Canvas, we need to pass the “figure” attribute of the first *ST_Visions* instance to all other instances, creating a linked chain of visualizers, of sort.

The below example sheds more light to the previous discussion. For demonstration purposes we use the Brest dataset in 3 different versions, namely:

- Raw;
- Linear Interpolated; and
- Linear Extrapolated;

The below code, in a nutshell, loads the 4 “different” datasets[2], creates the respective *ST_Visions* instance and renders them to the same canvas[3], with the result illustrated at Figure 5.3. Notice how the created canvas (“st_viz.figure” attribute) is passed along the other instances, in order to link them.

```
1  import pandas as pd
2
3  import st_visions.st_visualizer as viz
4  import st_visions.express as viz_express
5  import st_visions.geom_helper as viz_helper
6  import st_visions.callbacks as viz_callbacks
7
8
9  df = pd.read_csv('brest_raw.csv')
10 df_inter = pd.read_csv('brest_interp_linear.csv')
11 df_extra = pd.read_csv('brest_extrap_linear.csv')
12
13
```

[2]The preprocessing as well as the interpolation part is outside the scope of this report

[3]While filters and colormaps can be used normally in each instance without any perturbation to the loaded data of the other instances, in order to keep the example as simple as possible we ommited this step.

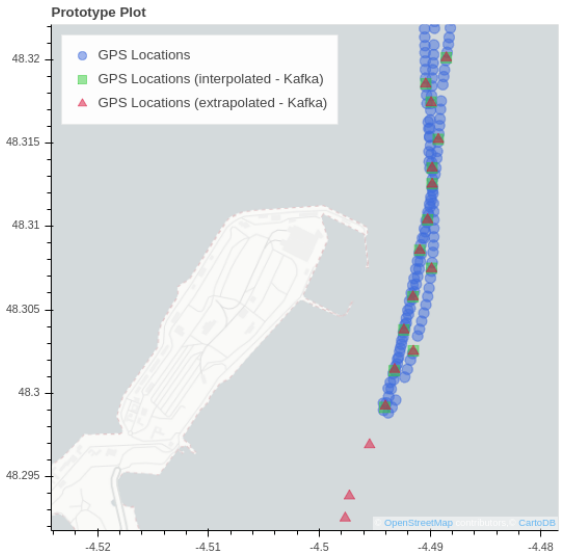


Figure 5.3: Visualizing Multiple Datasets at the same Canvas. A demonstration using the Brest Dataset.

```
14
15 # Create the First (and Main) ST\_Visions Instance
16 st_viz = st_visualizer(limit=len(df))
17 st_viz.set_data(df)
18
19 st_viz.create_canvas(title=f'Prototype Plot', sizing_mode='scale_width', plot_height=540, plot_width
       =1000, tools="pan,box_zoom,lasso_select,wheel_zoom,previewsave,reset")
20 st_viz.add_map_tile('CARTODBPOSITRON')
21
22 st_viz.add_glyph(glyph_type='circle', size=10, color='royalblue', alpha=0.5, fill_alpha=0.5,
       muted_alpha=0, legend_label=f'GPS Locations')
23 st_viz.add_hover_tooltips(tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@datetime'), ('Speed (knots
       )','@speed'), ('Coordinates','(@lon, @lat)'), ('Trajectory/Trip ID', '@traj_id/@trip_id')],
       renderers=st_viz.renderers)
24
25
26
27 # Create the Second ST\_Visions Instance
28 st_viz2 = st_visualizer(limit=len(df_inter))
29 st_viz2.set_data(df_inter)
30
31 # Link the Canvas of the Main Instance to the Current Instance
32 st_viz2.set_figure(st_viz.figure)
33 st_viz2.create_source()
34
35 st_viz2.add_glyph(glyph_type='square', size=10, color='limegreen', alpha=0.5, fill_alpha=0.5,
       muted_alpha=0, legend_label=f'GPS Locations (interpolated - Kafka)')
36 st_viz2.add_hover_tooltips(tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@datetime'), ('Speed (
       knots)','@speed'), ('Coordinates','(@lon, @lat)'), ('Trajectory/Trip ID', '@traj_id/@trip_id')],
       renderers=st_viz2.renderers)
37
38
39
40 # Create the Final ST\_Visions Instance
41 st_viz3 = st_visualizer(limit=len(df_extra))
42 st_viz3.set_data(df_extra)
43
44 # Link the Canvas of the Main Instance to the Current Instance
45 st_viz3.set_figure(st_viz.figure)
```

```
st_viz3.create_source()

st_viz3.add_glyph(glyph_type='square', size=10, color='limegreen', alpha=0.5, fill_alpha=0.5,
    muted_alpha=0, legend_label=f'GPS Locations (interpolated - Kafka)')
st_viz3.add_hover_tooltips(tooltips = [('Vessel ID','@mmsi'), ('Timestamp','@datetime'), ('Speed (
    knots)','@speed'), ('Coordinates','(@lon, @lat)'), ('Trajectory/Trip ID', '@traj_id/@trip_id')],
    renderers=st_viz3.renderers)


# Customizing Legend and Click Policy
st_viz3.figure.legend.location = "top_left"
st_viz3.figure.legend.click_policy = "mute"
st_viz3.figure.toolbar.active_scroll = st_viz3.figure.select_one(bkhm.WheelZoomTool)

# Render Canvas
st_viz3.show_figures(notebook=True)
```

5.3 Visualizing Multiple Datasets in a Grid

While in the previous datasets we used multiple *ST_Visions* instances with a shared canvas, in this use-case we will create separate instances with their own respective canvas, and use the "'show_figures"' method in order to visualize them into a grid.

For demonstration purposes we will use again the Brest Dataset in two versions (i.e., raw, linear interpolated). The code below summarizes the previous discussion, with the results illustrated at Figure 5.4

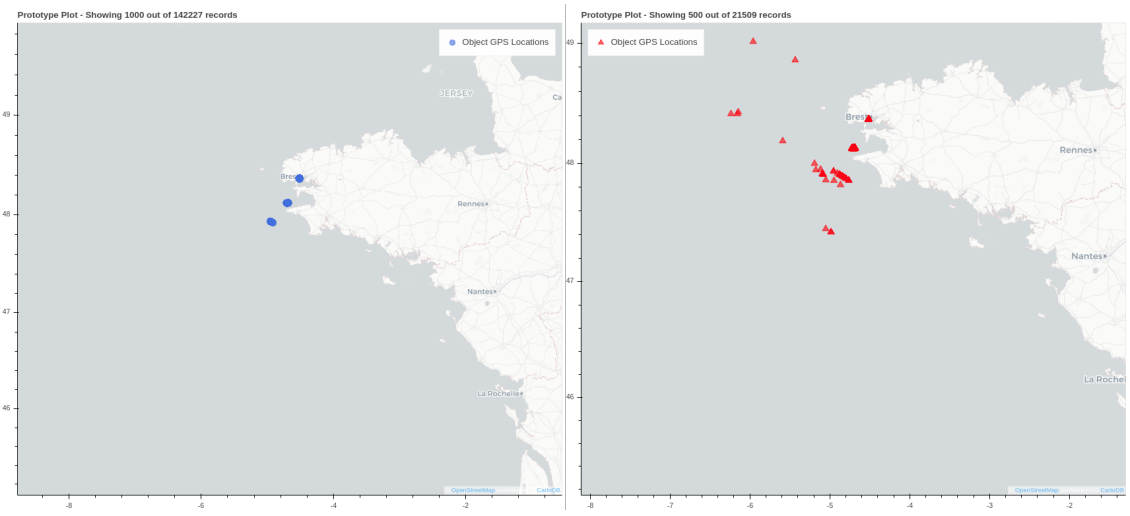


Figure 5.4: Visualizing Multiple Datasets at the same Canvas. A demonstration using the Brest Dataset.

```
import pandas as pd

import st_visions.st_visualizer as viz
import st_visions.express as viz_express
import st_visions.geom_helper as viz_helper
import st_visions.callbacks as viz_callbacks


df = pd.read_csv('brest_raw.csv')
df.loc[:, 'datetime'] = pd.to_datetime(df.ts, unit='s')

df_inter = pd.read_csv('brest_interp_linear.csv')
```

```
df_inter.loc[:, 'datetime'] = pd.to_datetime(df_inter.ts, unit='s')


# Create the First (and Main) ST_Visions Instance
st_viz = st_visualizer(limit=1000)
st_viz.set_data(df)

tooltips = [('MMSI','@mmsi'), ('Datetime','@datetime'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz, tools=['lasso_select'], tooltips=tooltips)


# Create the Second ST_Visions Instance
st_viz2 = st_visualizer(limit=500)
st_viz2.set_data(df_algn_inter)

tooltips = [('MMSI','@mmsi'), ('Datetime','@datetime'), ('Coordinates','(@lon, @lat)')]
viz_express.plot_points_on_map(st_viz2, tools=['lasso_select'], tooltips=tooltips, glyph_type='
    triangle', color='red')


# Customizing Legend and Click Policy for Second ST_Visions Instance
st_viz2.figure.legend.location = "top_left"
st_viz2.figure.legend.click_policy = "mute"
st_viz2.figure.toolbar.active_scroll = st_viz2.figure.select_one(bkhm.WheelZoomTool)

# Render Canvas
st_viz2.show_figures([[st_viz.figure, st_viz2.figure]], notebook=True, merge_tools=True,
    toolbar_location='right')
```

Chapter 6

Conclusions & Future Steps

In this report we proposed a new visualization library called *ST_Visions* that facilitates the interactive visualization of spatio-temporal data, and discussed its technical aspects. More specifically, by using code examples from real-life use-cases, we described the inner workings of the class' methods as well as its attributes, and provided a solid foundation for both novice and advanced python programmers alike. In the future, we aim to futher extend the functionality of the aforementioned library by adding generic methods for the "advanced" use-cases (c.f. Chapter 5), as well as integrating more real-life scenarios, such as, creating online web applications (e.g. dashboards) that will harness more of the power of Bokeh and further demonstrate the simplicity of *ST_Visions*.

Acknowledgement

Research funded by the 2018 National Funds Programme of the General Secretariat of Research and Technology (GSRT), Greece.

Bibliography

[1] Bokeh Development Team. 2018. *Bokeh: Python library for interactive visualization*. `https://bokeh.pydata.org/en/latest/`.

[2] Cyril Ray, Richard Dreo, Elena Camossi, Anne-Laure Jousselme, and Clément Iphar. 2019. Heterogeneous integrated dataset for maritime intelligence, surveillance, and reconnaissance. *Data in Brief*.

[3] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. 2008. Understanding mobility based on GPS data. In *UbiComp*. Volume 344. ACM, 312–321.

[4] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. 2009. Mining interesting locations and travel sequences from GPS trajectories. In *WWW*. ACM, 791–800.

[5] Yu Zheng, Xing Xie, and Wei-Ying Ma. 2010. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33, 2, 32–39.

[6] Wikipedia contributors. 2020. Choropleth map — Wikipedia, the free encyclopedia. [Online; accessed 07-May-2020]. (2020). `https://en.wikipedia.org/wiki/Choropleth_map`.