Part III
Mobility Analytics

Chapter 9
Mobility Data Cleaning

In Chap. 1 we have seen that data cleaning and preprocessing are crucial tasks in the data science process to obtain reliable results in the data analysis tasks. In this chapter, we study techniques and methodologies for cleaning mobility data, which requires specific tasks not present in standard data science applications.

In Sect. 9.1 we classify the kinds of mobility data that usually require cleaning tasks. These include static attributes, voyage-related attributes, temporal attributes, and spatiotemporal trajectories. We continue in Sect. 9.2 with basic cleaning tasks over the AIS dataset. In Sect. 9.3 we address cleaning of static attributes, which describe the moving object as a whole, and continue in Sect 9.4 with the cleaning of voyage-related data, which are attributes that remain static during a voyage. In Sect. 9.5 we delve into the cleaning of temporal data, namely, data that change during a trip.

The remaining of the chapter addresses the cleaning of spatiotemporal trajectories of moving objects, as reported by location-tracking devices such as GPS. These devices can introduce errors, typically within tens of meters, potentially leading to incorrect analysis, such as overestimating speed. Data cleaning is essential to detect and correct these errors. Here we consider two types of movement. **Free-space movement**, such as a ship in the sea, poses challenges for cleaning due to the lack of predefined routes. **Kalman filter** smoothing is a typical method to address this problem. In Sect. 9.6 we explain the theory behind Kalman filter and focus on the practical implementation over the AIS dataset using the Stone Soup tracking library. Finally, in Sec. 9.7 we address **network-constrained movement**, such as the movement of cars on roads. Here, sensor errors can cause the reported points to be incorrectly placed outside the network. This problem is typically addressed using **map matching**, which in turn, use the **hidden Markov model** (HMM) as a tool to align trajectory points with the underlying network. We illustrate the implementation of HMM map-matching using a Valhalla server.

9.1 Cleaning Mobility Data

In mobility data we can identify four categories of attributes that can be addressed by cleaning tasks:

1. Static attributes, which describe the moving object. Examples are the vehicle license plate number, the model, and the registration year.
2. Voyage-related attributes, which describe individual trips. Examples are the trip destination, the trip start time, and the cargo weight.
3. Temporal attributes, which change during the trip. Examples are the gear, the speed, and the course.
4. Spatiotemporal trajectories, which are captured by location-tracking devices, such as a GPS.

Note that a trip's beginning and end are rarely captured in the input data. Further, there is no agreed definition of a trip, since this highly depends on the application scenario. For some scenarios this is relatively easy. This is for example the case of a taxi trip, which can be captured via passenger collection and drop-off, and a flight trip between airports. Opposite to these cases, the boundaries of a car trip are less clear. During one day, a person can drive from home to work, drop the children to the school, buy a coffee on the way, and finally return back home. The sequences of events that constitute a trip are application dependent.

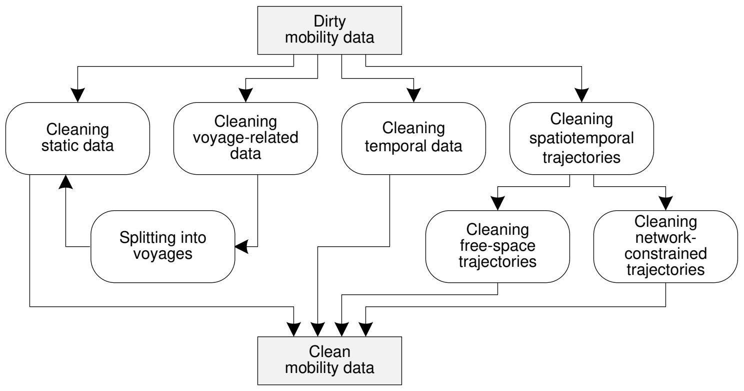

Dirty mobility data
Cleaning static data
Cleaning voyage-related data
Cleaning temporal data
Cleaning spatiotemporal trajectories
Splitting into voyages
Cleaning free-space trajectories
Cleaning network-constrained trajectories
Clean mobility data

Fig. 9.1 The process of cleaning mobility data.

Figure 9.1 describes the process for mobility data cleaning, which involves the four categories described above. Cleaning *static* data involves applying standard methods for detecting or correcting errors in constant attributes. As shown in the figure, cleaning *voyage-related* data is similar to cleaning static data after data are preprocessed and split into voyages. Cleaning *temporal* attributes involves methods such as those used for time series.

Cleaning *spatiotemporal trajectories* is specific to mobility data. These trajectories are collected using location-tracking sensors, GPS being the most common one. These sensors often introduce errors in the observed locations, typically in the range of tens of meters. Depending on the analytical task, these errors could lead to incorrect insights. For instance, a GPS point that is far away from the actual location might erroneously suggest that the object was moving at a speed much higher than the actual one. Therefore, it is essential to perform data cleaning to detect and rectify these errors.

Two main kinds of movement are considered when cleaning spatiotemporal trajectories. **Free-space movement** occurs when the movement of an object is not being restricted to follow a predetermined path, such as a ship in the sea. Alternatively, **network-constrained movement** occurs when the movement of an object is restricted by an underlying transportation network, such as car on a road network. In the first case, **Kalman filter** is one of the most often used techniques to smooth the noise and reduce measurement errors. This technique is used to enhance the accuracy of the estimation of position and velocity, by incorporating both, measurements from sensors and predictions from a dynamic model of the object. This results in a smoothed trajectory that more accurately reflects the true motion of the object. This is explained in Sect. 9.6. To clean network-constrained trajectories, **map matching** is a well-established technique that ensures that all trajectory points are aligned with the underlying network after cleaning. This is explained in Sect. 9.7.

9.2 AIS Data Cleaning

In this chapter, we study mobility data cleaning using mostly AIS data. We will use the same dataset as in Sect. 5.2.1, which corresponds to vessel tracks registered on March 1st, 2024 by the Danish Maritime Authority. We assume that the AIS data has already been loaded into a PostgreSQL database, that is, the table AISInput has been created as follows:

```
CREATE TABLE AISInput(T timestamp, TypeOfMobile varchar(100), MMSI integer,
  Latitude float, Longitude float, NavigationalStatus varchar(100), ROT float,
  SOG float, COG float, Heading integer, IMO varchar(100), CallSign varchar(100),
  Name varchar(100), ShipType varchar(100), CargoType varchar(100), Width float,
  Length float, TypeOfPositionFixingDevice varchar(100), Draught float,
  Destination varchar(100), ETA varchar(100), DataSourceType varchar(100),
  SizeA float, SizeB float, SizeC float, SizeD float, Geom geometry(Point, 4326));
```

and then populated using the COPY command.

When dealing with raw data files, such as CSV and JSON files, various problems usually arise due to the limitations of the file formats. For example, CSV files do not explicitly support data types, causing all data to be interpreted either as numeric or string values. Consequently, data types that require explicit formats, such as timestamps and geospatial coordinates, are not directly supported and need to be formatted separately. To ensure that data remain portable across various computing platforms, data standards often require information to be stored in a format that is independent of any specific software or hardware. This can be seen in standards such as the well-known text (WKT) or well-known binary (WKB) formats for geospatial data. Finally, sometimes data generation or collection produce nonstandard entries, such as text messages stored in place of expected numerical values, which require data transformation tasks.

Mobility data requires specific transformations. This is the case of timestamps, which typically arrive in a standard format like ISO 8601 or represented as epoch. An *epoch* is a numerical value that represents the number of non-leap seconds that have elapsed since a fixed date, which for Unix systems is 00:00:00 UTC on January 1st, 1970. In modern computing, values are sometimes stored with higher granularity, such as milliseconds or microseconds. As another usual case, geographical data are usually stored as separate longitude and latitude values, which requires converting them into a geographical point that can be processed in a spatial database.

As explained in Sect. 5.2.1, for the AIS dataset we have already dealt with the timestamp transformation in the COPY command. Further, we have also added an attribute Geom that contains the geometry obtained from the Longitude and Latitude attributes. In addition, we must transform the geometry attribute by projecting it to a metric coordinate system to facilitate the calculations in the following sections, as show next.

```
ALTER TABLE AISInput ADD COLUMN GeomProj geometry(Point, 25832);
UPDATE AISInput SET GeomProj = ST_Transform(Geom, 25832);
```

To explore errors and design the data cleaning pipeline for large datasets, we typically work with a sample of it. In this chapter, we will focus on two hours of data obtained as follows:

```
CREATE TABLE AISInputSample AS
SELECT *
FROM AISInput
WHERE EXTRACT(HOUR FROM T) BETWEEN 9 AND 10;
```

In the remainder of this chapter, we use this table to illustrate data cleaning tasks. Once the data cleaning pipeline is implemented and tested on the sample, it can be run over the whole dataset.

9.3 Cleaning Static Data

We start the data cleaning process by addressing the static attributes. We first analyze missing data, which may come in multiply variants, and then we analyze erroneous data, which we illustrate by analyzing the consistency between two possible identifiers for the ships, namely, MMSI and IMO.

9.3.1 Identifying Missing Data

Missing values can occur for various reasons. For example, sensor malfunctioning, connectivity issues, or environmental interference can disrupt data transmission. It is crucial to detect these missing values, since, if not properly addressed, they can lead to biases or inaccuracies. By identifying and addressing missing values, we can improve the quality of the dataset, making it more suitable for modeling and decision-making tasks.

We can obtain an overview of the missing values in the dataset by computing the total number of non-null values for each attribute as follows:

```
SELECT COUNT(*) AS TotalRows, COUNT(T) AS T_NN,
  COUNT(TypeOfMobile) AS TypeOfMobile_NN, COUNT(MMSI) AS MMSI_NN,
  COUNT(Latitude) AS Latitude_NN, COUNT(Longitude) AS Longitude_NN,
  -- remaining attributes
  ...
FROM AISInputSample;
/* TotalRows: 1334271,
   T_NN: 1334271,
   TypeOfMobile_NN: 1334271,
   ...
   ROT_NN: 885998,
   SOG_NN: 1216567,
   COG_NN: 1145613,
   ... */
```

The query above assumes that null values are encoded as NULL in the data. However, this is not always the case, since other placeholders like 'unknown' or 'unknown_value' can be used to express missing values. There are classic strategies that can be used to identify such placeholders. This requires an exploratory approach, involving human tasks, to inspect the data and determine the nature of these placeholders. We describe some of them next.

We first show a set of SQL queries for quickly identifying usual non-null placeholders such as 'none', 'n/a', 'not available', or 'unknown'.

```
SELECT DISTINCT Destination AS UniqueValues
FROM AISInputSample
WHERE Destination ILIKE '%none%' OR Destination ILIKE '%n/a%' OR
  Destination ILIKE '%not available%' OR Destination ILIKE '%unknown%';
/* N/A
   THYBOROEN/AGGER
   UNKNOWN
   Unknown */
```

The second row of the result shows an actual value that was selected while looking for the 'n/a' placeholder. We need do the same with all the other attributes that present possible non-null placeholders.

When dealing with a large number of columns and a potential variety of placeholders, proceeding on an individual basis with attributes may be awkward. Therefore, we rather write an SQL query that identifies the most frequent values of the columns as follows:

```
WITH ColsVals(ColumnName, ColumnValue) AS (
  SELECT 'TypeOfMobile', TypeOfMobile FROM AISInputSample UNION ALL
  SELECT 'NavigationalStatus', NavigationalStatus FROM AISInputSample UNION ALL
  SELECT 'IMO', IMO FROM AISInputSample UNION ALL
  SELECT 'CallSign', CallSign FROM AISInputSample UNION ALL
  SELECT 'Name', Name FROM AISInputSample UNION ALL
  SELECT 'ShipType', ShipType FROM AISInputSample UNION ALL
  SELECT 'CargoType', CargoType FROM AISInputSample UNION ALL
  SELECT 'TypeOfPositionFixingDevice', TypeOfPositionFixingDevice
  FROM AISInputSample UNION ALL
  SELECT 'Destination', Destination FROM AISInputSample UNION ALL
  SELECT 'ETA', ETA FROM AISInputSample UNION ALL
  SELECT 'DataSourceType', DataSourceType FROM AISInputSample )
SELECT ColumnName, ColumnValue, COUNT(*) AS Frequency
FROM ColsVals
WHERE ColumnValue IS NOT NULL
GROUP BY ColumnName, ColumnValue
ORDER BY Frequency DESC;
```

With the results of the above query, we identify the placeholders shown in Table 9.1. Depending on the dataset, we must ensure that placeholders like 'Unknown' are indeed meant to be interpreted as NULL. Sometimes, such terms might be legitimate data values depending on the context. In the results of the above query, there are other situations where it is not clear if the data value should or should not be replaced by NULL.

ColumnName	ColumnValue	Frequency
IMO	'Unknown'	555,622
Destination	'Unknown'	344,362
NavigationalStatus	'Unknown value'	211,223
ShipType	'Undefined'	136,437
CargoType	'No additional information'	133,413
CallSign	'Unknown'	121,852
TypeOfPositionFixingDevice	'Undefined'	87,609
ShipType	'Other'	62,958
Destination	'UNKNOWN'	10,034

Table 9.1 Most often used missing value placeholders in AIS data.

The query above helps to uncover potential data quality issues by identifying frequent entries in the dataset. The assumption here is that commonly recurring values, especially in textual columns, are often used as placeholders for missing data. Clearly, this is not always the case. For instance, data might have columns from a low-cardinality domain, such as the gear of a car, which is an enumeration from 1 to 6 in addition to the reverse gear. In such cases, normal values (e.g., an integer) may be more frequent than the missing values, and the query above might miss such missing values.

The query above includes all varchar and text columns, so that the attributes types in the UNION ALL statements are the same. Numeric and geospatial attributes are typically less likely to have such string-based placeholders but should be checked for nonstandard values if necessary. Therefore, we may need to repeat the above query for numerical attributes, and possibly for other groups of compatible data types.

Based on the placeholders for missing data identified in Table 9.1, we can write UPDATE statements to replace these placeholders with actual NULL values in each specified column of the AISInputSample table as follows:

```
UPDATE AISInputSample SET
  IMO = CASE WHEN IMO = 'Unknown' THEN NULL ELSE IMO END,
  Destination = CASE WHEN Destination ILIKE 'Unknown' THEN NULL ELSE
    Destination END,
  NavigationalStatus = CASE WHEN NavigationalStatus = 'Unknown value' THEN
    NULL ELSE NavigationalStatus END,
  ShipType = CASE WHEN ShipType = 'Undefined' OR ShipType = 'Other' THEN
    NULL ELSE ShipType END,
  CargoType = CASE WHEN CargoType = 'No additional information' THEN
    NULL ELSE CargoType END,
  CallSign = CASE WHEN CallSign = 'Unknown' THEN NULL ELSE CallSign END,
  TypeOfPositionFixingDevice = CASE WHEN TypeOfPositionFixingDevice = 'Undefined'
    THEN NULL ELSE TypeOfPositionFixingDevice END
WHERE IMO = 'Unknown' OR Destination ILIKE 'Unknown' OR
  NavigationalStatus = 'Unknown value' OR ShipType = 'Undefined' OR
  ShipType = 'Other' OR CargoType = 'No additional information' OR
  CallSign = 'Unknown' OR TypeOfPositionFixingDevice = 'Undefined';
```

Note that the query above accounts for the two missing value placeholders for the attributes Destination and ShipType.

After this, when we recompute the NULL statistics, we get larger values than those in Table 9.1. This more accurately reflect the status of the data, since potential missing values have been replaced with actual NULL values. In the next section, we explain how we can impute some of these NULL values.

9.3.2 Consistency of Ship Data

The AIS dataset has two attributes that identify the ships: MMSI (Maritime Mobile Service Identity) and IMO (International Maritime Organization number). We therefore need to verify that each MMSI maps to exactly one IMO and vice versa. The next query validates this relationship.

```
SELECT MMSI, COUNT(DISTINCT IMO) AS UniqueIMOCount
FROM AISInputSample
GROUP BY MMSI
HAVING COUNT(DISTINCT IMO) > 1;
```

The query returns no rows. We next check for duplicates in the IMO values.

```
SELECT IMO, COUNT(DISTINCT MMSI) AS UniqueMMSICount
FROM AISInputSample
GROUP BY IMO
HAVING COUNT(DISTINCT MMSI) <> 1;
-- NULL | 1370
```

The query results show that there are 1,370 different MMSI values that appear in combination with a NULL IMO. To understand the reason for this, we start by examining records that do successfully pair these MMSI numbers with an IMO. If we could identify consistent pairings in the data, we could use these associations to fill in the missing IMO values for records with the same MMSI, enhancing the completeness of the dataset. We create next a table that establishes a mapping of the most frequent IMOs for each MMSI.

```
CREATE TABLE IMOMapping(MMSI, MostFrequentIMO) AS
SELECT MMSI, MODE() WITHIN GROUP (ORDER BY IMO)
FROM AISInputSample
WHERE MMSI IN (
  SELECT DISTINCT MMSI FROM AISInputSample WHERE IMO IS NULL )
GROUP BY MMSI
HAVING MODE() WITHIN GROUP (ORDER BY IMO) IS NOT NULL;
-- SELECT 35
```

We use the MODE function in the WITHIN GROUP (ORDER BY IMO) clause to compute the statistical mode of IMO values for each group of records with the same MMSI. For example, for MMSI 211291170, we obtain that the most frequent value of IMO is 7904906. This can be checked by writing

```
SELECT MMSI, IMO, COUNT(*) AS Freq
FROM AISInputSample
WHERE MMSI = 211291170
GROUP BY MMSI, IMO
ORDER BY Freq DESC;
/* 211291170 | 7904906 | 414
   211291170 | NULL | 11 */
```

We update table AISInputSample using the IMOMapping table as follows.

```
UPDATE AISInputSample a
SET IMO = MostFrequentIMO
FROM IMOMapping m
WHERE a.MMSI = m.MMSI AND a.IMO IS NULL AND
  m.MostFrequentIMO IS NOT NULL;
```

The update sets the IMO to the most frequently occurring IMO, wherever it is NULL, but only for those rows where MostFrequentIMO is not NULL. This ensures that all missing IMO values are systematically filled with the most plausible data, based on observed patterns within the dataset. In this case, we could only impute a small number of missing IMOs, the remaining missing ones cannot be imputed using only the information we have.

The same cleaning process can be repeated for other ship-related attributes such as SizeA to SizeD, which store the measures of the ship with respect to the GPS device. Since these attributes are always the same for the same ship, they should have a one-to-one correspondence with the MMSI. Note that this correspondence is an example of a **functional dependency** between attributes. Recall that a functional dependency is a constraint specifying that the value of a set of columns determines the value of another one, which can help data cleaning, as explained for the missing IMO numbers. Automated tools such as OpenRefine[1] can discover these dependencies and use them for data cleaning. These tools can automatically detect patterns or dependencies in the data, allow users to define rules based on these dependencies, and perform cleaning operations such as normalizing data, correcting anomalies, and filling missing values according to the observed functional dependencies.

9.4 Cleaning Voyage-Related Data

We mentioned before that voyage-related attributes are static per voyage. For instance, in AIS data the Destination does not change for the whole voyage, but it could change for different voyages of the same ship. Therefore, after splitting the data into voyages, we can clean the voyage-related attributes in the same way as for static attributes.

Splitting into voyages is an application-dependent task. The definition of a voyage depends on what we want to analyze afterward, for example, analyze one-way versus two-way voyages, study the driver's versus the passengers' perspectives, defining what is considered to be a stop, and so on. In some cases, certain attributes are relevant for detecting the boundaries of voyages. For example, in AIS data, the Destination attribute can be used for splitting voyages. Accordingly, the cleaning scripts in the previous section can be repeated, yet the grouping should be done as follows:

```sql
CREATE TABLE DestinationMapping(MMSI, MostFrequentDestination) AS
SELECT MMSI, MODE() WITHIN GROUP (ORDER BY Destination)
FROM AISInputSample
WHERE MMSI IN (
  SELECT DISTINCT MMSI
  FROM AISInputSample
  WHERE Destination IS NULL )
GROUP BY MMSI;
UPDATE AISInputSample a
SET Destination = MostFrequentDestination
FROM DestinationMapping m
WHERE a.MMSI = m.MMSI AND a.Destination IS NULL AND
  m.MostFrequentDestination IS NOT NULL;
```

For example, here we can see that the most common destination for MSSI 246218000 is Antwerpen, although many of its tuples contain a NULL value. We can fix this, using the script above, and use this attribute to split trips into voyages, in this case updating 2,087 tuples in the sample.

[1] https://openrefine.org/

9.5 Cleaning Temporal Data

We now focus on the temporal attributes of AIS data, specifically SOG (Speed Over Ground), COG (Course Over Ground), and Heading. These attributes are, basically, **time series** data. Unlike in the previous sections, where we addressed data on a tuple-by-tuple basis, time series data require a different representation. These series are functions that map timestamps to numerical values. This representation helps to better understand the underlying patterns and behaviors in the data. It also enables the use of well-established time series analysis methods for data cleaning, including:

- **Visual exploration**, to ensure that the relationships between SOG, COG, and Heading are plausible given the operational parameters;
- **Time series smoothing**, to reduce noise and smooth the time series by applying mean or median filters;
- **Outlier detection**, to identify and correct or remove statistical anomalies that deviate significantly from typical patterns by using methods such as **Z-score**, **interquartile range** (IQR), or more sophisticated machine learning approaches;
- **Data interpolation**, to fill-in missing values or to correct erroneous values for ensuring continuity and completeness of the time series.

In the following sections we explore the first three of these techniques.

9.5.1 Visual Exploration

We first show how we can visualize analyze samples of the time series data to understand the kinds of data quality issues that we should address. For this, we use interactive visualization tools, namely, Plotly graphs in combination with the dashboard library Dash.[2] The Python script below creates an interactive web-based visualization of time series data that can be used not only for AIS data but also for other scenarios.

```
# Library imports
import dash
from dash import dcc, html, Input, Output
import plotly.graph_objs as go
import pandas as pd
from sqlalchemy import create_engine

# Database connection setup
username = 'your_username'
password = 'your_password'
database = 'your_postgres_database_name'
host = 'your_host'
```

[2] https://github.com/plotly/dash

```
database_url = f'postgresql://{username}:{password}@{host}/{database}'
engine = create_engine(database_url)

# Fetch data for 10 random MMSIs
query = """
SELECT MMSI, T AS Timestamp, SOG, COG, Heading
FROM AISInputSample
WHERE MMSI IN (
  SELECT MMSI
  FROM (SELECT DISTINCT MMSI FROM AISInputSample) AS UniqueMMSIs
  ORDER BY RANDOM() LIMIT 10 )
ORDER BY MMSI, t;
"""
df = pd.read_sql_query(query, engine)
df['timestamp'] = pd.to_datetime(df['timestamp'])
```

We start by querying the database to fetch data for ten randomly selected MMSIs (we use just a small sample to facilitate the reading). These data include timestamps and various navigational parameters like SOG (speed over ground), COG (course over ground), and Heading. The last line converts the DataFrame column into a datetime type.

The following portion of the script initializes a Dash application.[3] Dash uses Flask[4] as the web framework. The app's layout is defined using HTML and Dash components, including a dropdown menu where the user can pick an MMSI value and a graph, for displaying the time series associated with it.

```
# Dash initialization
app = dash.Dash(__name__)
app.layout = html.Div([
  dcc.Dropdown(
    id='mmsi-dropdown',
    options=[{'label': i, 'value': i} for i in df['mmsi'].unique()],
    value=df['mmsi'].unique()[0] ),
    dcc.Graph(id='time-series-plot') ])
```

Since Dash uses callbacks for interactivity, we must define a callback function that updates the graph based on the MMSI selected from the dropdown. It plots time series data for the SOG, COG, and Heading attributes. The visualization is thus interactive in many ways: we can choose an MMSI, and zoom and pan the time series. The corresponding code is shown below.

```
# Callback to update graphs based on selected MMSI
@app.callback(
  Output('time-series-plot', 'figure'),
  [Input('mmsi-dropdown', 'value')] )
def update_graph(selected_mmsi):
  filtered_df = df[df['mmsi'] == selected_mmsi]
  scaled_sog = filtered_df['sog'] * 5
  return {
    'data': [
```

[3] https://plotly.com/dash/

[4] https://flask.palletsprojects.com/

```
        go.Scatter(
          x=filtered_df['timestamp'], y=filtered_df['sog'],
            mode='lines', name='Scaled SOG (x5)' ),
        go.Scatter(
          x=filtered_df['timestamp'], y=filtered_df['cog'],
            mode='lines', name='COG' ),
        go.Scatter(
          x=filtered_df['timestamp'], y=filtered_df['heading'],
            mode='lines', name='Heading' ) ],
      'layout': go.Layout(
        title='Time Series for MMSI: {}'.format(selected_mmsi),
        xaxis_title='Timestamp',
        yaxis_title='Value',
        margin={'l': 40, 'b': 40, 't': 50, 'r': 10},
        hovermode='closest',
        transition={'duration': 500} ) }
```

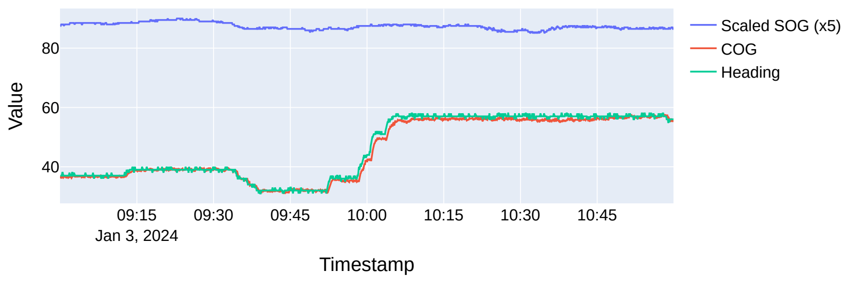
We scale the SOG values multiplying them by five, to transform them into a similar range as the COG and Heading values, thus enhancing their visualization in the same figure.

Finally, we ran the visualization app. It will load into the Jupyter notebook or can be accessed via a Web browser, typically at the address http://127.0.0.1:8050/. The following line in a Jupyter notebook calls the app.

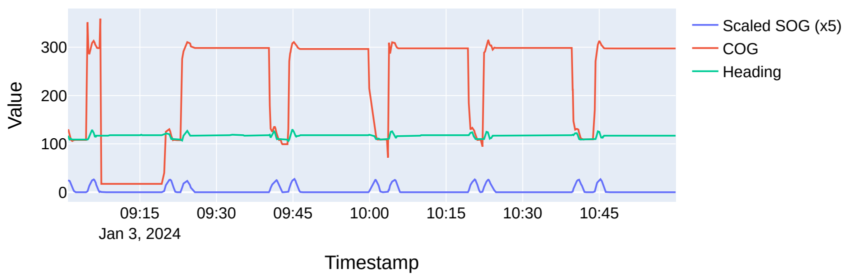
```
if __name__ == '__main__':
  app.run_server(debug=True)
```

We use this visualization to explore the temporal data and to understand the data issues. Figure 9.2 shows some examples that reveal interesting insights and opportunities for data cleaning and analysis:

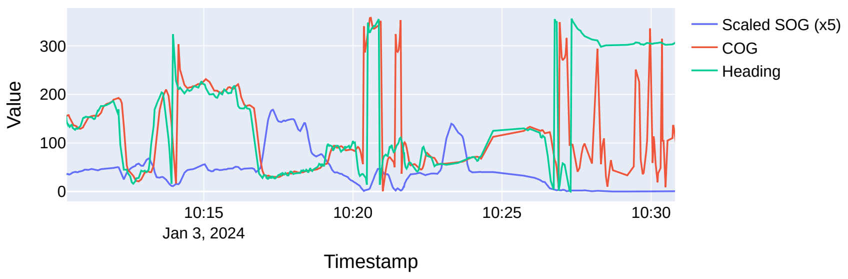
- Noise in signals: All three signals SOG, COG, and Heading have noise. A typical way to reduce this noise consists in applying **time series smoothing**. This is explained in the following section.
- Similarity between COG and Heading: Both COG and Heading seem to behave similarly, which is expected when environmental factors such as wind and current are not too strong. Notice that COG is derived from the ship's GPS system, indicating the actual path over the ground, whereas Heading is determined by the ship's orientation, typically controlled by the captain and measured by a gyrocompass or a magnetic compass.
- Randomness at low SOG: As shown in Figs. 9.2b and 9.2c, the values of both COG and Heading show random fluctuations and deviate from each other when the values of SOG are close to zero. The reason is that even at low speed, GPS sensors report slight movements due to noise around the ship's actual position. Depending on the analysis requirements, various strategies may be applied to solve this, such as setting COG and Heading values to NULL or to a constant when SOG is near zero.
- Zigzag navigation: The Heading and COG signals in Fig. 9.2d show a zigzag trend, and one may wonder whether this is noise. The vessel's type in ShipType states that the ship is actually a tanker. Further investigation



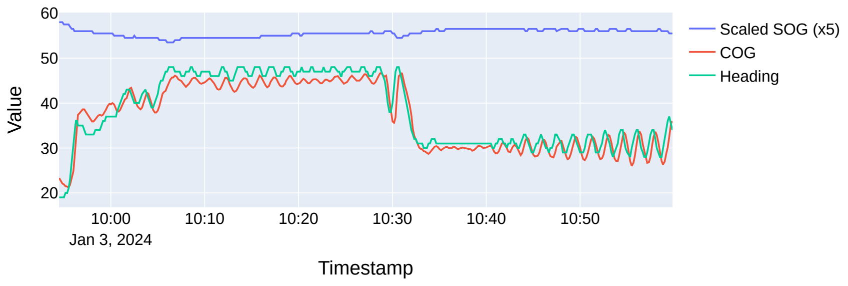
(a) MMSI 311001076



(b) MMSI 219014579



(c) MMSI 219019011



(d) MMSI 246541000

Fig. 9.2 Visualization of the SOG, COG, and Heading attributes of various vessels.

in the maritime navigation domain revealed that tanker ships engage in such maneuvers to maintain stability under adverse wind and wave conditions. This gives evidence that such zigzag is not a data error.
- Repeating patterns in SOG: In Fig. 9.2b, the SOG shows a cyclic pattern of movement followed by stationary periods, roughly every hour. Since the vessel's type is a passenger ferry, this suggests that these patterns likely correspond to scheduled two-way trip along its route.

9.5.2 Time Series Smoothing

By applying smoothing methods to time series, we can improve their interpretability and forecast accuracy, and understand long-term patterns and cyclic behavior. Several methods exist for smoothing time series data, each with its own advantages and applications. We review some of them next.

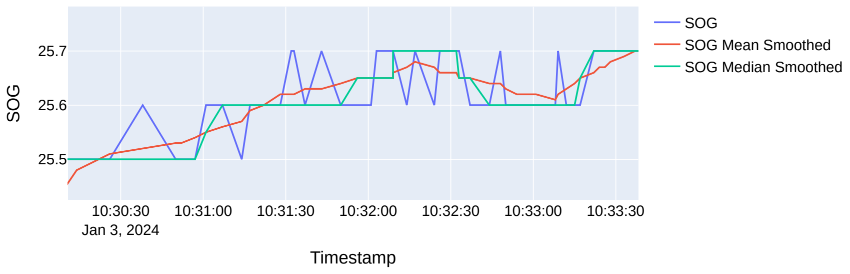
- **Mean smoothing**, also referred to as *moving average*, is the simplest and most commonly used method, where the value at each point is replaced by the average (or median) of values around it within a defined window. It effectively reduces random variation and highlights longer-term trends.
- **Exponential smoothing** assigns exponentially decreasing weights to older observations and is useful for data with trends and seasonality.
- **Locally estimated scatterplot smoothing** (LOESS) adaptively fits local models such as linear or quadratic regression to subsets of the points of the time series. The resulting fits are then combined into a continuous line that represents a smoothed variant of the underlying time series.

We illustrate the first method next. Implementing the moving average (or median) is straightforward, as both are supported as DataFrame window functions in the Pandas[5] library. They can also be easily implemented in SQL using window functions. The following portion of a Python script shows the smoothing of the SOG values using both moving average and moving median:

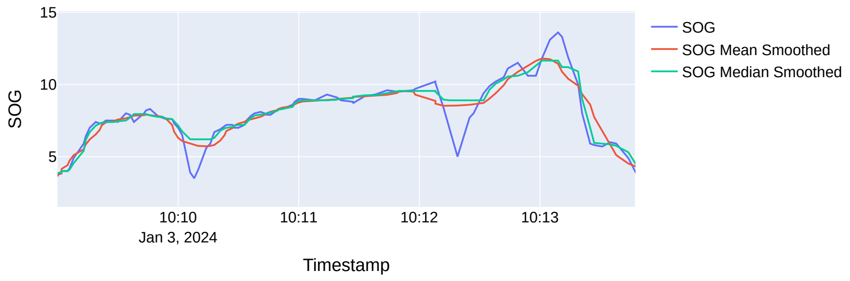
```python
# Define the window size for smoothing
window_size = 10 # Higer values result in a smoother signal
# Apply rolling mean and median
df['sog_mean_smoothed'] = df['sog'].rolling(window=window_size, center=True).mean()
df['sog_median_smoothed'] = df['sog'].rolling(window=window_size,
 center=True).median()
df['cog_mean_smoothed'] = df['cog'].rolling(window=window_size, center=True).mean()
df['cog_median_smoothed'] = df['cog'].rolling(window=window_size,
 center=True).median()
df['heading_mean_smoothed'] = df['heading'].rolling(window=window_size,
 center=True).mean()
df['heading_median_smoothed'] = df['heading'].rolling(window=window_size,
 center=True).median()
```

[5] https://pandas.pydata.org/

Figure 9.3 shows the original signals together with their smoothed versions using moving average and moving median. The moving median, since it is always based on actual data points, tends to adhere more closely to the original values of the time series. In contrast, the moving average tends to weave around the original points and does not necessarily align directly with any specific data point. Moreover, the median is generally more robust to outliers compared to the average, making it a preferable choice in scenarios where outlier cleaning is crucial.



(a) MMSI 21269660



(b) MMSI 219019011

Fig. 9.3 Smoothing of the SOG attribute of two vessels.

Both smoothing techniques require a careful setting of the sliding window size. A larger window may lead to *over-smoothing*, which can smooth away important features of the signal that would not be perceived, while *under-smoothing* might leave too much noise in the data, affecting the clarity and utility of the analysis. Another consideration, which is particularly relevant for the COG and Heading signals, is that they are encoded in angles. Angle values are circular, e.g., 360 is equal to zero. A small actual change in direction can appear as a large jump in numerical terms (e.g., from 359 to 2 degrees). Smoothing may consider a jump between 359 and 2 to be an outlier, and try to smooth it away. To solve this issue, angles should be presented in noncircular format, using for example **quaternions**, which are a generalization of two-dimensional complex numbers to three dimensions.

9.5.3 Outlier Detection

Outliers are data points that deviate significantly from the rest of the data. They can indicate anomalies, errors, or extraordinary events that are essential to identify and understand. We discuss next common types of outliers.

Global outliers are data points that are significantly different from the rest of the data. They can occur due to measurement or data entry errors, or due to abrupt changes in the data, in which they case are not errors but actual events. In time series, these are usually easy to spot as single points where the signal spikes or dips dramatically compared to surrounding data points.

Contextual or *conditional outliers* are data points that are considered outliers within a specific context or condition, but they might not be outliers when considered outside that context. For example, a speed of 200 km/h in AIS data is certainly an outlier for ships, but it is normal for helicopters that put an AIS antenna in a search and rescue (SAR) mission, and that may appear in the data. In time series, the context can be defined by other attributes, such as ShipType, or by time, such as seasons, cycles, or patterns like weekdays versus weekends.

Collective outliers occur when a subset of data points within a time series collectively deviates from the entire pattern. These might not be outliers on their own, but are anomalous when appearing together in sequence. For instance, a few scattered gaps in the AIS track of a vessel can be considered normal. However, a longer gap, such as one hour, may suggest a serious problem such as piracy or illegal fishing.

There is no easy way to differentiate outlier that are data errors from outliers that are interesting events. It would require domain-specific solutions to achieve this. Therefore, our focus in this section is primarily on the detection of outliers in general, and we will explore a technique to identify them in the time series data. The task of deciding how to handle these outliers, that is, whether to discard them as errors or further analyze them as events of interest, is equally important but is out of the scope of this section.

One of the most often used techniques for detecting outliers in data is the *interquartile method*. This method is known to be appropriate for handling noisy data and provides an intuitive approach to outlier detection. In Chap. 4 we studied the box plot visualization technique, which displays the division of the data into quartiles. For example, the first quartile, denoted Q1, represents data such that 25 percent of the values are less than or equal to a particular threshold. The other quartiles are defined analogously. The **interquartile range** (IQR) is the difference between the 75th (Q3) and the 25th (Q1) percentiles of the data. Based on these values, we call outliers the observations that fall below $Q1 - 1.5 * IQR$ or above $Q3 + 1.5 * IQR$.

The Python script for detecting outliers using IQR is as follows.

```
# Detect outliers using IQR
def detect_outliers(data, column):
  Q1 = data[column].quantile(0.25)
```

```
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Return a Boolean series where True indicates the presence of an outlier
    return (data[column] < lower_bound) | (data[column] > upper_bound)
df['sog_outliers'] = detect_outliers(df, 'sog')
df['cog_outliers'] = detect_outliers(df, 'cog')
df['heading_outliers'] = detect_outliers(df, 'heading')
df
```

We can extend the dashboard built in Sect. 9.5 to display the outliers. The visualization would help a data analyst to decide whether the outliers are errors or important events.

9.6 Cleaning Spatiotemporal Trajectories

A trajectory is essentially a geospatial time series. In this section, we will apply smoothing to clean the vessel trajectories using a technique known as **Kalman filter**. Unlike the simpler smoothing methods studied in Sect. 9.5.2, which can only refine the directly measured values such as location, the Kalman filter can also estimate other dynamic variables such as speed and acceleration. We give next and introduction to this method. For a more detailed explanation on the topic, we refer to the references mentioned in the bibliographic notes at the end of this chapter.

9.6.1 Kalman Filter

The Kalman filter is a mathematical method that estimates the state of a linear dynamic system from a series of noisy measurements. Intuitively, it as a way to guess the position of a moving object by combining two uncertain pieces of information: (1) the noisy information from the GPS sensor; and (2) a prediction of where the object might be now, based on its last known position and based on its motion model, as expressed by the speed, velocity, acceleration, etc. Over time, as more data comes in, the Kalman filter updates its guesses to become more accurate. We study next the two main models of this technique: the *measurement model* and the *transition model*.

Measurement Model

The measurement model provides a mathematical model for the trajectory and the noise. Due to sensor noise, the measurements that are going to be smoothed are not exact. This error is usually modeled by adding unknown, random Gaussian noise to the actual trajectory points to give the known, measured trajectory. This can be expressed as $z_i = x_i + v_i$, where z_i is a measurement at time i, x_i is the corresponding unknown object's position, and v_i is the Gaussian noise component. Note that all these variables are 2D or 3D vectors, depending on the space of the movement. Since in our AIS example the movement is in 2D, the measurement vector would be represented as $z_i = [z^x, z^y]$. However, 3D vectors must be used for aircraft trajectories.

The noise vector v_i in 2D, is assumed to be drawn from a two-dimensional Gaussian probability density with zero mean and a diagonal covariance matrix R, which is denoted as $v_i \hat{=} N(0, R)$, $R = [(\sigma^2, 0), (0, \sigma^2)]$. The two dimensions account for various noise distributions for the x and y coordinates. For GPS measurements, the same distribution for both dimensions is typically used, and thus we repeat σ^2 over the diagonal. The standard deviation of the error σ is a parameter that should be set based on the knowledge about the GPS sensor. Certain manufacturers include such information in the sensor's manual. In the example we show later, we will estimate $\sigma = 10$ meters. It is important to note that the value of σ must be in the same units as the coordinates z_i. Therefore, it is preferred to transform the coordinates of the trajectory into a suitable projected coordinate system before applying the Kalman filter, rather than directly applying the filter on longitude and latitude geographical coordinates. The reason for this is that it is easier to estimate the GPS error standard deviation in meters than in degrees. For the AIS example, we will extract the coordinates from the GeomProj attribute of the AISInputSample table, which is in the SRID 25832.

Transition Model

The Kalman filter can be thought of having two distinct phases. In the *predict* phase, the object's position is modified according to the laws of motion, which is the state transition model. A new position estimate and a new covariance are calculated. In the *update* phase, a measurement of the object's position is obtained from the GPS, which has some inherent uncertainty. Therefore, the covariance relative to the prediction from the previous phase determines how much the new measurement will affect the updated prediction.

In the model, the *state vector* describes the full state of the object being tracked. The Kalman filter gives estimates for this vector. In our case, the state vector will include both the object's location and the velocity:

$$x_i = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \begin{array}{l} \text{Position } X \\ \text{Position } Y \\ \text{Velocity in } X \\ \text{Velocity in } Y \end{array}$$

The elements x and y are the actual position of the object at time i. This position is unknown and needs to be estimated. The $\dot{x}$ and $\dot{y}$ values are the x

and y components of the actual velocity at time i, which is also unknown and needs to be estimated. Although the velocity is not directly measured, the Kalman Filter is designed to produce an estimate of all the components of the state vector x_i, including the velocity. Kalman filters have the capability of inferring higher order variables such as velocity and acceleration by including them as part of the state vector.

Since the object moves at constant velocity, the dynamic model shall be described by the first law of motion given by $x_i = \mathbf{A} \cdot x_{i-1}$ which gives:

$$x_i = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}_{i-1}$$

This is a simple linear translation of the x and y coordinates in the direction of the movement, with an amount proportional to Δt between the past and current timestamps. Note that the assumption of the object moving at constant velocity is far from reality since, for example, ships do change speed and heading during their movement. However, Kalman filters account for this inaccuracy by introducing a noise term to the equation as follows: $x_i = \mathbf{A} \cdot x_{i-1} + w_{i-1}$. Also here, we assume that w_i is a zero mean Gaussian noise. For clarity, we consider that w_i is a 4×1 vector as follows: $w_i \hat{=} [0, 0, N(0, \sigma_v^2), N(0, \sigma_v^2)]^T$. This means that the random noise term in the transition model is applied to the velocity components $\dot{x}, \dot{y}$. The transition model in the Kalman filter can thus be understood as:

$$x_i = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}_{i-1} + \begin{bmatrix} 0 \\ 0 \\ N(0, \sigma_v^2) \\ N(0, \sigma_v^2) \end{bmatrix}_{i-1}$$

The first two rows translate the previous position with the magnitude and direction of the previous velocity, in proportion to Δt. These two rows are thus assuming an exact constant velocity. The last two rows apply the Gaussian noise to the velocity components. Accordingly, in the next iteration, the velocity may be different. This is how the Kalman filter is still able, up to some limit, to process real-world trajectories, while using the constant velocity assumption. However, it is important to note that these formulas and explanation are a simplification of the underlying mathematics and functionality of the Kalman filter. The reader interested in delving into more detail can refer to the bibliographic notes at the end of this chapter.

We show next how to implement this in Python. We use the Stone Soup library,[6] an open-source Python library which provides multiple models for tracking and state estimation of moving objects.

[6] `https://stonesoup.readthedocs.io/`

9.6.2 Implementation of the Kalman Filter

We now show how the theoretical ideas explained in the previous section, can be implemented in Python. We follow the same organization as above, explaining first the implementation of the measurement model and then the transition model. Finally, we show how to display the solution graphically.

Measurement Model

The Python script below, shows the library imports and the definition of the measurement model explained in Sect. 9.6.1.

```
# Library imports
import geopandas as gpd
import pandas as pd
import numpy as np
from sqlalchemy import create_engine
from stonesoup.models.transition.linear import
    CombinedLinearGaussianTransitionModel, ConstantVelocity
from stonesoup.models.measurement.linear import LinearGaussian
from stonesoup.predictor.kalman import KalmanPredictor
from stonesoup.updater.kalman import KalmanUpdater
from stonesoup.types.state import GaussianState
from stonesoup.types.detection import Detection
from stonesoup.types.array import CovarianceMatrix
from stonesoup.types.hypothesis import SingleHypothesis

measurement_noise_std= [10.0, 10.0]
measurement_model = LinearGaussian(
  ndim_state=4, # Number of dimensions of the state vector
  mapping=(0, 2), # Mapping measurement vector index to state index
  noise_covar=np.diag( # Covariance matrix for Gaussian PDF
    [measurement_noise_std[0] ** 2, measurement_noise_std[1] ** 2] ))

measurement_model.matrix() # Display the measurement model matrix
```

As mentioned, the linear Gaussian measurement model (implemented using the LinearGaussian class), is set up by indicating the number of dimensions in the state vector and the dimensions that are measured, specifying the number of state dimensions and the noise covariance matrix. The mapping argument is set to $[0, 2]$ because those are the x and y position indexes from the state vector. The last line is added in this example to allow the reader to visualize the output at this stage, which is given next.

$$\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. \end{bmatrix}$$

Transition Model

The script for the transition model is given next. We use a CombinedLinearGaussianTransitionModel for modeling the state transitions, defining models for constant velocity in both x and y directions. This model assumes that, between two measurements, the position changes linearly based on velocity, while the velocity itself remains constant with some amount of Gaussian process noise. The standard deviation of this noise is supplied as a parameter.

```
# Define the transition model (constant velocity)
process_noise_std= [1, 1] # Should be supplied as an argument
transition_model = CombinedLinearGaussianTransitionModel([
  ConstantVelocity(process_noise_std[0] ** 2),
  ConstantVelocity(process_noise_std[1] ** 2)])
transition_model
```

In the script above, the ConstantVelocity class creates a one-dimensional constant velocity model with Gaussian noise. Also, the CombinedLinearGaussianTransitionModel class takes a number of 1D models and combines them in a linear Gaussian model of arbitrary dimension (in our case, the simulation is in 2D, as we can see in the function arguments). The last line in the code allows to show the result of the model, which is:

```
CombinedLinearGaussianTransitionModel(
    model_list=[ConstantVelocity(noise_diff_coeff=1, seed=None),
                ConstantVelocity(noise_diff_coeff=1, seed=None)],
    seed=None)
```

As an example, taking an interval of one second, the transition model matrix can be displayed with

```
transition_model.matrix(time_interval=timedelta(seconds=1))
```

and the result is:

$$\begin{bmatrix} 1. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 1. \\ 0. & 0. & 0. & 1. \end{bmatrix}$$

The covariance matrix over the same interval can be displayed as:

```
transition_model.covar(time_interval=timedelta(seconds=1))
```

giving the following result:

$$\begin{bmatrix} 0.33333333 & 0.5 & 0. & 0. \\ 0.5 & 1. & 0. & 0. \\ 0. & 0. & 0.33333 & 0.5 \\ 0. & 0. & 0.5 & 1. \end{bmatrix}$$

We next continue with the process. Although actually the initialization and execution of the model can be considered as part of the implementation of the transition model, we explain them separately, for clarity.

Initializing the Kalman Filter

The process begins with some preparation for the Stone Soup library. We construct a list of Detection objects from the measurements we have loaded into the GeoPandas DataFrame from the AISInputSample table. Each detection, that is, a GPS sample, is created by extracting the x and y coordinates from the GeomProj attribute of each row in the DataFrame gdf, and pairing these coordinates with their corresponding timestamps. Additionally, each detection is associated with the measurement model, which describes how measurements relate to the internal states of the filter and includes noise characteristics that reflect the expected accuracy of these measurements. In our case, the same measurement model applies to all detection objects.

```
detections = [
  Detection(np.array([row.geomproj.x, row.geomproj.y]),
    timestamp = row.timestamp, measurement_model = measurement_model)
  for _, row in gdf.iterrows()]
```

After creating the detections, the initial state of the Kalman filter must be defined. The initial state estimate can usually be obtained from the first measurement, i.e., z_0. The uncertainty in this initial state is quantified by a diagonal covariance matrix, where we use the noise parameters.

```
# Initial state (assuming we start from rest)
initial_state_mean =
  # [x, x_velocity, y, y_velocity]
  [gdf.geomproj.iloc[0].x, 0, gdf.geomproj.iloc[0].y, 0]
initial_state_covariance =
  np.diag([measurement_noise_std[0]**2,
    process_noise_std[1]**2,
    measurement_noise_std[0]**2,
    process_noise_std[1]**2])
initial_state = GaussianState(initial_state_mean, initial_state_covariance,
  timestamp=detections[0].timestamp)
```

The class GaussianState is parameterized by a mean state vector, a covariance matrix, and an instant. Initially, we have a state with the first detection and the initial matrixes.

As explained in Sect. 9.6.1, since the probabilistic model is based on successive applications of predicting and updating methods, two methods are needed, namely, predictor and updater. The former applies the transition model based on an hypothesis that associates a prediction with a measurement. The latter calculates the next state estimate. At this point, we instantiate the predictor and the updater of the Kalman filter. We explain these in detail in the next section.

```
predictor = KalmanPredictor(transition_model)
updater = KalmanUpdater(measurement_model)
```

We are now ready to run the Kalman filter process.

Running the Kalman Filtering Process

Recall that our goal is, given a sequence of noisy measurements $z_i, i \in [1, \ldots, n]$, to infer the actual positions x_i. Kalman filtering is a two-step process that runs in multiple iterations. In every iteration i, the following two steps are executed:

1. Prediction: Using the state transition model, the object position is estimated using its previous *estimated state*, that is, $p(x_i) = f(x_{i-1}, w_{i-1})$. The very first state is clearly not predicted. Rather, as we did above, it is initialized by the first observation.
2. Update: The goal of the update process is to incorporate the measurement into the prediction result. It thus takes the output of the prediction step, and trades it off with the observed value z_i. Statistically speaking, the Kalman filter is an *optimal estimator*. In the update step, it minimizes the expected value of the squared error between the estimated state and the true state. This form of optimality is specifically referred to as the minimum mean square error (MMSE) estimator.

The code for running the Kalman filter is given next.

```
# List to store filtered states
filtered_states = []

# Filtering process
for i, detection in enumerate(detections):
  if i == 0:
    # For the first measurement, there is no prediction step
    predicted_state = initial_state
  else:
    # Predict the next state using the prior state
    predicted_state =
      predictor.predict(filtered_states[-1], timestamp=detection.timestamp)

  # Create a hypothesis associating the predicted state with the detection
  hypothesis = SingleHypothesis(predicted_state, detection)

  # Update the state with the hypothesis
  updated_state = updater.update(hypothesis)

  # Store the filtered state
  filtered_states.append(updated_state)

# Extract the smoothed coordinates
smoothed_coords = np.array(
  [[state.state_vector[0, 0], state.state_vector[2, 0]]
  for state in filtered_states])
return smoothed_coords
```

Note that we start with an empty list of estimated states, called filtered_states, which will be populated as the iterations progress. Then, the code iterates over the list of detections, which is, basically, the list of measurements. Stone Soup requires that a prediction and a detection are associated explicitly. This is done using a Hypothesis, and we use SingleHypothesis, the simplest one, which associates a single predicted state with a single detection. Finally, the state is updated and the result is appended to the filtered_states list. This output list represents a smoothed version of the original trajectories, where noisy measurements are edited into coordinates that fit the physics properties of the motion.

Figure 9.4 illustrates an example of an AIS trajectory and its corresponding smoothed version using Kalman filter.

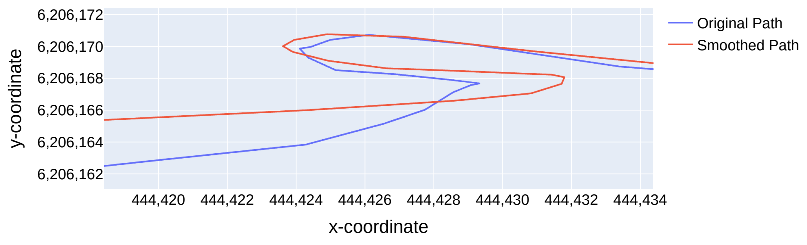


Fig. 9.4 Illustration of trajectory smoothing using Kalman filter for the MMSI 219019011.

9.7 Cleaning Network-Constrained Trajectories

The previous section explored how to clean vessel trajectories. Note that while vessels in certain areas might follow designated routes, deviations of several meters are acceptable due to the absence of physical constraints. This contrasts with other types of transportation such as cars and trains, which are limited to specific roads and rail tracks, respectively. The latter kind of movement is referred to as **network-constrained movement**.

When it comes to gathering GPS data for network-constrained vehicles, the recorded GPS points may not always be precise and, when shown on maps, can fall outside the actual road or track. For example, consider a GPS route recorded by a mobile phone shown in Fig. 9.5, where such inaccuracies are clearly visible. Knowing that the movement of such vehicles is confined to a network (like roads for cars) offers an opportunity for data cleaning. In this scenario, the road map provides essential context for movement. The main task in cleaning such data involves repositioning the erroneous GPS points back onto the correct network path, using the road map as a guiding framework. This will ensure that the data reflects the actual, intended routes of travel, enhancing the accuracy and usability of GPS tracking information.

Map matching is a well-known cleaning task for network-constrained trajectories, which involves aligning GPS points with the actual roads they

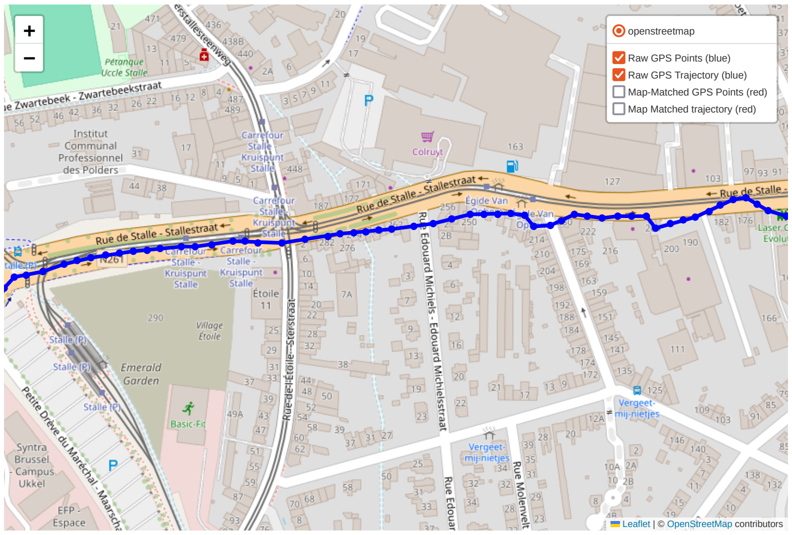

openstreetmap
Raw GPS Points (blue)
Raw GPS Trajectory (blue)
Map-Matched GPS Points (red)
Map Matched trajectory (red)
Rue de Stalle - Stallestraat
Rue Edouard Michiels - Edouard Michielsstraat
Rue de l'Etoile - Sterstraat
Rue Zwartebeek - Zwartebeekstraat
Petite Drève du Maréchal - Maarschalk
Rue Molenveld
Colruyt
Emerald Garden
Leaflet | © OpenStreetMap contributors

Fig. 9.5 A segment of the raw GPS track of a car driving in Brussels. Many coordinates fall outside the road network due to GPS errors.

traverse. This process is critical, not only for ensuring the accuracy of the recorded data, but also for analyzing traffic flow, deducing vehicle speed, and predicting future movements. However, map-matching is more complex than it might initially seem. We explain this in this section.

At a first glance, one may think to solve the map-matching problem by simply aligning each GPS point with the nearest road. This trivial approach seems intuitive but often leads to inaccurate results. The reason for this inaccuracy is twofold: the inherent noise in GPS data and the proximity of multiple roads. For instance, consider a hypothetical scenario depicted in Fig. 9.6, where a car is moving from P1 to P7. Applying the simplistic technique described above, would lead to incorrectly mapping the point P1 to the nearby road on its right and P2 correctly to the road on its left. These mismatches can result in a trajectory that suggests that the vehicle is traveling backwards from P1 to P2, before moving forward from P2 to P3, which is clearly wrong. A similar mistake would also occur with P4. A reasonable map-matching algorithm needs to go beyond merely snapping individual GPS points to the nearest roads: it must consider the sequence of movements between consecutive points. It must also be able to deal with GPS data inaccuracies, such as occasional outliers, and low sampling rates, which can significantly affect the predictability of the actual path.

A popular approach is to use a **hidden Markov model** (HMM) and **Viterbi decoding**, with the actual road segments as hidden states and the GPS points as observations. This approach was proposed by Newson and

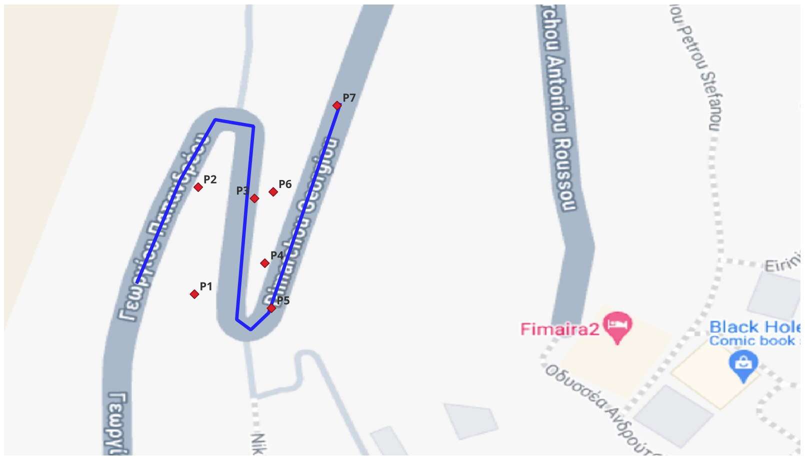


Fig. 9.6 Noisy GPS measurements (red points) of a car trajectory in Syros, Greece. Map matching consists in reconstructing the actual car trajectory (in blue), which is unknown, by placing the GPS points within the road network.

Krumm and implemented in various services such as Valhalla,[7] GraphHopper,[8] and Mapbox's Map Matching API for OpenStreetMap.[9] The HMM, being a probabilistic model, accounts for the variability and uncertainty inherent to GPS data. Unlike simpler geometric methods, probabilistic algorithms do not just look for the nearest road but evaluate multiple potential paths through the road network. By considering various possible trajectories, these algorithms aim at determining the most likely path taken by the vehicle. This enables a more dynamic and accurate (compared against the simplistic options) interpretation of the raw GPS data, accommodating errors, and providing a realistic and reliable reconstruction of the vehicle's route.

9.7.1 Map-Matching using the Hidden Markov Model

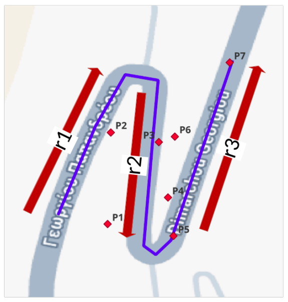
A hidden Markov model (HMM) assumes that the true state of a system is unknown, i.e., hidden, yet it is observable via noisy measurements over time. In map matching, the GPS traces recorded by a device represent these noisy measurements, which indirectly provide information about the actual object's movement on a road network, including the roads taken and turns made. Therefore, HMM is a natural fit for solving the map-matching problem.

[7] https://github.com/valhalla

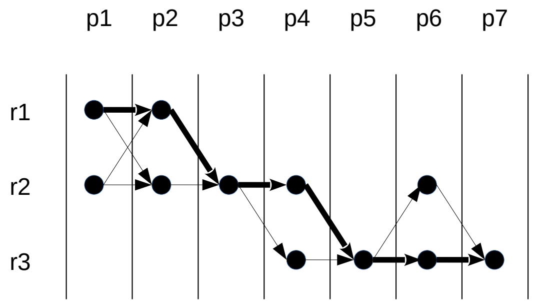
[8] https://graphhopper.com

[9] https://www.mapbox.com/

Given a sequence of *position measurements* $\{p_1, \ldots, p_T\}$, the goal of HMM map matching is to find, for each measurement p_t taken at time t $(1 \leq t \leq T)$, its most likely corresponding location on the map, denoted by s_t^i, from a set of *candidates* $S_t = \{s_t^1, \ldots, s_t^n\}$, which is referred to as a *candidate vector*. That means, for each observation at a certain instant t, we have a set of candidates. For each consecutive pair of candidate vectors S_t and S_{t+1} $(0 \leq t < T)$, transitions exist between each pair of *matching candidates* s_t^i and s_{t+1}^j, which represent the possible routes between these locations on the map.



(a)



(b)

Fig. 9.7 Applying hidden Markov model (HMM) map-matching to the trajectory in Fig. 9.6. (**a**) The noisy position measurements (red points), the true trajectory (blue line), and the underlying road segments (red arrows), (**b**) The candidate and true routes (light and bold arrows, respectively).

Figure 9.7a shows a sequence of position measurements $\{p_1, \ldots, p_7\}$ illustrated as red points. The corresponding true trajectory is illustrated in blue. This true trajectory is unknown and the goal of map-matching is to construct it. The red arrows in the figure highlight three road segments against which the map-matching is done. In Fig. 9.7b, the candidate vectors are the mappings from observations to road segments, illustrated as the columns in the figure. For example, this means that the observation p_1 could be matched to roads r_1 or r_2. The pairs of candidate matches are connected by the arrows. The bold arrows illustrate the true matchings. For instance, this means that r_1 was chosen for p_1 and p_2, meaning that the transition was $p_1 - p_2$.

To determine the most likely matching candidate, we need to compute the probabilities of the matching candidates and their respective transitions. An HMM defines two types of probability measures:

- **Emission probability**: A position measurement is subject to error and uncertainty. This is quantified with an emission probability $p(p_t \mid s_t)$, which defines the conditional probability of observing the measurement p_t given that the true position on the map is s_t. Emission probabilities are modeled using the distance between the measured position and its true position, which is typically described by a Gaussian distribution with a standard deviation σ. Although it is known that GPS errors are not strictly Gaussian, this assumption proved effective in map-matching algorithms. We use a value of $\sigma = 5$ meters, although this depends on the use case.

- **Transition probability**: All transitions between pairs of matching candidates $s_{t-1} \to s_t$ are annotated by transition probabilities, quantified as $p(s_t \mid s_{t-1})$. This defines the conditional probability of reaching s_t, given that the previous state is s_{t-1}. Transition probabilities can account for several information in the base map, including the segment length and speed limit, the segment direction, and the mode (in case of multimodal trajectories). Sometimes it may happen that some transitions are very unlikely, such as those requiring a complicated set of maneuvers. Typically the implementations of HMM map-matching use simple calculations as default, such as the difference between the routing distance and the direct line-of-sight distance between the respective positions.

After the two types of probabilities are calculated for the candidates and the matchings between them, the **Viterbi decoding** finds the most likely sequence of candidates. It is essentially an optimization tool used to find the path through the state space that maximizes the likelihood of the observed sequence of events, given the model parameters. In HMM map-matching, the objective of the Viterbi decoding is to maximize the probability of a particular state path given the observed data. This is achieved by applying the arg max function over the probabilities of all possible paths that could lead to the current observation. Dynamic programming is used to efficiently compute the path that has the maximum probability. The arg max operation finds the argument that returns the maximum value from a target function. It is typically used in machine learning for finding the class with the largest predicted probability. The core of the decoding is to find the most probable sequence of states $s^*_{1:n}$ that could have produced the input sequence of n observations. At each step t, the decoding solves the subproblem where only the observations up to t are considered:

$$s^*_{1:t} = \arg\max_{s_1,\ldots,s_t} p(s_1,\ldots,s_t|p_1,\ldots,p_t)$$

We next show an implementation through an example.

9.7.2 Implementing Map Matching in Valhalla

In this section, we perform map matching over GPS data using Python and Valhalla, an open-source routing engine and accompanying libraries for OpenStreetMap data. Valhalla comprises several modules covering various functionalities, such as generating directions, optimizing routes, and map-matching. For the latter, the module name is Meili. Valhalla is launched as a server, and its functions are accessed as RESTful APIs. For visualization we will use Folium,[10] a Python library that allows creating Leaflet maps.

The procedure to set up a Valhalla server can be found in the official Valhalla documentation.[11] There is also the option to use a docker image of a Valhalla server. An important step in the server setup is the loading of the map. The map can be obtained from some OpenStreetMap mirror. A common option is Geofabrik.[12] These sources allows to download OSM data in PBF (Protocolbuffer Binary Format) files, which are optimized for performance and widely used in geospatial applications. For users who want to quickly experiment with Valhalla's capabilities without the overhead of setting up their own server, there is an alternative to use a demo server hosted by FOSSGIS. The server is open to the public and includes a full planet map.[13] In this illustration, we will use this demo server.

We start by the necessary imports, and reading the GPS data from the file Tolxelles.csv, representing a sequence of observations in Brussels, Belgium:

```
# Library imports
import folium
import pandas as pd
import geopandas as gpd
import json
import requests
from shapely.geometry import LineString
from pyproj import Geod
from decode_functions import decode

csv_file = "Tolxelles.csv"
df_rawGPS = pd.read_csv(csv_file)

# Convert 'Timestamp' to datetime format
df_rawGPS['Timestamp'] = pd.to_datetime(df_rawGPS['Timestamp'])

# Convert 'Timestamp' to epoch seconds
df_rawGPS['epoch_seconds'] = df_rawGPS['Timestamp'].astype('int64') // 10**9

# Rename columns as Valhalla Meili expects
df_rawGPS = df_rawGPS.rename(
  columns={'Latitude': 'lat', 'Longitude': 'lon', 'epoch_seconds': 'time'})

# Create a GeoDataFrame
gdf_rawGPS = gpd.GeoDataFrame(df_rawGPS,
  geometry=gpd.points_from_xy(df_rawGPS.lon, df_rawGPS.lat), crs="EPSG:4326")
```

Next, we prepare the request to the Valhalla service with our GPS data to get the map-matched route. Since the GPS data can be fairly large, the

[10] https://github.com/python-visualization/folium
[11] https://github.com/valhalla/valhalla
[12] https://www.geofabrik.de/
[13] https://valhalla1.openstreetmap.de/

request uses the POST method. The endpoint for map-matching requests is trace_route. Notice the URL of the demo server hosted by FOSSGIS in the variable url. This should be replaced by the user's URL, e.g., localhost, when using the docker version, or installing the Valhalla server.

```
# Convert the DataFrame to a list of dictionaries (JSON records)
meili_coordinates = df_rawGPS.to_dict(orient='records')

# Valhalla Meili Request
head = '{"shape":'
tail = """,
  "search_radius": 10,
  "shape_match":"map_snap",
  "costing":"auto",
  "use_timestamps": true,
  "format":"osrm"}
  """
meili_request_body = meili_head + meili_coordinates + meili_tail
url = "https://valhalla1.openstreetmap.de/trace_route"
headers = {'Content-type': 'application/json'}
data = str(meili_request_body)
r = requests.post(url, data=data, headers=headers)
```

In this Valhalla Meili request, several parameters are configured to optimize the map-matching process. The search_radius is set to 10 meters, defining the area within which Valhalla will search for potential road candidates corresponding to each GPS measurement. We choose the value of 10 meters as a reasonable indication of a GPS error. Clearly we do not want to choose a big radius, otherwise the algorithm will examine too many segments and become slower. The costing parameter is set to "auto", which activates a short-time-path costing model designed for driving routes, ensuring that the route obeys typical automobile driving rules. Other possibilities include "bus" and "pedestrian". The shape_match parameter is set to "map_snap", indicating that the input GPS data might be noisy and requires alignment to the closest road edges in Valhalla's map, thereby enabling the map-matching functionality. The use_timestamps parameter is set to true, meaning that timestamps or durations will be used to compute the elapsed time at each edge along the matched path, providing a time-aware route. Finally, the format is specified as osrm, which ensures that the output data is formatted to be compatible with the Open Source Routing Machine (OSRM) format, a widely-used standard for route calculation on road networks.

Parsing the response is not that simple. Valhalla's map-matching uses an encoded polyline format to compress the result into a single string. A special decode function is thus used to parse the results.[14]

```
# Polyline decoding in Valhalla
response = r.json()
search_1 = response.get('matchings')
search_2 = dict(search_1[0])
```

[14] https://valhalla.github.io/valhalla/decoding/

```
polyline = search_2.get('geometry')
search_3 = response.get('tracepoints')
lst_MapMatchingRoute = LineString(decode(polyline))
```

Finally, we wish to visualize the original trajectory along with its map-matched counterpart. The next script does this job.

```
# Decode the polyline and create a LineString object
lst_MapMatchingRoute = LineString(decode(polyline6))

# Create a GeoDataFrame for the map-matching route
gdf_MapMatchingRoute = gpd.GeoDataFrame(
  geometry=[lst_MapMatchingRoute], crs="EPSG:4326")

# Extract the individual points from the LineString
gdf_MapMatchingRoute_points = gdf_MapMatchingRoute.explode(index_parts=False)

# Create a GeoDataFrame for map-matched GPS points
df_mapmatchedGPS_points = pd.DataFrame(
  [d['location'] for d in search_3 if d and 'location' in d],
  columns=['lon', 'lat'])
gdf_mapmatchedGPS_points = gpd.GeoDataFrame(
  df_mapmatchedGPS_points, geometry=gpd.points_from_xy(
    df_mapmatchedGPS_points['lon'], df_mapmatchedGPS_points['lat']),
  crs="EPSG:4326")

# Drop JSON nonserializable columns from the raw GPS GeoDataFrame
gdf_rawGPS = gdf_rawGPS.drop(columns=['Timestamp'])

# Initialize the map
m = folium.Map(location=[50.802545, 4.341914], tiles='openstreetmap',
  zoom_start=14)

# Add raw GPS points to the map
folium.GeoJson(gdf_rawGPS, style_function=lambda x: {'color': 'red'},
  marker=folium.CircleMarker(radius=4, weight=0, fill_color='red', fill_opacity=1),
  name='rawGPS_points').add_to(m)

# Add map-matched GPS points to the map
folium.GeoJson(gdf_mapmatchedGPS_points, marker=folium.CircleMarker(
  radius=6, weight=0, fill_color='blue', fill_opacity=1),
  name='MapMatching_rawGPS_points').add_to(m)

# Add map-matching route to the map
folium.GeoJson(gdf_MapMatchingRoute, style_function=lambda x: {'color': 'green'},
  marker=folium.CircleMarker(radius=4, weight=0, fill_color='green', fill_opacity=1),
  name='MapMatching_Route').add_to(m)

# Add layer control to switch between layers
folium.LayerControl(position='topright', collapsed=False).add_to(m)

# Display the map in the notebook
m
```

The resulting visualization is illustrated in Fig. 9.8, where the blue trace is obtained from the raw GPS measurements and the red trace is the result of the map-matching process. The red trace connects the map-matched points by traversing the road segment between them. Note that this is more complex than a simple linear interpolation.

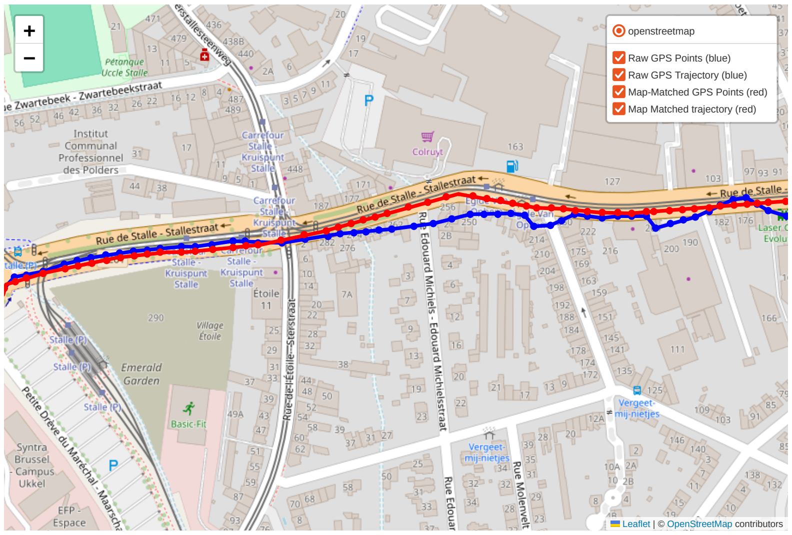

openstreetmap
Raw GPS Points (blue)
Raw GPS Trajectory (blue)
Map-Matched GPS Points (red)
Map Matched trajectory (red)
Rue de Stalle - Stallestraat
Rue Edouard Michiels - Edouard Michielsstraat
Rue de l'Etoile - Sterstraat
Leaflet | © OpenStreetMap contributors

Fig. 9.8 Map matching the trajectory in Fig. 9.5 with Valhalla. The blue trace represent the GPS measurements and the red trace is the map-matched counterpart.

9.8 Summary

In this chapter, we studied the topic of cleaning mobility data, which is crucial in the mobility data science workflow. We focused on the problems that are present in mobility data, in contrast to alphanumeric data cleaning. Since the attributes of mobility data can be classified into static, voyage-related, temporal, and trajectories, the chapter explained how to clean each type. First, we addressed the cleaning of static attributes including problems of null detection and imputation. Voyage-related attributes are cleaned in a similar way to static attributes, yet locally within individual voyages. The main challenge then is to identify voyages in the complete trajectory. Temporal attributes like time series were also studied, focusing on smoothing techniques. Finally, we studied methods for the two kinds of spatiotemporal trajectories, namely, free-space and network-constrained trajectories. For the first case, Kalman filters were presented and applied for cleaning AIS data.

For the second case, the hidden Markov model was presented and applied for map matching GPS trajectories to road segments in Brussels.

9.9 Bibliographic Notes

The broader context of data cleaning and quality assessment in relational data is addressed in [5, 53]. The specific challenges posed to data science by mobility data, including data cleaning, are extensively discussed in several publications [146, 183, 185]. More specific to spatiotemporal trajectory data, in [89] is described a protocol for discovering data quality issues. Other AIS-specific examples include outlier detection and cleaning in vessel trajectories [77]. While GPS trajectory data typically undergoes noise and anomaly removal, some studies, like the work on shuttle data [101], reveal valuable insights by analyzing rather than discarding unexpected GPS readings, using noise patterns to infer environmental factors like tall buildings or dense forests near specific locations.

The application of Kalman filters and hidden Markov models for trajectory cleaning is explained in [248], which elaborates on the theory behind these methods. The application of Gaussian processes for trajectory interpolation and prediction is detailed in [154]. One of the early solutions for map-matching has been proposed in [38]. HMM map-matching has been first proposed in [153]. This method has become a cornerstone for the vast majority of existing implementations of map matching. Another approach for map-matching based on the shortest path, instead of considering all possible paths is studied in [43]. SnapNet [143] addresses map-matching of cellular-based trajectories, which have high amount of errors and high sparseness.

MovingPandas [88] is a Python library for trajectory processing. It includes an implementation of Kalman filter for trajectory smoothing based on the Stone Soup library. This library, described in [102], provides a comprehensive framework for tracking and state estimation, and includes bindings to multiple programming languages, including Python.

9.10 Review Questions

9.1 Explain the four classes of data attributes in mobility data.
9.2 Why is data cleaning essential in mobility data analysis?
9.3 What challenges does cleaning spatiotemporal trajectory data present?
9.4 Explain the difference between movement in free space and movement on networks.
9.5 How can GPS errors affect mobility data analysis?

9.6 Describe the general process of cleaning static data in mobility datasets.
9.7 What are static attributes in mobility data? Provide examples.
9.8 Explain how functional dependencies between attributes can aid in data cleaning.
9.9 Why is it needed to define a trip before cleaning voyage-related attributes?
9.10 Why might the definition of a trip vary depending on the application scenario?
9.11 What are temporal properties in mobility data? Provide examples.
9.12 How does time series filtering help in cleaning temporal properties of mobility data?
9.13 How does time-series smoothing improve the quality of temporal data in mobility datasets?
9.14 What are collective outliers, and how do they differ from global outliers?
9.15 Explain the use of the interquartile range (IQR) in detecting outliers in mobility data.
9.16 What is the role of a Kalman filter in mobility data cleaning?
9.17 How does the Kalman filter enhance the accuracy of position and velocity estimates?
9.18 Why is it necessary to consider both position and velocity in Kalman filtering for trajectory cleaning?
9.19 Why is it necessary to project geographical coordinates before applying a Kalman filter?
9.20 What is the role of map matching important in cleaning trajectory data?
9.21 What are the potential pitfalls of using a simple nearest-road approach for map matching?
9.22 How does map matching help in cleaning network-constrained trajectories?
9.23 What is an emission probability in the context of HMM map matching?
9.24 What is a transition probability in HMM map matching, and how is it calculated?
9.25 What is the role of the Viterbi algorithm in map matching.
9.26 What role does the search radius play in the map-matching process using Valhalla?

9.11 Exercises

Exercise 9.1. In Sect. 9.5, we observed that the randomness in both COG and Heading signals when the SOG is low. Propose a cleaning strategy to address these random fluctuations in the following steps:

a. Identify the vessels that demonstrate this pattern. One way to do so, is to search for vessels whose COG and Heading signals deviate from each other significantly.
b. For individual vessels, identify the time durations where such pattern can be seen. One way to do so, is to search for durations where the SOG signal is below some threshold, e.g., 0.5 knots. Try however to make your code robust towards noise and instantaneous SOG fluctuations.
c. Clean the COG and Heading signals during these identified durations. Consider different variants and reflect on their consequences on the analysis: set to NULL, impute with the last clean value, and impute with a constant.

Exercise 9.2. Section 9.5.2 explains timeseries smoothing using rolling average and rolling median, and an implementation in Python is explained. Write in SQL the following queries.

a. For 10 random MMSIs, smooth the SOG signal of each vessel using a rolling median and a window size 11, centered at the value.
b. For the same 10 MMSIs, smooth the SOG signal of each vessel using a rolling mean and a window size 11, centered at the value.
c. Plot the smoothing results of the previous two questions (in Python) and reflect on their differences.
d. Repeat the whole exercise using different window sizes: 3, 7, 15, 35, and reflect on the results.

Exercise 9.3. Consider the Kalman filter code in Sect. 9.6.1. Vary the values of measurement_noise_std and process_noise_std and discuss how the results change.

Exercise 9.4. Repeat the map-matching example in Sect. 9.7 using OSRM map-matching.[15]

[15] https://project-osrm.org/