EPiC Series in Computing

Volume 67, 2019, Pages 73–84

Proceedings of the 7th OpenSky Workshop 2019

A Python Toolbox for Processing Air Traffic Data: A Use Case with Trajectory Clustering

Xavier Olive[1] and Luis Basora[1]

ONERA DTIS, Université de Toulouse
Toulouse, France
{xavier.olive, luis.basora}@onera.fr

Figure 1: Clustering of aircraft trajectories crossing Switzerland airspace on August 1st, 2018

Abstract

Problems tackled by researchers and data scientists in aviation and air traffic management (ATM) require manipulating large amounts of data representing trajectories, flight parameters and geographical descriptions of the airspace they fly through. The `traffic` library for the Python programming language defines an interface to usual processing and data analysis methods to be applied on aircraft trajectories and airspaces. This paper presents how `traffic` accesses different sources of data, leverages processing methods to clean, filter, clip or resample trajectories, and compares trajectory clustering methods on a sample dataset of trajectories above Switzerland.

1 Introduction

The large volume of available aircraft time-series, or more specifically trajectory data, be it data from onboard Flight Monitoring Systems (FMS), Automatic dependent surveillance–broadcast

(ADS-B) trajectories from large-scale databases like the OpenSky Network [19] or radar tracks available from air navigation service provider (ANSP), opens up a wide range of possibilities for statistical analytics [18], open-data aircraft modelling [22, 21], performance reviews, academic studies [16, 17, 15] or journalistic stories [12].

All these studies follow a general process starting from data acquisition (storage and querying) and data preprocessing (cleaning, selection, normalisation, transformation, feature extraction), and ending with data visualisation.

The `traffic` library [14] is an open-source Python package available under the MIT licence. The library provides:

1. parsers to common sources of air traffic management (ATM) related data including a download assistance tool for the OpenSky Impala shell (Section 2);
2. a declarative grammar to process trajectories, including resampling, filtering faulty data, projecting, querying, intersecting with airspaces (Section 3);
3. exporting facilities to common visualisation tools such as Matplotlib and Altair [24], or Cartopy, Google Earth, Leaflet and Cesium for geospatial data.

We will go in this paper through the possibilities of the library and apply them to the common use case of trajectory clustering. After a brief review of common and lesser-known trajectory clustering methods in Section 4, we will compare results in Section 5.

2 Sources of data

The `traffic` library[1] has been designed to manipulate large amounts of data representing trajectories, flight parameters and geographical descriptions of the airspace they fly through. It is based on three main core classes for handling aircraft trajectories (Flight), collections of trajectories (Traffic) and airspaces (Airspace). Common operations relevant to trajectories evolving in controlled airspaces range from basic attributes: time of entry, time of exit, duration, maximum or minimum altitudes or speed; to more complex operations like intersections of trajectories with airspaces, distances between pairs of trajectories, etc.

Basic navigational data is embedded in the library, together with parsing facilities for the most common sources of information with a main focus on Europe for the time being. A very basic (and outdated) database of airports with their runways, navigational beacons, ATS routes and European FIRs is currently available:

```
from traffic.data import airports, navaids, eurofirs

airports["LSZH"]  # accessible through "ZRH" as well
navaids.extent("Switzerland")["ZUE"]  # ZURICH EAST VOR-DME
eurofirs["LSAS"]  # SWITZERLAND FIR
```

Eurocontrol gives access to their Demand Data Repository (DDR) files in a standardised format to describe trajectories and filed flight plans. A specific class is provided to parse these files commonly referred to by their .so6 extension. Descriptions of the structure of the European airspace during an AIRAC cycle may also be accessed by providing a reference to the appropriate location on disk.

The opensky module is dedicated to handling data from the OpenSky Network infrastructure, both from the publicly available REST API and from the Impala historical database open to

[1] A comprehensive documentation is available at https://traffic-viz.github.io/

academics. The main `history()` function builds and executes the proper SQL requests based on its arguments (including date, callsign, tail number and geographic extent), then downloads and parses the response.

Any Flight instance with trajectory information coming from ADS-B capable transponders, from multilateration or from radar tracks may be enriched with additional information from Mode S Enhanced Surveillance (EHS)[2] such as heading, roll angle, indicated airspeed or selected altitude. The `.query_ehs()` method builds the appropriate request to download all relevant raw DF20 and DF21 messages before decoding them with the `pyModeS` [20] library.

In the following, we use one day of data of traffic above Switzerland on August 1st, 2018:

```
from traffic.data import eurofirs, opensky

switzerland_raw = opensky.history(
    "2018-08-01 05:00",  # UTC time by default
    "2018-08-01 22:00",
    bounds=eurofirs["LSAS"]
)
```

3 Trajectory processing

One of the most natural way to model time series is by using a set of named vectors associated to semantic properties such as type (float, integer, dates, etc.), sampling rate or physical unit. Several programming languages provide appropriate data structures to fit that need, such as data frames in R or the Pandas library in Python.

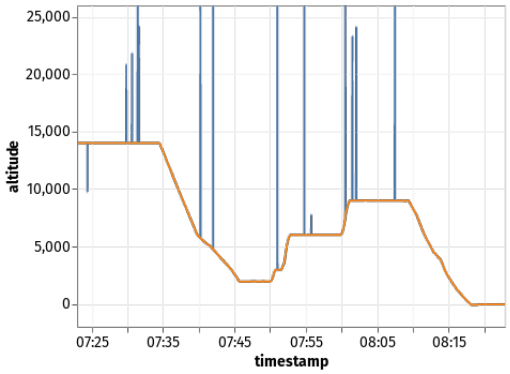
Pandas comes with a set of chainable methods appropriate for generic processing including handling missing data, slicing, querying or resampling. Collections of trajectories need specific operations related to their evolution in controlled airspaces. These operations range from the calculation of basic attributes (e.g. time of entry, time of exit, duration, maximum or minimum altitudes or speed) to more complex operations like intersections of trajectories with airspaces, distances between pairs of trajectories and more.

ADS-B does not provide unique flight identifiers. Transponder codes (hexadecimal identifiers) and callsigns (associated to missions—like emergency services or aerial surveys,—or to routes for commercial aircraft) may be enough to identify most of the commercial trajectories in a short time interval (less than a day), but they are not usually sufficient as stop-over (and most diverted) flights consist of several legs with the same callsigns. `traffic` provides default heuristics to iterate on large scale data frames, following unique identifiers if provided, and yields legs one at a time:

```
for flight in switzerland_raw:
    pass  # yields flight legs one at a time
```

Data from a large scale ADS-B capable receivers' network such as OpenSky Network is provided as is, i.e. noisy and faulty data included. `traffic` includes a set of methods to filter out unusable data such as trajectories with nothing but `NaN` latitude and longitude coordinates, replicated faulty samples or wrongly decoded callsigns. A commonly seen noisy pattern in OpenSky data is the presence of spikes in several features, including altitude. To filter them out, `traffic` offers specific methods implementing algorithms based on a cascade of median filters (see Figure 2).

[2] Mode S EHS improves situation awareness for air traffic controllers who may cross-check aircraft intentions with given clearances. Aircraft are selectively interrogated by Mode S capable Secondary Surveillance Radars (SSR)—unlike ADS-B data which is continuously broadcasted. Mode S EHS is rather well implemented in Europe, partially implemented in Northern America and nonexistent in areas without radar coverage.




```python
# Download data and filter to remove spikes
raw_data = opensky.history(
    "2018-05-30", callsign="TRA051"
)
filtered = raw_data.filter().assign(
    callsign="TRA051 (filtered)"
)

( # Visualisation using the Altair framework
    original.last(hours=1).encode("altitude") +
    filtered.last(hours=1).encode("altitude")
)
```

Figure 2: Altitude signal before and after filtering on the last hour of flight for the B738 flying TRA051 on May 30th, 2018.

Trajectory simplification is a widely shared concern when handling trajectory data, esp. when memory resources are at stake. The Douglas-Peucker [7] algorithm is a well-known simplification technique for geospatial trajectories that works by discarding the points that are relatively close to a straight line. Resampling is a different way to reduce the size of the time series by changing its sampling rate, or by only keeping a given number of samples equally distributed along time. Figure 3 plots various simplifications applied to a sample trajectory. Depending on the needs, shape preservation or data equally distributed along time may be preferable.

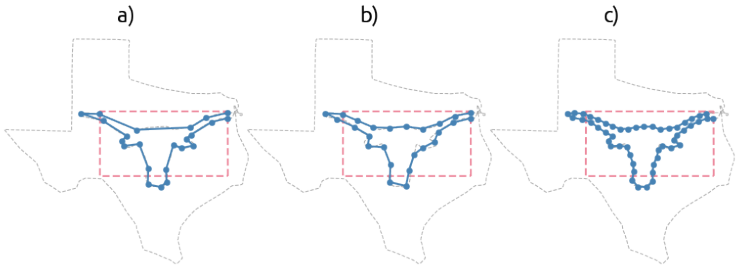
After applying a set of processing operations to all the flights in a Traffic structure, a final merging operation concatenates the data back into a single data frame so as to return a new Traffic instance. Processing a collection of trajectories can be expressed in a programmatic, imperative form like:

```python
cumul = []
for flight in collection:
    if flight.min("altitude") < 30000:
        continue
    cumul.append(flight.filter().resample("30s"))
new_collection = Traffic.from_flights(cumul)
```

or in an equivalent, equally efficient, flattened human-readable declarative form, which could read like *"select trajectories flying above FL300; then apply filtering; then apply resampling"*.

The flattened human-readable declarative description avoids creating an unnecessary large number of full iterations and is kept computing efficient by allowing operations on a Traffic structures to be lazily stacked on for a future evaluation in one single (possibly multiprocessed) loop. More specifically:

- **map** operations typed Flight → Optional[Flight] ❶ are stacked: the flight is discarded if None is returned.
- **filter** operations Flight → bool ❷ return True if the Flight should be kept, False if not. They are stacked using the `.filter_if()` method.
- the final **reduce** operation ❸ is called `.eval()` and returns a Traffic structure.




```python
from shapely.geometry import box
from traffic.data.samples import texas_longhorn

red = box(-96.2, 29.6, -95.7, 29.82)  # bounding box

# a) Douglas-Peucker (1km resolution) yields 23 samples
texas_longhorn.clip(red).simplify(1e3)
# b) 25 samples
texas_longhorn.clip(red).resample(25)
# c) one sample per minute yields 50 samples
texas_longhorn.clip(red).resample("1T")
```

Figure 3: Trajectory of aircraft N56821 on Sep. 17th, 2017, clipped to its inside Longhorn shape part (2981 samples) and simplified using various methods.

In our example, we applied the following preprocessing operations on the raw dataset of trajectories flying over Switzerland:

```python
import pandas as pd

def enroute(flight: "Flight") -> bool:  # ❷
    "Returns True if flight is most likely enroute."
    return (
        flight.duration > pd.Timedelta("10 minutes")
        # filter ground vehicles with no track angles (NaN)
        and flight.min("track").notnull()
        # we consider enroute flights never fly below FL300
        and flight.min("altitude") > 30000
    )

switzerland = (
    switzerland_raw
    # a set of heuristics to remove most faulty data
    .clean_invalid()
    # assign identifiers (default pattern: {callsign}_{index})
    .assign_id()  # ❶
    # cascade of median filters to remove spikes
    .filter().filter(altitude=53)  # ❶
    # keep only en-route flights
    .filter_if(enroute)  # ❷
    # resample to one point every 10 seconds
    .resample("10s")  # ❶
    # multiprocessed evaluation using 4 cores
    .eval(desc="preprocessing", max_workers=4)  # ❸
)
```

4 Trajectory clustering

Clustering is an unsupervised data analysis technique widely used to group similar entities into clusters according to a similarity (or distance) function. Multiple clustering algorithms exist in the literature to cluster point-based data such as K-means [13], BIRCH [26], OPTICS [1], DBSCAN [8] and HDBSCAN [5]. When clustering is applied to trajectories, it requires a proper distance function to be defined between trajectory pairs, which is challenging because of the functional nature of trajectories. The most common approach is to simply sample the trajectory to obtain a n-dimensional vector of points for the use of point-based clustering algorithms and distances such as the Euclidean one.

Trajectory clustering methods based on Euclidean distance do not always produce satisfying results, especially when applied to trajectories with different lengths. Fortunately, more specific distances for trajectory and time series exist in the literature [4]. For instance, warping-based distances such DTW [3], LCSS [25], EDR [6], ERP [6] find an optimal way of aligning the time dimension of trajectories to achieve a perfect match between them. Other distances take better into account the geometry of the trajectories and in particular its shape. The best well known shape-based distances are the Hausdorff [11] and the Fréchet [9] but they do not compare trajectories as a whole. More recently, a more promising shape-based called Symmetrized Segment-Path Distance (SSPD) distance has been proposed [4, 10] which takes into account the total length, the variation and the physical distance between two trajectories. The SSPD distance has been used by Basora et al. [2] in their framework for the analysis of en-route flows based on trajectory clustering with the HDBSCAN algorithm.

Another approach for trajectory clustering that are commonly represented as highly dimensional data is to reduce data dimensionality, project trajectories in a low dimension space as a preprocessing step and apply a traditional clustering algorithms in the lower dimension space. Various techniques can be used to project high-dimensional data: principal component analysis [17] identifies the directions with the more variance and proceeds with a linear transformation of the input data. Autoencoders [15] are a particular kind of hourglass-shaped neural networks, trained to learn a (non-linear) projection/encoding and a reconstruction/decoding operation so that the input and the output of the network match. In the following, we unfold the implementation of a clustering in a two-dimensional space obtained through a t-SNE algorithm [23].

`traffic` provides a convenient clustering API applicable to collections of trajectories. The label associated to each trajectory will be added to the underlying DataFrame:

```
Traffic.clustering(
    clustering: ClusteringProtocol,  # provides fit() and predict()
    nb_samples: int,  # all trajectories are first resampled
    features: List[str] = ["x", "y"],
    projection: Optional[pyproj.Proj],  # computes x and y
    transform: Optional[TransformerProtocol] = None  # StandardScaler()
).fit_predict()
```

The Traffic.clustering method prepares the data present in a traffic.core.Traffic structure in order for it to match the API of usual clustering objects, providing fit(), predict() and/or fit_predict() methods. Geographical coordinates (latitude and longitude) are first projected into x and y coordinates in a local referential through projection methods. The use of standard projections (like EuroPP in Europe) is recommended; if none available, consider conformal conic projections centered around the area of interest. Selected features are then resampled to match a given dimension before being scaled. The final fit_predict() call applies the clustering to the prepared data and subsequently label the whole trajectories.

```python
# CH1903+ is a standard projection system
# in Switzerland
from traffic.core.projection import CH1903p
from traffic.drawing import countries

with plt.style.context("traffic"):
    ax = plt.axes(projection=CH1903p())
    ax.add_feature(countries())
    switzerland.plot(ax, alpha=0.1)
```

Figure 4: Sample dataset of flight trajectories over Switzerland on August 1st 2018 used for clustering in Section 5

Traditional clustering algorithms provided in common ML libraries such as scikit-learn can be passed as is. More complex methods can also be defined. In the following example, t-SNE first projects the trajectories to a two-dimensional space and we apply DBSCAN to label samples on that space. Note that the nature of t-SNE algorithms does not allow to label new samples to make them fit an existing cluster, but the self.labels_ attribute lets us go around this limitation to produce a "one-shot" clustering.

```python
# See implementation: https://github.com/DmitryUlyanov/Multicore-TSNE
from MulticoreTSNE import MulticoreTSNE

class TSNE_Clustering():
    def __init__(self):
        self.tsne = MulticoreTSNE(n_jobs = 4)
        self.latent_clustering = DBSCAN(eps=2, min_samples=10)

    def fit(self, X):
        self.Y = self.tsne.fit_transform(X)
        self.labels_ = self.latent_clustering.fit_predict(self.Y)
```

5 Comparison of clustering methods

In this section, we compare the results of three different clustering methods on the sample collections of 1244 trajectories flying over Switzerland. Figure 4 plots the whole set of trajectories with enough transparency to get an idea of the main flows. Results are on Figure 6:

1. DBSCAN with a default $\varepsilon = 0.5$ value and a minimum of 10 trajectories per cluster yields 19 clusters and a fair share of 50% of outliers;

2. a GaussianMixture of 19 components for a fair comparison with previous method—however GMM allows no outlier;

3. a custom implementation with a DBSCAN applied on the non-linear two-dimensional projected space (see Figure 5) through t-SNE with default parameters yields 20 clusters and very few outliers (less than 1%).

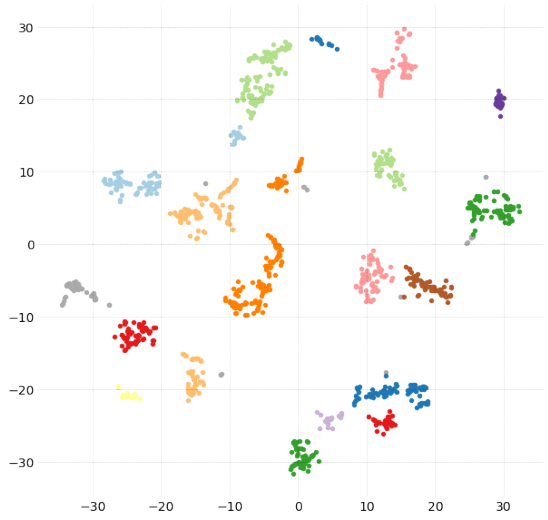


Figure 5: Result of the DBSCAN clustering on the two-dimensional space where t-SNE projected all the trajectories

Sample trajectories cluster naturally in the two-dimensional space as t-SNE is designed to keep close (resp. far) samples in the trajectory space (according to Euclidean distance) close (resp. far) in the two-dimensional space. The relative position of all clusters in the two-dimensional space has no particular meaning.

```
from sklearn.cluster import DBSCAN
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler

from traffic.core.projection import CH1903p

methods = [
    DBSCAN(eps=0.5, min_samples=10),
    GaussianMixture(n_components=19),
    TSNE_Clustering()
]

t_cluster = [
    switzerland.unwrap()
    .clustering(
        nb_samples=15,
        projection=CH1903p(),
        features=["x", "y", "track_unwrapped"],
        clustering=clustering,
        transform=StandardScaler(),
    ).fit_predict()
    for clustering in methods
]
```

The following table describes the number of trajectories per cluster $k \in \{0, \cdots 5\}$:

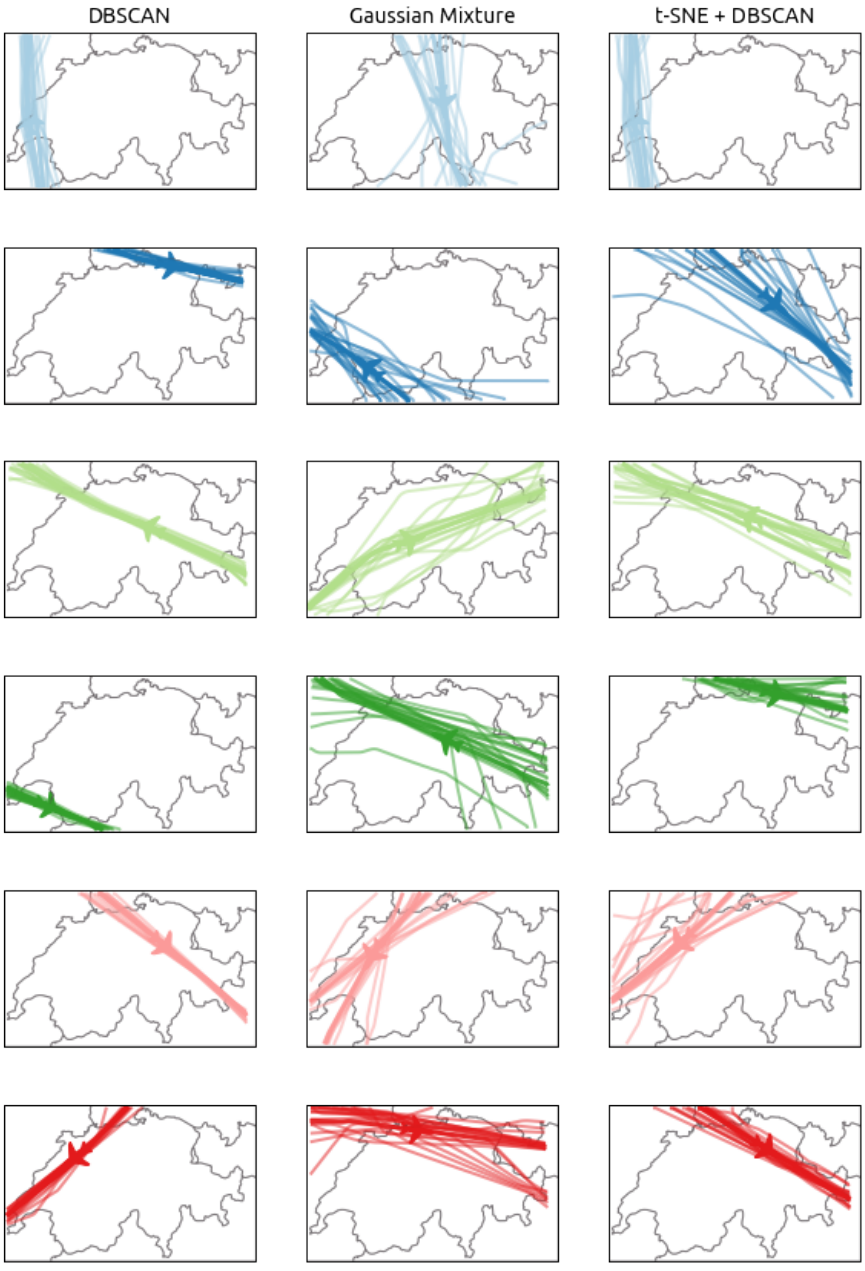


Figure 6: Comparison of clustering methods: DBSCAN, Gaussian Mixture and DBSCAN on the latent space produced by t-SNE

	DBSCAN	GMM	t-SNE
0	62	98	75
1	30	107	77
2	23	27	112
3	17	119	55
4	14	156	78
5	25	29	34

Figure 6 plots the six first clusters yielded by the three considered methods. There is no reason why cluster k according to DBSCAN should match cluster k according to GaussianMixture (GMM) or t-SNE + DBSCAN (t-SNE). However, some comparisons can safely be made:

- generally speaking, clusters yielded by GMM have more dispersion that DBSCAN and to a lesser extent, than t-SNE.
- clusters yielded by t-SNE (T) usually catch more trajectories than DBSCAN (D): compare D0 (62) with T0 (75); D1 (30) with T3 (55); D2 (23) with T2 (112); D4 (14) with T5 (34) and D5 (25) with T4 (78);
- since GMM (G) allows no outlier, the dispersion associated to each cluster comes with clusters making not much sense, like G3 in dark green probably related to D2 in light green but including trajectories coming from the south, or G4 in pink which groups two flows into one cluster.

6 Conclusion

The `traffic` library brings a convenient API to work with common sources of air traffic data. In this paper, we went through the whole process of downloading (Section 2) and processing (Section 3) trajectories over Switzerland on a given days before implementing a custom clustering algorithm (Section 4) comparing it with reference clustering algorithms (Section 5).

An executable Google Colab version of the code presented in this paper is available from the documentation of the `traffic` library on https://traffic-viz.github.io/.

References

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[2] Luis Basora, Jérôme Morio, and Corentin Mailhot. A trajectory clustering framework to analyse air traffic flows. In *Proceedings of the 7th SESAR Innovation Days*, 2017.

[3] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370, 1994.

[4] Philippe Besse, Brendan Guillouet, Jean-Michel Loubes, and Royer François. Review and perspective for distance based trajectory clustering. *arXiv preprint arXiv:1508.04904*, 2015.

[5] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 160–172. Springer, 2013.

[6] Lei Chen, M Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the SIGMOD international conference on Management of data*, pages 491–502, 2005.

[7] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[9] Maurice Fréchet. Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 22(1):1–72, 1906.

[10] Brendan Guillouet. *Apprentissage statistique: application au trafic routier à partir de données structurées et aux données massives*. PhD thesis, Université de Toulouse, Université Toulouse III – Paul Sabatier, 2016.

[11] Felix Hausdorff. *Grundzuge der Mengenlehre*, volume 61. American Mathematical Society, 1978.

[12] Simon Huwiler, Priska Wallimann, Marcel Aerni, and Janine Gygax. Gewimmel am Himmel – So ist der Schweizer Luftraum organisiert, 2018.

[13] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[14] Xavier Olive. traffic, a toolbox for processing and analysing air traffic data. *Journal of Open Source Software*, 4(39), 2019.

[15] Xavier Olive and Luis Basora. Identifying Anomalies in past en-route Trajectories with Clustering and Anomaly Detection Methods. In *Proceedings of the 13th USA/Europe Air Traffic Management Research and Development Seminar*, 2019.

[16] Xavier Olive and Pierre Bieber. Quantitative Assessments of Runway Excursion Precursors using Mode S data. In *Proceedings of the 8th International Conference for Research in Air Transportation*, 2018.

[17] Xavier Olive, Jeremy Grignard, Thomas Dubot, and Julie Saint-Lot. Detecting Controllers' Actions in Past Mode S Data by Autoencoder-Based Anomaly Detection. In *Proceedings of the 8th SESAR Innovation Days*, 2018.

[18] Matthias Schäfer, Xavier Olive, Martin Strohmeier, Matthew Smith, Ivan Martinovic, and Vincent Lenders. OpenSky Report 2019: Analysing TCAS in the Real World using Big Data. In *Proceedings of the 38th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2019.

[19] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. Bringing up OpenSky: A large-scale ADS-B sensor network for research. In *Proceedings of the 13th international symposium on Information processing in sensor networks*, pages 83–94, 2014.

[20] J. Sun, H. Vû, J. Ellerbroek, and J. M. Hoekstra. pymodes: Decoding mode-s surveillance data for open air transportation research. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[21] Junzi Sun. *Open Aircraft Performance Modeling Based on an Analysis of Aircraft Surveillance Data*. PhD thesis, Delft University of Technology, June 2019.

[22] Junzi Sun, Joost Ellerbroek, and Jacco M. Hoekstra. Aircraft initial mass estimation using Bayesian inference method. *Transportation Research Part C: Emerging Technologies*, 90, May 2018.

[23] Laurens van der Maaten and Geoffrey Hinton. Visualizing High-Dimensional Data using t-SNE. *Journal of Machine Learning Research*, 9, 2008.

[24] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 2018.

[25] Michail Vlachos, George Kollios, and Dimitrios Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings 18th international conference on data engineering*, pages 673–684. IEEE, 2002.

[26] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. In *ACM Sigmod Record*, volume 25, 1996.