Jheronimus Academy of Data Science

Engineering Doctorate in

Data Science

Synthetic data generation for Maritime routes

04/06/2025

Table of Contents

1. Introduction

1.1. Problem background

The MARIT-D (Multi AI Real-time Intelligence Tool on Drugs) project is a collaborative pan-EU initiative aimed at disrupting illicit trafficking in Europe by leveraging data and intelligence. MARIT-D develops a robust set of tools that integrate multiple data sources, such as satellite imagery and Automatic Identification System (AIS) data, to detect illegal activities and emerging criminal modus operandi. JADS Research, together with the National Police of the Netherlands, plays a central role in the project's development and success.

Ships and vessels operating in the EU are required to equip Automatic Identification Systems (AIS). AIS is an automatic tracking system that continuously broadcasts a vessel's position, speed, course, and other relevant information. This data is primarily used for maritime safety, allowing port authorities and other ships to track vessel movements and avoid collisions. In the MARIT-D project, AIS data is also analyzed to identify suspicious interactions between vessels, which could indicate illegal activities.

The vessel interaction detection tool analyzes AIS data to identify potential suspicious interactions between ships. However, real AIS data does not come with labels indicating which behaviors are suspicious, and relevant patterns are often rare. This makes it difficult to evaluate and improve the tool effectively. To address this, synthetic AIS data is generated, producing realistic vessel trajectories that mimic real-world movement patterns. This synthetic data allows the tool to be tested and refined under a wider range of scenario.

1.2. Project Objective and Methodology

The goal of this project is to explore and design methodologies for simulating AIS data. Specifically, the project focuses on using an Autoregressive Sequence Model, which learns vessel trajectory patterns from real AIS data and generates new sequences based on these patterns. In this approach, vessel movements are represented as sequences of position points over time. These sequences are used to train the autoregressive model, which can then generate new sequences that mimic realistic vessel trajectories. This allows the creation of synthetic AIS data that closely follows the patterns observed in real-world data. The main advantage of this method is its ability to produce realistic trajectories without manual rule creation. However, its accuracy and performance depend heavily on the quality of the input data, making high-quality training data essential for effective simulation.

2. Autoregressive Sequence Model

2.1. Introduction

We investigate the use of generative AI for generating AIS data, with a specific focus on autoregressive models. Autoregressive models are a type of generative model that generate sequences by predicting the next element based on preceding elements. In essence, they learn to predict the future from the past.

We apply this concept to vessel trips, which are naturally sequential in nature. A vessel trip can be viewed as an ordered sequence of geospatial positions. These sequences begin with a start position, continue through intermediate positions, and end with a final position, capturing the trajectory of a vessel over time. In Figure 1 it can be seen that the vessel trajectory can be represented as an ordered sequence of position points, with each position point having a latitude and longitude coordinate.

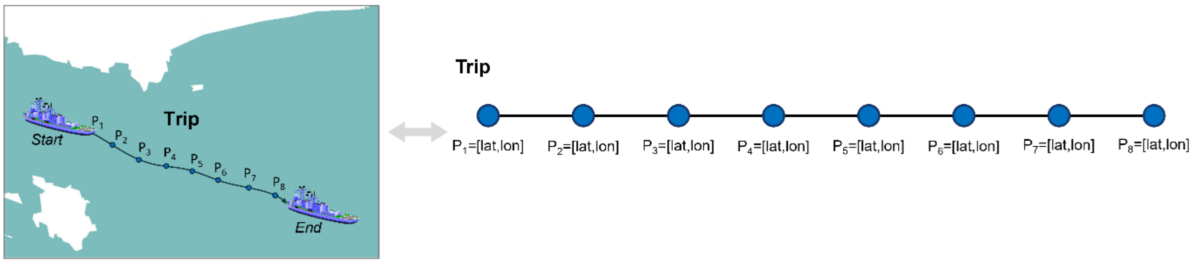


Figure 1. Vessel trajectory on the sea (left) and its corresponding trajectory represented as a sequence of position points ordered by time (right).

The goal of this approach is to train an autoregressive model on real-world AIS data to learn movement patterns and spatial structures inherent in actual vessel trajectories. Once trained, the model should be capable of generating realistic synthetic trajectories by predicting the next position in a trip based on the sequence of previous positions. These two steps are also illustrated in Figure 2. This approach is motivated by the strengths of autoregressive models in modeling time-series data. Their ability to capture temporal dependencies makes them suitable for sequential geospatial tasks like vessel trajectory prediction.

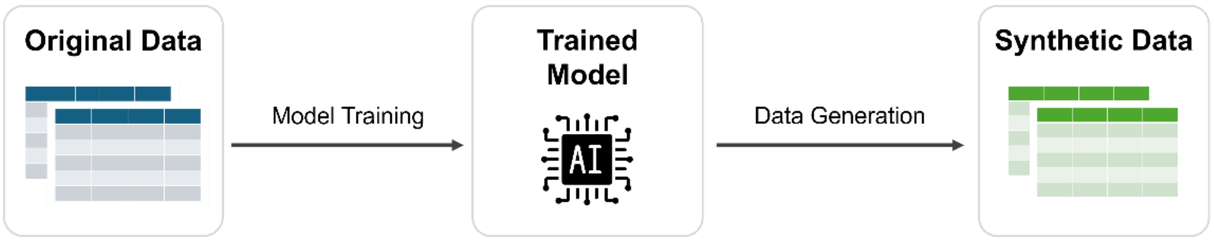


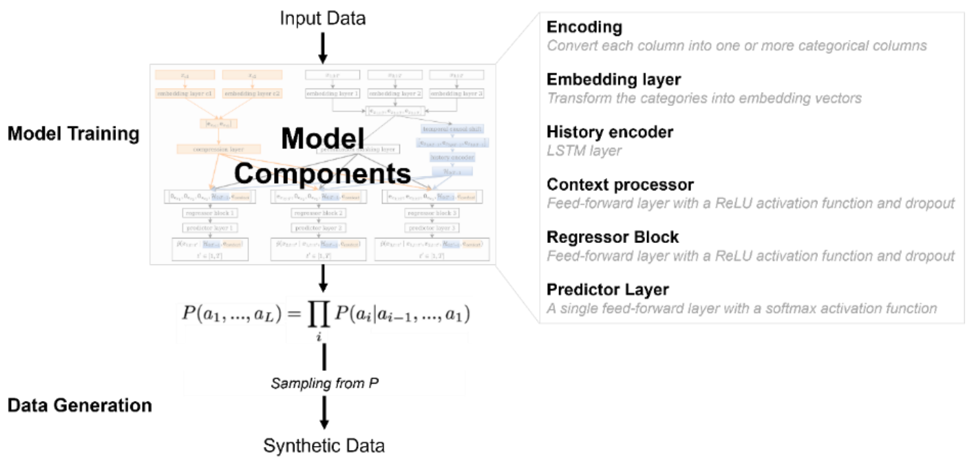
Figure 2. Overview of the approach: the model is trained, and then the trained model is used to generate synthetic data.

2.2. Methodology

Our implementation uses the TabularARGN framework, introduced by (Tiwald, et al. 2025). TabularARGN is an open-source autoregressive framework specifically designed for synthetic tabular data generation. It can be used to transform real input data into synthetic data that closely mimics the original in terms of patterns and statistical relationships. A detailed explanation of the framework is available in the original paper. The TabularARGN framework is provided as a Python package called `mostlyai`, and their code is available on GitHub. The package can be run entirely offline, ensuring that all training data remains local.

The main use case of synthetic data generation using TabularARGN is to allow organizations to safely share, analyze, or use data without exposing any sensitive information. The synthetic data preserves privacy by ensuring that individual records cannot be traced back to real people or entities, making it ideal for privacy-compliant data handling. Although our use case differs, exploring this framework is relevant because it aligns with our goal of training a model to generate new synthetic data that preserves the patterns found in the input data. Additionally, the framework supports various data types, including geospatial locations, making it well suited for our approach.

Using this framework, we train an autoregressive model that, once trained, can generate sequences of geospatial positions representing synthetic vessel trajectories. Autoregressive sequence modeling in TabularARGN involves two steps, as shown in Figure 3. The first step is *model training*. The model consists of different layers, which will be described in the next paragraph. The output of training is a joint probability distribution over the data. The second step is synthetic *data generation*, in which data is generated by sampling from this distribution using random draws.



Input Data
Model Training
Model Components
Encoding
Convert each column into one or more categorical columns
Embedding layer
Transform the categories into embedding vectors
History encoder
LSTM layer
Context processor
Feed-forward layer with a ReLU activation function and dropout
Regressor Block
Feed-forward layer with a ReLU activation function and dropout
Predictor Layer
A single feed-forward layer with a softmax activation function
$P(a_1, ..., a_L) = \prod_i P(a_i | a_{i-1}, ..., a_1)$
Sampling from P
Data Generation
Synthetic Data

Figure 3. From input data to synthetic data using TabularARGN Framework.

Model Architecture

The model architecture consists of the following sequence of components: `Inputs → Embeddings → Context + History → Regressors → Predictors → Output`

An overview of these components is provided in the table below. For a detailed explanation of the model, please refer to the original paper by (Tiwald, et al. 2025). The input data can include various types, but all non-categorical features are first encoded into categorical columns by the framework. The encoding process that converts all columns in our input data, including geospatial coordinates (latitude and longitude), into categorical variables is explained in more detail in Appendix A.

Table 1. TabularARGN Model Components

Component	Description
Inputs	- Supports various data types. - Non-categorical features are encoded into categorical sub-columns.
Embedding Layer	- Each sub-column is assigned an embedding vector. - Produces dense representations.
History Encoder	- Learns dependencies across sequences. - Uses LSTM with dropout. - Learns from all previous time steps.
Context Processor	- Enables conditional generation using static table context. - Processes static table (e.g., Trips) into context embeddings. - Embeddings are concatenated with sequence and history representations.
Regressor Block	- Feed-forward layers with ReLU with dropout. - Dropout prevents overfitting. - Produces intermediate features per column.
Predictor Layer	- Final layer per column. - Outputs probability distribution via softmax. - Supports sampling during generation.

TabularARGN Models

A distinction can be made between two types of input tables: *flat* tables, which lack temporal order, and *sequential* tables, which incorporate a time dimension. The model architecture and training process depend on the type of input table. The three main types of models are:

- **Flat Model**
  - Autoregressively models the joint probability across columns.
  - Suitable for simple table generation without temporal dependencies.
- **Sequential Model**
  - Incorporates temporal ordering by modeling the time dimension.
  - Predicts values based on both prior time steps and previous columns.
- **Conditional Sequential Model**
  - Combines features from both flat and sequential models.
  - Conditions sequence generation on a static context table.

2.3. Two-table Setup

In our project, we use the *Conditional Sequential Model*, also known as the two-table setup. This model combines a flat table and a context table, making it suitable for our task, where we work with a flat table (`Trips`) and a sequential table (`Positions`).

Table 2. Comparison Between Flat Table (Trips) and Sequential Table (Positions)

Table: `Trips`	**Table: `Positions`**
Flat Table	**Sequential Table**
`Trips` table is a *flat* table because it contains *time-independent data* — static information that does not change over time.	`Positions` table is a *sequential* table because it contains *time-dependent data* — dynamic information that changes over time.

The context processor component in the model integrates both tables, allowing the model to learn from both the historical sequence data and the static context provided by the flat table.

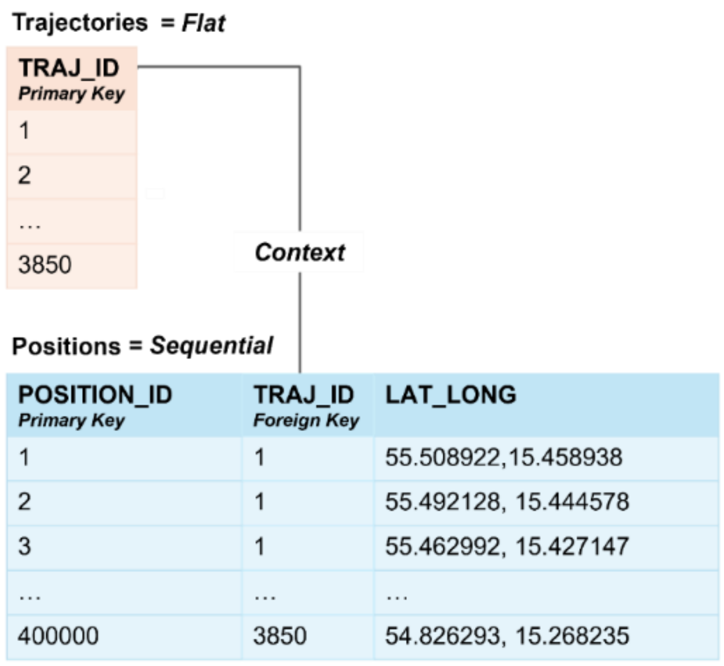
This two-table setup enables the model to generate more accurate predictions by conditioning not only on values from the current time step and past history but also on static, contextual information from the flat table.

Relationship

The two tables are connected through a *one-to-many relationship*, where one trip can have many positions. Both tables contain the column `TRIP_ID`. In the `Trips` table, `TRIP_ID` is the primary key, while in the `Positions` table, `TRIP_ID` serves as a foreign key that references the primary key in the `Trips` table. This structure ensures that each position record in the `Positions` table is associated with a specific trip in the `Trips` table.

Table 3. Comparison Between Subject Table (Trips) and Linked Table (Positions)

Table: `Trips`	**Table: `Positions`**
Subject Table	**Linked Table**
`Trips` table is a *subject* table because each row represents a unique entity: a single vessel trip. The table holds main information about each trip. For our experiment, this is limited to the `TRIP_ID`, though future work may include additional static trip details.	`Positions` table is a *linked* table because it is linked to the `Trips` table and provides additional information for each trip. The `Positions` table stores multiple position points, each with latitude and longitude coordinates, that represent the trip's path.
Every record has a unique `TRIP_ID`, which acts as the primary key.	Linked tables consist of records connected to entries in the subject table. Each linked record includes a foreign key that references the unique ID of the subject table.



TRAJ_ID *Primary Key*
1
2
…
3850

POSITION_ID *Primary Key*	TRAJ_ID *Foreign Key*	LAT_LONG
1	1	55.508922,15.458938
2	1	55.492128, 15.444578
3	1	55.462992, 15.427147
…	…	…
400000	3850	54.826293, 15.268235

Figure 4. Two-table setup: the flat Trajectories table and the sequential Positions table

Although the `Positions` table does not contain a literal timestamp column, the `POSITION_ID` is ordered chronologically, reflecting the sequence of positions over time. A separate model is trained for each table. For training the flat table, the model is trained using only the flat table itself. However, for training the model for the sequential table, the flat table is provided as context through the context processor.

2.4. Experiment

In this experiment, we use AIS data from Danish waters, obtained from a dataset published by (Olesen, Christensen and Clemmensen 2023). The dataset is available through the website of the Technical University of Denmark.[1] For this experiment, we use a subset of the data.[2]

Pre-processing

A key step in the pre-processing is *downsampling*, where we reduce the number of position points in each trajectory. Specifically, a position point is retained only if its distance from the previous point exceeds 1 kilometer. This step is motivated by earlier experiments, which showed that having too many position points per trip made it difficult for the model to effectively learn and generate realistic trajectories. Having too much data made it difficult for the generator to learn effectively, while downsampling resulted in improvements in the quality of the synthetic trajectories. Another step in the pre-processing is combining the latitude and longitude columns into a single column called `LAT_LNG`, which stores the latitude and longitude as a string separated by a comma. This format is required to make it compatible with the TabularARGN framework.

Input data

For this experiment, we use 3,850 vessel trajectories derived from real AIS data. After cleaning and pre-processing, the dataset consists of approximately 40,000 position rows. The trajectories are visualized in Figure 5, where distinct patterns of frequently traveled routes in Danish waters can be observed, highlighting the most common paths vessels follow in the region. It is important to note that the trajectories end in the middle of the sea because the input AIS data is limited to a specific rectangular area.

[1] Dataset: https://data.dtu.dk/collections/AIS_Trajectories_from_Danish_Waters_for_Abnormal_Behavior_Detection/6287841

[2] Link to download subset of the data: https://data.dtu.dk/ndownloader/files/38129544

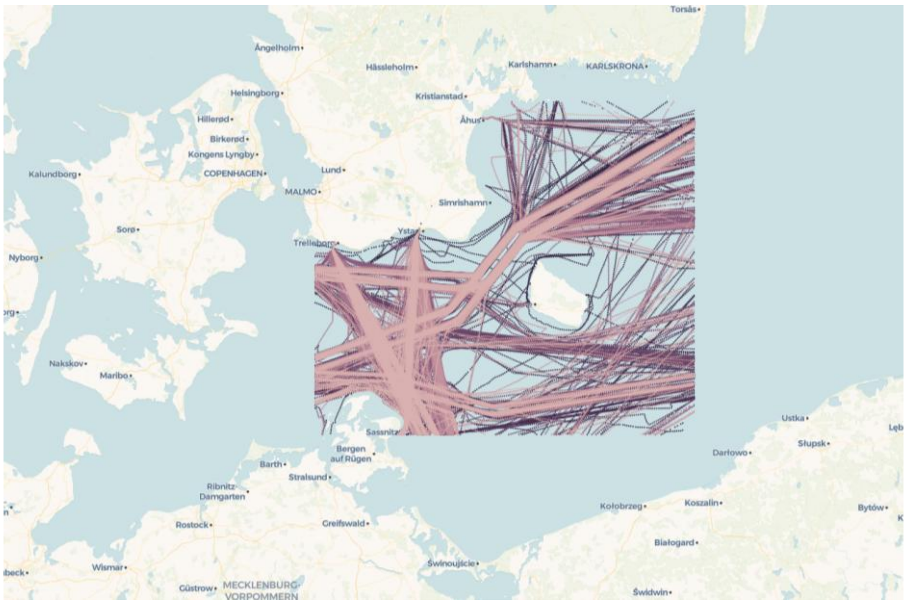


Figure 5. Input training dataset consisting of 3850 real vessel trajectories

Setup

The TabularARGN framework was implemented by the original authors and published as an open-source Python package called `mostlyai`. For our work, we use this package. Specifically, we use the *local GPU* version to ensure all data remains on our machine while leveraging GPU acceleration for efficient model training. The package is installed using pip with the following command: `pip install 'mostlyai[local-gpu]'`. At the time of writing, the local GPU version of `mostlyai` is only supported on Linux. Therefore, we set up our virtual environment on a Linux system.

Implementation

The complete implementation of the approach can be found in *Appendix B.* Using the `mostlyai` package makes the implementation straightforward, allowing it to be done in just a few lines of code. To train a synthetic data generator for vessel trip data, we use the `mostly.train` function from the `mostlyai` package with a custom configuration that defines two tables: `trips` and `positions`. The `trips` table contains a single column named `TRIP_ID`, which is set as the primary key. For this table, we specify the base model as `MOSTLY_AI/Large`. The `positions` table includes three columns: `POSITION_ID`, `TRIP_ID`, and `LAT_LONG`. Here, `LAT_LONG` is explicitly encoded with the type `TABULAR_LAT_LONG` to ensure the column is treated as geospatial data. The primary key for the `positions` table is `POSITION_ID`. We define a foreign key on `TRIP_ID` referencing the

`TRIP_ID` column in the `trips` table, with the `is_context` flag set to `True`. This informs the model to consider the relational context between the two tables during training. The base model for the `positions` table is also set to `MOSTLY_AI/Large`.

Synthetic Data Generation

After training the synthetic data generator, we use the `mostly.probe` function to generate synthetic samples. Specifically, we query the trained generator to produce 100 representative synthetic records, which are then exported as a CSV file. We can also save the generator to a file for later use, allowing it to be shared with others. An existing generator can be loaded at any time to generate new samples without the need for retraining.

The results of the synthetic data generation can be explored by visualizing the generated data in the same manner as the input training dataset. Figure 6 shows 100 synthetic trajectories produced by the trained model. Since the model was trained to learn the patterns in the input data, we observe that the synthetic data follows these same patterns, reflecting the characteristics and structure of the original vessel trajectories. This demonstrates the effectiveness of the generator in replicating the patterns and behaviors present in the real AIS data. It is important to note that the trajectories end in the middle of the sea because the input AIS data is limited to a specific rectangular area, as shown in Figure 5. If we show the vessel trajectories, it becomes clear that they look realistic, although further research is needed to assess how realistic they are and to measure their quality.

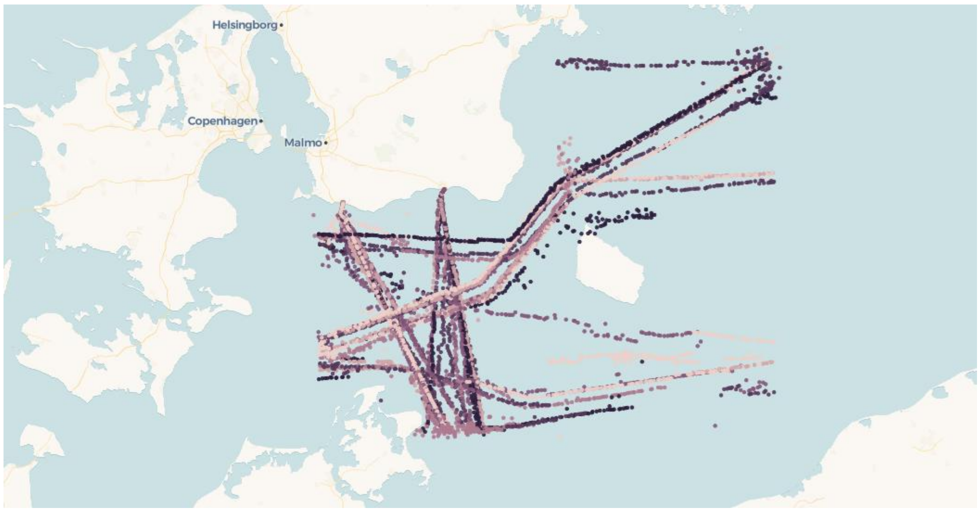


Figure 6. Map showing 100 synthetic trajectories generated by the trained model.

3. Conclusions

In conclusion, the approach of using an autoregressive model to generate synthetic vessel trajectories has both strengths and limitations. It is particularly effective at producing realistic trajectories, which is a major advantage. Additionally, it does not require manual rule creation or hardcoded logic. However, one key limitation is that the model depends heavily on having a large dataset for training. Its performance is directly affected by the quality of the input data. If the data is incomplete, noisy, or limited, the model may not produce reliable results.

To address this, one recommendation is to combine different data sources to create an input training dataset. For example, MARIT-D is conducting pilot tests at sea where they manually collect data from different scenarios, and this data could be included in the training set. Using an autoregressive model, the experimental data from these pilots can be expanded. It may also be helpful to combine this with publicly available AIS data to ensure there is enough data for effective model training.

Overall, while the approach shows good potential in generating realistic synthetic vessel trajectories, further work is needed to understand how it can be used to generate trajectories that follow certain behavioral patterns or styles described in text (modus operandi).

4. References

Kuffner, J.J. and LaValle, S.M. 2000. "RRT-connect: An efficient approach to single-query path planning." *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065).* https://ieeexplore.ieee.org/document/844730.

LaValle, S. 1998. "Rapidly-exploring random trees: A new tool for path planning." *Research Report 9811.* https://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf.

Olesen, Kristoffer Vinther, Anders Nymark Christensen, and Line Katrine Harder Clemmensen. 2023. *AIS Trajectories from Danish Waters for Abnormal Behavior Detection.* https://data.dtu.dk/collections/AIS_Trajectories_from_Danish_Waters_for_Abnormal_Behavior_Detection/6287841.

Tiwald, Paul, Ivona Krchova, Andrey Sidorenko, Mariana Vargas Vieyra, Mario Scriminaci, and Michael Platzer. 2025. "TabularARGN: A Flexible and Efficient Auto-Regressive Framework for." https://arxiv.org/pdf/2501.12012.

Appendix A. Encoding Input Data as Categorical Variables

The TabularARGN framework supports various data types such as categorical, numerical, datetime, string, and geospatial. However, its models operate exclusively on categorical columns. All other data types are converted into one or more categorical sub-columns using specific encoding strategies provided by the framework.

A.1 Numeric-discrete values

The columns `POSITION_ID` and `TRIP_ID` in the `Position` table, as well as the `TRIP_ID` column in the `Trips` table, are all numeric-discrete values. In TabularARGN, each unique numerical value is treated as a distinct category during conversion to categorical format.

A.2 Geospatial data

The `LAT_LONG` column in the `Position` table contains geospatial data. In TabularARGN, the two numeric coordinates are internally converted into a character sequence of quadtiles.[3] To explain the encoding process from coordinates to a quadtile, we provide an example.

Consider a position with the following `LAT_LNG` coordinates stored as a single string:

```
LAT_LNG = "51.68834629893225, 5.2985024395888285"
```

The first step in the process is to split this string into its individual components. Each coordinate is then stored as a floating-point number:

```
latitude = 51.68834629893225
longitude = 5.2985024395888285
```

Next, we represent these coordinates as integers by removing the decimal points.

```
latitude  = 5168834629893225
longitude = 52985024395888285
```

Both the latitude and longitude will then be converted into binary representations, resulting in the following:

```
latitude  = 00000100111011011110110000010
longitude = 00000000100000010101101111010
```

[3] https://wiki.openstreetmap.org/wiki/QuadTiles

Next, the *interleave* step is performed. This involves combining the two binary strings by alternating their bits one by one. Specifically, one bit is taken from the latitude binary string, followed by one bit from the longitude binary string, then the next bit from the latitude, then the next bit from the longitude, and so forth, continuing until all bits are combined. The result of the interleaving is:

```
interleaved = 00000000001000001110100010100011101110011110010101001100
```

The final step involves a bit-pair lookup, where the interleaved binary string is processed two bits at a time. Each pair of bits is mapped to a corresponding symbol according to the following mapping:

```
("0", "0") → "A" (North-West)
("1", "0") → "B" (North-East)
("0", "1") → "C" (South-West)
```

("1", "1") ⟶ "D" (South-East)

To perform this, the interleaved binary string is split into consecutive pairs of bits. Each pair is then translated into its respective symbol based on the mapping above. The resulting quadtile is:

```
quadtile = AAAAABAADBBABBADBDBCDBCCCADA
```

A.3 Visual representation of interleaving and bit-pair lookup

A visual representation of converting binary latitude and longitude values into a quadtile is shown in Figure 7. The top part shows interleaving of latitude and longitude bits, while the bottom part illustrates bit-pair lookup using a predefined mapping.

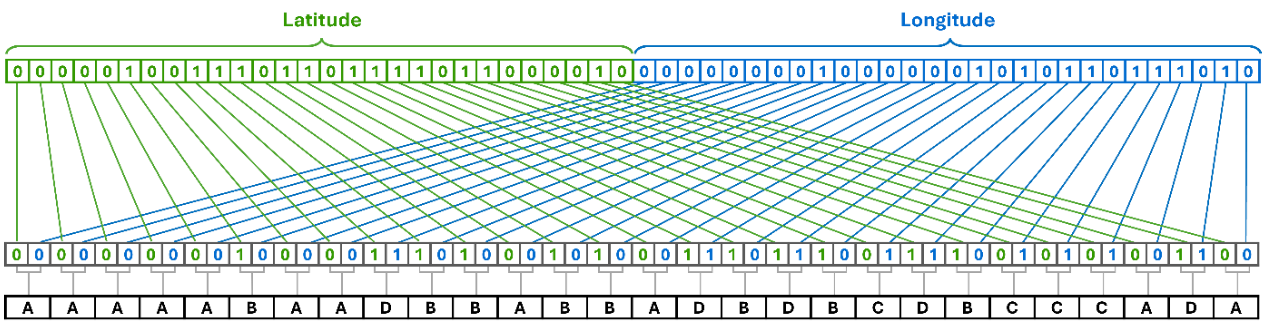


Figure 7. Visual representation of converting binary latitude and longitude values into quadtile.

Appendix B. Implementation for Autoregressive Model

```
import pandas as pd
from mostlyai.sdk import MostlyAI

# Read input data
positions_df = pd.read_csv("positions.csv")
trips_df = pd.read_csv("trips.csv")

# Initialize SDK
mostly = MostlyAI(local=True)

# Train a synthetic data generator
g = mostly.train(
    config={
        "name": "Vessel Trip Generator",
        "tables": [
            {
                "name": "trips",
                "data": trips_df,
                "tabular_model_configuration": {
                    "model": "MOSTLY_AI/Large"
                },
                "primary_key": "TRIP_ID",
                "columns": [{"name": "TRIP_ID"}]
            },
            {
                "name": "positions",
                "data": positions_df,
                "tabular_model_configuration": {
                    "model": "MOSTLY_AI/Large"
                },
                "columns": [
                    {"name": "POSITION_ID"},
                    {"name": "TRIP_ID"},
                    {
                        "name": "LAT_LONG",
                        "model_encoding_type": "TABULAR_LAT_LONG"
                    }
                ],
                "primary_key": "POSITION_ID",
                "foreign_keys": [
                    {
                        "column": "TRIP_ID",
                        "referenced_table": "trips",
                        "is_context": True,
                    }
                ]
            }
        ]
    }
)

# Probe generator for 100 synthetic samples
syn_samples_df = mostly.probe(g, size=100)
```