TDT4173 Report - Group 116

Johan Solbakken
Department of Computer Science, NTNU
Trondheim, Norway
Email: johsol@stud.ntnu.no

Morten Tobias Rinde Sunde
Department of Computer Science, NTNU
Trondheim, Norway
Email: mtsunde@stud.ntnu.no

December, 2024

Contents

1 Introduction

This report summarizes the steps taken in our project for the TDT4173 course. We discuss our exploratory data analysis, the feature engineering process, the various models we explored, and reflections on their effectiveness.

Our team name was *[116] Kvorum AS*, and our efforts culminated in a score of **115.8** on Kaggle.

2 Initial attempt

For our first attempt, we focused on building a pipeline that would run end-to-end and produce predictions for submission on Kaggle. We conducted no exploratory data analysis or detailed feature engineering at this stage. The goal was to ensure that we had functional code capable of making predictions and obtaining a preliminary score in the competition. The following Python code outlines our initial approach:

```python
import pandas as pd
import numpy as np
from datetime import datetime
from geopy.distance import geodesic
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.model_selection import train_test_split

def calculate_distance(lat1, lon1, lat2, lon2):
    return geodesic((lat1, lon1), (lat2, lon2)).kilometers

def prepare_sequences(data, feature_cols, target_cols, sequence_length):
    X, y = [], []
    data_values = data[feature_cols + target_cols].values
    for i in range(len(data_values) - sequence_length):
        X.append(data_values[i:i+sequence_length, :len(feature_cols)])
        y.append(data_values[i+sequence_length-1, len(feature_cols):])
    return np.array(X), np.array(y)

def prepare_test_sequences(data, feature_cols, sequence_length):
    X = []
    data_values = data[feature_cols].values
    for i in range(len(data_values) - sequence_length + 1):
        X.append(data_values[i:i+sequence_length])
    return np.array(X)

def prepare_test_data(ais_test, vessels, vessel_type_categories):
    merged_test = pd.merge(ais_test, vessels[['vesselId', 'vesselType']], on='vesselId', how='left')
    merged_test['vesselType'].fillna('Unknown', inplace=True)
    merged_test['time'] = pd.to_datetime(merged_test['time'])
    merged_test['hour'] = merged_test['time'].dt.hour
    merged_test['day_of_week'] = merged_test['time'].dt.dayofweek
    merged_test['vesselType'] = pd.Categorical(merged_test['vesselType'],
    ↪ categories=vessel_type_categories)
    merged_test['vessel_type_encoded'] = merged_test['vesselType'].cat.codes
    features = ['hour', 'day_of_week', 'vessel_type_encoded']
    merged_test[features] = merged_test[features].fillna(0)
    return merged_test, features

# Load and prepare data
ais_train = pd.read_csv('ais_train.csv', sep='|')
ais_test = pd.read_csv('ais_test.csv')
vessels = pd.read_csv('vessels.csv', sep='|')

merged_data = pd.merge(ais_train, vessels[['vesselId', 'vesselType']], on='vesselId', how='left')
merged_data['vesselType'].fillna('Unknown', inplace=True)
merged_data['time'] = pd.to_datetime(merged_data['time'])
merged_data['hour'] = merged_data['time'].dt.hour
merged_data['day_of_week'] = merged_data['time'].dt.dayofweek
merged_data['future_latitude'] = merged_data.groupby('vesselId')['latitude'].shift(-1)
merged_data['future_longitude'] = merged_data.groupby('vesselId')['longitude'].shift(-1)
merged_data.dropna(subset=['future_latitude', 'future_longitude'], inplace=True)

merged_data['vesselType'] = merged_data['vesselType'].astype('category')
vessel_type_categories = merged_data['vesselType'].cat.categories
merged_data['vessel_type_encoded'] = merged_data['vesselType'].cat.codes
features = ['hour', 'day_of_week', 'vessel_type_encoded']
target = ['future_latitude', 'future_longitude']

train_data, val_data = train_test_split(merged_data, test_size=0.2, shuffle=False)
sequence_length = 5
X_train, y_train = prepare_sequences(train_data, features, target, sequence_length)
X_val, y_val = prepare_sequences(val_data, features, target, sequence_length)
```

```
# Build and train the model
model = Sequential()
model.add(LSTM(64, return_sequences=False, input_shape=(sequence_length, len(features))))
model.add(Dense(2))
model.compile(optimizer='adam', loss='mean_absolute_error')

model.fit(X_train, y_train, epochs=1, validation_data=(X_val, y_val))

# Predictions and error calculation
predictions_val = model.predict(X_val)
val_data = val_data.iloc[sequence_length:]
val_data['pred_latitude'] = predictions_val[:, 0]
val_data['pred_longitude'] = predictions_val[:, 1]
val_data['error_distance'] = val_data.apply(lambda row: calculate_distance(
    row['future_latitude'], row['future_longitude'], row['pred_latitude'], row['pred_longitude']), axis=1)
mean_error_distance = val_data['error_distance'].mean()
print(f'Mean Geodetic Error: {mean_error_distance} km')

# Prepare submission data
merged_test, features = prepare_test_data(ais_test, vessels, vessel_type_categories)
X_test = prepare_test_sequences(merged_test, features, sequence_length)
predictions_test = model.predict(X_test)

# Save predictions
prediction_indices = np.arange(sequence_length - 1, len(merged_test))
submission_df = merged_test.iloc[prediction_indices].copy()
submission_df['longitude_predicted'] = predictions_test[:, 1]
submission_df['latitude_predicted'] = predictions_test[:, 0]
submission_df['ID'] = ais_test.iloc[prediction_indices]['ID'].values
submission_df = submission_df[['ID', 'longitude_predicted', 'latitude_predicted']]
submission_df.to_csv('submission.csv', index=False)
```

This code handled data preparation, model training, and prediction. It included functions for preparing sequences of features and targets and predicting future vessel locations based on past data. We utilized an LSTM model with a single layer and trained it on sequences of time and vessel-related features to predict latitude and longitude.

Although the primary objective of this attempt was to produce a working model, the results helped us understand the challenges and set a foundation for further improvements. The geodetic error calculated during validation gave us a rough measure of prediction accuracy, which provided valuable feedback for future iterations.

This model scored 600 on Kaggle, which we thought was a good start. However, it became clear that conducting an exploratory analysis and applying feature engineering would be necessary to create a model with better predictive performance.

3 Exploratory Analysis

We found that we had to do some exploratory analysis of the data. Our previous attempt suggested that there probably was a lot of noise in the data. To investigate this, we plotted the data using `matplotlib` to identify which parts of the dataset should be cleaned up.

3.1 Correlation Analysis

In the initial phase of our exploratory data analysis, we performed a correlation analysis on the ais_train dataset. The code below illustrates our approach:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Select only numeric columns for correlation matrix
numeric_columns = ais_train.select_dtypes(include=['float64', 'int64'])

# Calculate the correlation matrix
corr_matrix = numeric_columns.corr()

# Plot the heatmap
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm')
plt.show()
```

This gave us a heatmap representation of the correlation between the numerical features in the dataset 1. From this, we could observe the following correlations:

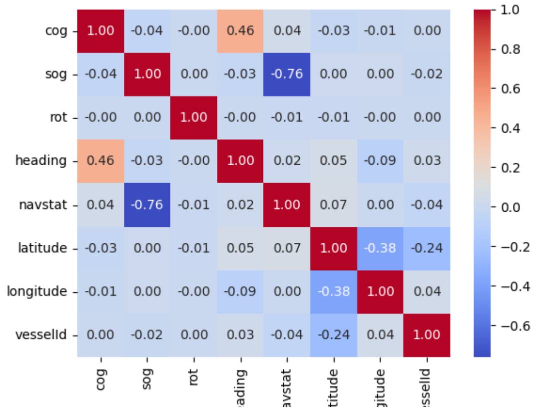

Figure 1: Correlation `ais_train.csv`

- **COG** (Course Over Ground) and **heading**: A moderate positive correlation (0.46), which suggests that vessels with higher COG tend to have a higher heading, as expected, since these slight directional metrics.
- **SOG** (Speed Over Ground) and **NAVSTAT** (Navigation Status): A strong negative correlation (-0.76), which indicates that vessels with specific NAVSTAT values (e.g., underway vs. anchored) have significantly different speed patterns. Vessels with NAVSTAT as 0 (Underway) will have a positive SOG, while other statuses will have lower or zero SOG. This will be interesting to investigate.
- **latitude** and **longitude**: There is a slight negative correlation between latitude and longitude (-0.38), which could imply geographical characteristics of the data, where locations tend to have this inverse relationship in this dataset's geographic region.
- **NAVSTAT** and **heading**: A slight negative correlation between these two (0.02), which might indicate that certain navigation statuses only marginally impact vessel heading.

Most features show weak or no correlation, except for the abovementioned cases. **vesselId** doesn't show strong correlations with other variables, which is expected since it's more of a categorical identifier.

3.2 Plotting SOG by NAVSTAT

Based on the findings from the previous section, we decided to investigate the relationship between SOG and NAVSTAT. In the dataset, SOG is represented as a floating-point number, capturing precise speed measurements, while NAVSTAT is stored as integer codes corresponding to specific navigational states. To make this data more interpretable, we mapped the integer values of NAVSTAT to their respective descriptive labels, as illustrated in Listing 1. This mapping was based on `api.vtexplorer.com`'s AIS (Automatic Identification System) documentation [1].

We thought a natural relationship between a vessel's SOG and NAVSTAT should exist. For example, if a ship is "At Anchor," it is logical to assume that its SOG would be zero, indicating that the vessel is stationary. Additionally, the problem description provided in the project mentioned that some ships might use more than one NAVSTAT code to describe their current activity. Specifically, most vessels tend to use code 0 ("Underway using engine") as their default but occasionally switch to code 8 ("Underway sailing") when transitioning to or from using sails. These assumptions suggested that plotting SOG against NAVSTAT could help us identify patterns or inconsistencies in the data. By visualizing the relationship between these two variables, we could understand how well the data aligns with our expectations. Listing 2 details the plot generating process based on the `ais_train` dataset.

The data loading code in Listing 3 enabled us to generate Figure 2, which presents the relationship between Speed Over Ground (SOG) and Navigational Status (NAVSTAT) across the entire dataset. A noteworthy observation from this plot is the presence of vessels with speeds reaching up to 100 knots while underway. Since container ships typically operate within the range of 16 to 24 knots [2], we classi-

```
navstat_mapping = {
    0: 'Under way using engine',
    1: 'At anchor',
    2: 'Not under command',
    3: 'Restricted maneuverability',
    4: 'Constrained by her draught',
    5: 'Moored',
    6: 'Aground',
    7: 'Engaged in Fishing',
    8: 'Under way sailing',
    9: 'Reserved for future amendment of Navigational Status for HSC',
    10: 'Reserved for future amendment of Navigational Status for WIG',
    11: 'Reserved for future use',
    12: 'Reserved for future use',
    13: 'Reserved for future use',
    14: 'AIS-SART is active',
    15: 'Not defined (default)'
}
```

Listing 1: NAVSTAT mapping

```
import matplotlib.pyplot as plt
import seaborn as sns

# load the ais_train

# Create a box plot to visualize the distribution of sog for each navstat description
plt.figure(figsize=(12, 6))
sns.boxplot(x='navstat_description', y='sog', data=filtered_data)
plt.xlabel('Navigation Status')
plt.ylabel('Speed Over Ground (sog)')
plt.title('Distribution of Speed Over Ground by Navigation Status')
plt.xticks(rotation=45, ha='right')  # Rotate labels for better readability
plt.tight_layout()  # Adjust layout to prevent label overlap
plt.show()
```

Listing 2: Plotting `ais_train` for SOG and NAVSTAT

fied these high-speed instances as noise. Furthermore, by inspecting the `vessels.csv` dataset, we observed that the recorded maximum, minimum, and average speeds for vessels with non-null `maxSpeed` values were 23.3, 16.7, and approximately 21.7 knots, respectively. This reinforced our assumption that the original speed data contained unrealistic values, likely due to sensor errors or misreporting.

```
"""
    Load Data
"""
# Read ais_train.csv
ais_train = pd.read_csv("ais_train.csv", sep='|')

# Map the numeric navstat values to their descriptive names
ais_train['navstat_description'] = ais_train['navstat'].map(navstat_mapping)

# Filter data to include only the navstat descriptions we're interested in (e.g., navstat < 6)
filtered_data = ais_train[ais_train['navstat'] < 6]
```

Listing 3: Loading the dataset

A second significant observation was that vessels reported as being "Moored" or "At Anchor" had non-zero speeds in several instances. Based on the assumption that moored or anchored ships should be stationary, with zero speed, we decided to clean the dataset by removing any records that violated this assumption. Specifically, we pruned the data to exclude instances where vessels were moored or anchored but had a reported speed greater than zero. Moreover, we identified several outliers where vessels were labeled as "Not Under Command" yet had SOG values exceeding 5 knots, which seemed implausible. These outliers were also removed as part of our data-cleaning process.

Furthermore, given that NAVSTAT code 8 ("Underway sailing") effectively conveys the same information as NAVSTAT code 0 ("Underway using engine"), we unified these categories by reassigning all data points with NAVSTAT 8 to NAVSTAT 0. This helped reduce redundancy in the dataset and ensured that our model would not be confused by the same maritime status represented by two different codes. We also removed NAVSTAT codes 6 through 15, which correspond to statuses like "On the ground," "Fishing," and "Reserved for future use," as these statuses were irrelevant to our analysis. The code used to perform this data pruning is shown in Listing 4. Figure 3 illustrates the improved plot after applying these pruning steps, with the SOG values for the "Underway using the engine," "At anchor," and "Moored" statuses now appearing much more reasonable.

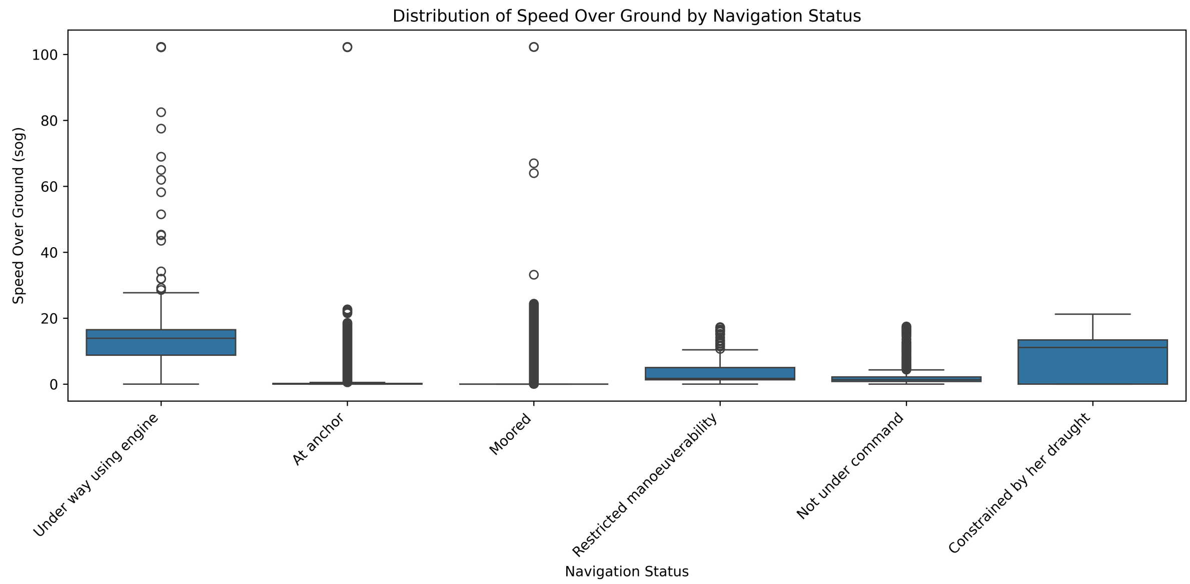

Figure 2: SOG over NAVSTAT plot of the entire dataset

```
"""
    Load Data
"""
# Read ais_train.csv
ais_train = pd.read_csv("ais_train.csv", sep='|')

# We assume that speed less than 25 knots are appropriate
ais_train = ais_train[(ais_train['sog'] < 25)]
ais_train['navstat'] = ais_train['navstat'].replace(8, 0) # every boat that is under sail is under way
ais_train = ais_train[~((ais_train['navstat'].isin([1, 5])) & (ais_train['sog'] > 0))] # standing
↪   still, should have 0 speed
ais_train = ais_train[~((ais_train['navstat'].isin([2])) & (ais_train['sog'] > 5))] # remove outliers
↪   for not under command:;
```

Listing 4: Loading data, pruning SOG and NAVSTAT outliers

3.3 Interpolating Vessel Routes

One challenge with the dataset was that the time intervals between recorded data points were highly inconsistent. Some data points were recorded only 5 seconds apart, while others were separated by as much as 13 days. Such significant discrepancies in time intervals posed a problem for trajectory analysis, as having evenly spaced data points is crucial for creating reliable predictions and models. To address this issue, we aimed to ensure that each vessel in the dataset had at least one recorded data point per day. This regularization of the time intervals would provide a more stable foundation for modeling. Inspired by the work in [3], we experimented with cubic spline interpolation to fill in the gaps between irregularly spaced data points. Cubic splines allow for smooth interpolation between known data points, making them suitable for creating a continuous trajectory from our discrete data.

To illustrate this, as an example, let us consider a specific vessel, identified by `61e9f3bfb937134a3c4bfe9f`. We can visualize the trajectory of this vessel using Python's `plotly` library, which allows for interactive data plotting. The code provided below demonstrates how we performed this visualization.

```
import pandas as pd
import numpy as np
import plotly.graph_objects as go

# Read ais_train.csv
ais_train = pd.read_csv("ais_train.csv", sep='|')

# Temporal features
ais_train['time'] = pd.to_datetime(ais_train['time'])
ais_train['elapsed_time'] = (ais_train['time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
ais_train['day_of_week'] = ais_train['time'].dt.dayofweek  # Monday=0, Sunday=6
ais_train['hour_of_day'] = ais_train['time'].dt.hour
ais_train = pd.get_dummies(ais_train, columns=['day_of_week', 'hour_of_day'], drop_first=True)
```

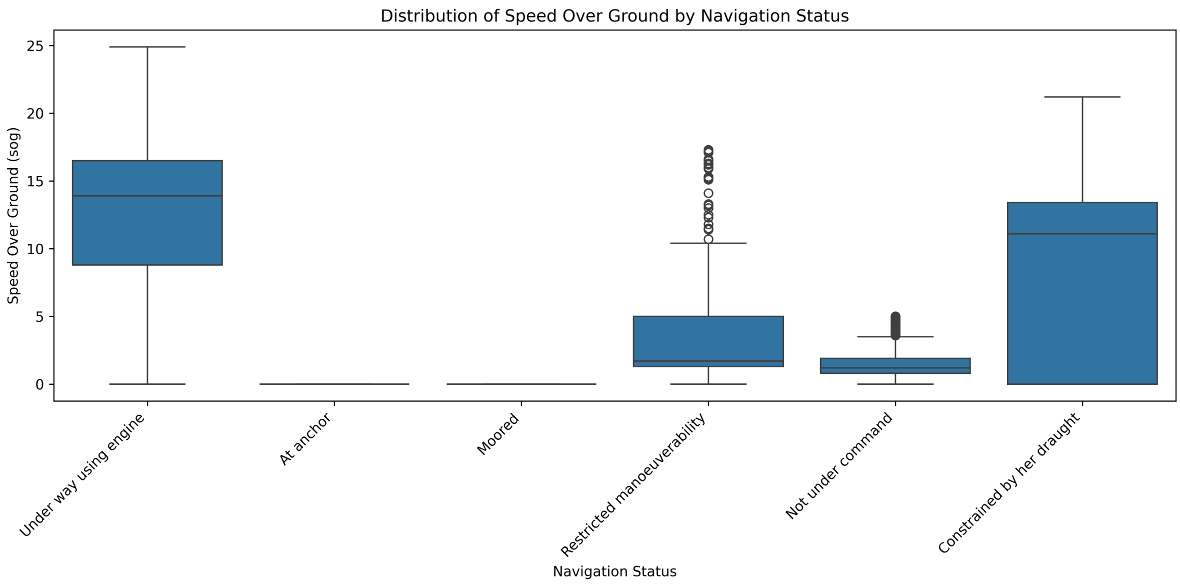

Figure 3: SOG over NAVSTAT after pruning

```python

# Filter out unrealistic speeds
ais_train = ais_train[ais_train['sog'] < 25]

# Map 'navstat' values
ais_train['navstat'] = ais_train['navstat'].replace(8, 0)  # Under way sailing -> Under way using engine
ais_train = ais_train[~((ais_train['navstat'].isin([1, 5])) & (ais_train['sog'] > 0))]
ais_train = ais_train[~((ais_train['navstat'] == 2) & (ais_train['sog'] > 5))]

# One-hot encode 'navstat'
ais_train = pd.get_dummies(ais_train, columns=['navstat'])

# Split cyclic values into x and y
ais_train['cog_sin'] = np.sin(np.radians(ais_train['cog']))
ais_train['cog_cos'] = np.cos(np.radians(ais_train['cog']))
ais_train['heading_sin'] = np.sin(np.radians(ais_train['heading']))
ais_train['heading_cos'] = np.cos(np.radians(ais_train['heading']))

# Merge with vessel data
vessels = pd.read_csv("vessels.csv", sep='|')[['shippingLineId', 'vesselId']]
vessels['new_id'] = range(len(vessels))
vessel_id_to_new_id = dict(zip(vessels['vesselId'], vessels['new_id']))
ais_train = pd.merge(ais_train, vessels, on='vesselId', how='left')

vesselId = "61e9f3bfb937134a3c4bfe9f"

# Filter the data for the selected vesselId and sort by 'time'
vessel_data = ais_train[ais_train['vesselId'] == vesselId].sort_values('time').reset_index(drop=True)

# Clean vessel_data
vessel_data = vessel_data.dropna(subset=['latitude', 'longitude'])

# Ensure that 'time' is datetime and sorted
vessel_data = vessel_data.sort_values('time')

# Convert 'time' to numeric format for interpolation (seconds since epoch)
vessel_data['time_numeric'] = vessel_data['time'].astype(np.int64) // 10**9

# Ensure that 'time_numeric' is strictly increasing
vessel_data = vessel_data.drop_duplicates(subset='time_numeric')
vessel_data = vessel_data.sort_values('time_numeric').reset_index(drop=True)
plotting_data = vessel_data.sort_values('time').reset_index(drop=True)

# Remove the last row (as it has NaN bearing)
plotting_data = plotting_data[:-1]

# Create the map figure
fig = go.Figure()

# Add the trajectory line
```

```
fig.add_trace(go.Scattermapbox(
    mode='lines+markers',
    lon=plotting_data['longitude'],
    lat=plotting_data['latitude'],
    marker=dict(size=6, color='blue'),
    line=dict(width=2, color='blue'),
    name='Trajectory'
))

fig.update_layout(
    mapbox={
        'style': "open-street-map",
        'zoom': 1 if interpolate else 4,
        'center': {'lon': plotting_data['longitude'].mean(), 'lat': plotting_data['latitude'].mean()}
    },
    title="Trajectory of vessel {vesselId} with Direction Arrows"
)

fig.show()
```

This script begins by reading the `ais_train.csv` file, which contains multiple vessels' raw AIS (Automatic Identification System) data. The first step is to convert the time column from its original format into Python's `datetime` format, making it easier to manipulate and plot time-series data. We then extracted the features necessary for our analysis, a process we had already refined in earlier stages of our project. Next, the script filters the dataset to isolate the data points corresponding to the vessel with `vesselId = "61e9f3bfb937134a3c4bfe9f"`. To avoid redundancy, we drop duplicate entries before plotting the vessel's trajectory using a map plot.

Figure 4 displays the result of plotting the uninterpolated data points for this vessel. Several key observations can be made from this plot. Firstly, there are instances where some data points are clustered closely together near land, reflecting periods where the vessel was likely stationary or moving slowly. In contrast, other data points are located far from each other, reflecting instances where the ship moved over large distances without frequent AIS updates. For example, the points with the longest distance between them were recorded ten days apart, which creates significant gaps in the vessel's trajectory. These gaps hinder our ability to accurately track the vessel's movements over time, especially for modeling purposes.

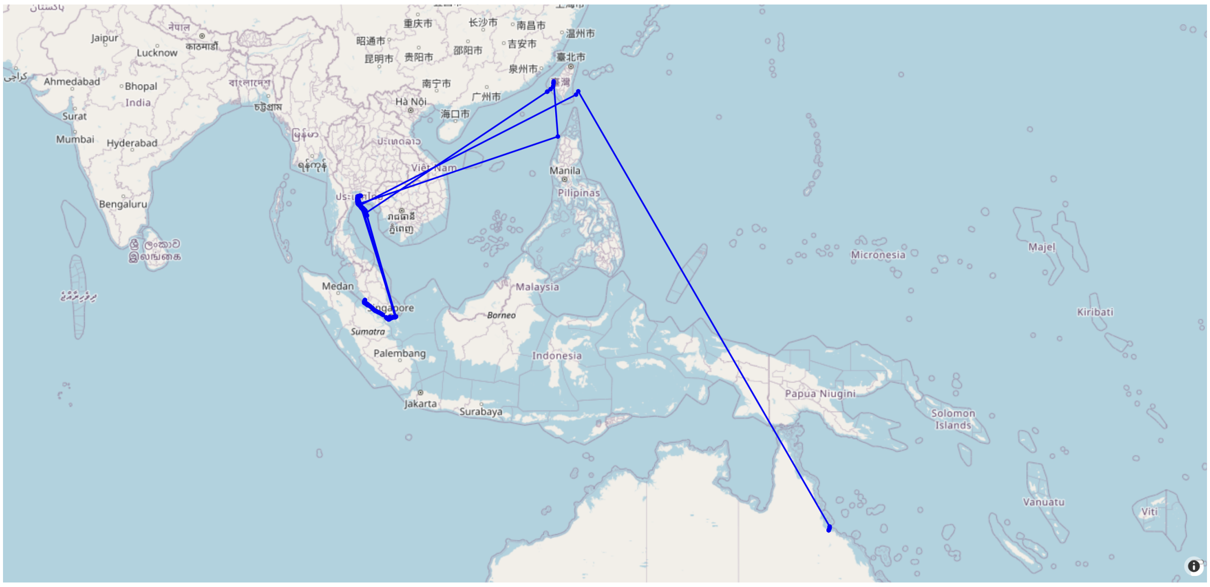

Figure 4: Uninterpolated trajectory of `vesselId=61e9f3bfb937134a3c4bfe9f`

To address this, we applied cubic spline interpolation using Python's `scipy` library. This method allowed us to create smooth interpolated paths between the known data points, effectively filling in the missing data while maintaining the continuity of the vessel's trajectory. Cubic spline interpolation is particularly useful for this type of data because it generates smooth curves that pass through the known points while accounting for the natural fluctuations in the vessel's movement over time.

```
from scipy.interpolate import CubicSpline
```

```python

# ... read and preprocess data

# Create a new time index with daily frequency
start_time = vessel_data['time'].min().normalize()
end_time = vessel_data['time'].max().normalize()
new_time_index = pd.date_range(start=start_time, end=end_time, freq='1D')

# Convert 'new_time_index' to numeric format
new_time_numeric = new_time_index.astype(np.int64) // 10**9

# Prepare original data for interpolation
original_times = vessel_data['time_numeric'].values
original_latitudes = vessel_data['latitude'].values
original_longitudes = vessel_data['longitude'].values

# Convert latitude and longitude to radians
lat_rad = np.radians(original_latitudes)
lon_rad = np.radians(original_longitudes)

# Convert to 3D Cartesian coordinates on a unit sphere
x = np.cos(lat_rad) * np.cos(lon_rad)
y = np.cos(lat_rad) * np.sin(lon_rad)
z = np.sin(lat_rad)

# Create cubic spline interpolators for x, y, z
cs_x = CubicSpline(original_times, x)
cs_y = CubicSpline(original_times, y)
cs_z = CubicSpline(original_times, z)

# Interpolate at new time points within the range of original_times
t_min = original_times.min()
t_max = original_times.max()
valid_mask = (new_time_numeric >= t_min) & (new_time_numeric <= t_max)
interp_times_valid = new_time_numeric[valid_mask]

if len(interp_times_valid) == 0:
    print("No valid interpolation times within the range of original data.")
    plotting_data = vessel_data
else:
    x_interp = cs_x(interp_times_valid)
    y_interp = cs_y(interp_times_valid)
    z_interp = cs_z(interp_times_valid)

    # Normalize the interpolated coordinates to lie on the unit sphere
    norm = np.sqrt(x_interp**2 + y_interp**2 + z_interp**2)
    x_interp /= norm
    y_interp /= norm
    z_interp /= norm

    # Convert back to latitude and longitude
    lat_interp = np.degrees(np.arcsin(z_interp))
    lon_interp = np.degrees(np.arctan2(y_interp, x_interp))

    # Create interpolated DataFrame
    vessel_data_interp = pd.DataFrame({
        'time_numeric': interp_times_valid,
        'latitude': lat_interp,
        'longitude': lon_interp
    })

    # Convert 'time_numeric' back to datetime
    vessel_data_interp['time'] = pd.to_datetime(vessel_data_interp['time_numeric'], unit='s')

    # Use the interpolated data for plotting
    plotting_data = vessel_data_interp.sort_values('time').reset_index(drop=True)

# ... plot data
```

Figure 5 shows the plot using cubic spline interpolation. In this plot, there is a noticeable improvement in the regularity of the data, given that the amount of data has increased by 140%, as now there is at least one data point for each day. The interpolation has filled the gaps between the previously uneven time intervals, creating a smoother trajectory for the vessel. However, despite this improvement, we needed to address some potential sources of error. Specifically, we observed that the interpolated data points occasionally took abrupt or unrealistic turns, with some interpolated paths even crossing over land, which cargo vessels obviously cannot do.

These unrealistic segments in the trajectory are likely due to the nature of cubic spline interpolation, which, while effective for smoothing data, does not account for geographical constraints such as coastlines or landmasses. Cubic splines interpolate based purely on the mathematical relationship between data points without considering the real-world implications of a vessel's path. As a result, the spline

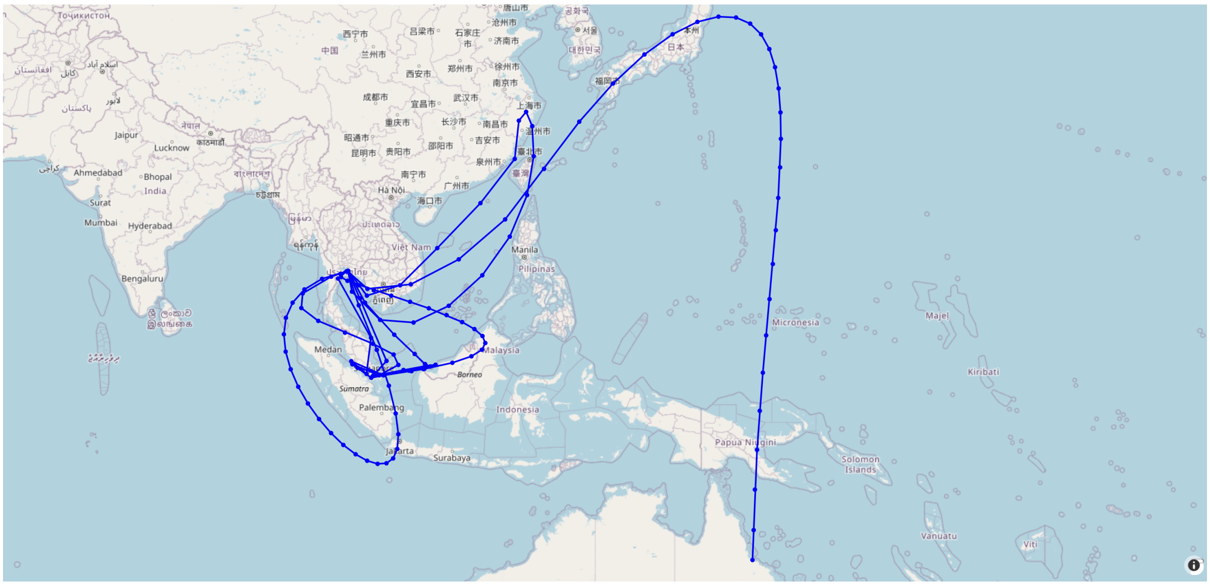

Figure 5: Interpolated trajectory of `vesselId=61e9f3bfb937134a3c4bfe9f`

interpolation can generate curves that deviate from expected sea routes, especially when there are significant gaps between the original data points or when the vessel's movement near land involves sharp directional changes.

This might be because some data points correspond to periods when the vessel was moored or anchored, meaning the ship was stationary. Interpolating movement data for these periods would be inappropriate, as the boat was not in motion.

3.4 Distribution of Time Gaps Between Consecutive AIS Records

We also wanted to investigate the time intervals apparent in the dataset, as we observed that AIS timestamps were inconsistent. To give ourselves an overview, we plotted the distribution of time gaps between consecutive AIS records (in seconds) on a logarithmic frequency scale 6. From this, we noticed the following:

Dominance of Small Gaps: The most significant number of time differences is concentrated around tiny gaps (near zero). This suggests that, for many records, the data points are closely spaced in time.

Exponential Decline: As the time difference increases, the frequency of occurrence decreases exponentially. This suggests that significant time gaps between consecutive AIS records are less common but still exist.

Long Tail Distribution: There is a long tail towards significant time gaps, reaching as high as 6 million seconds (approximately 69 days). While less frequent, there are still notable occurrences of considerable gaps in the dataset.

From this observation, it's clear that the time difference between data points needs to be accounted for. This allows us to create a model that effectively handles data points in close proximity as well as those with larger intervals.

4 Improved LSTM Model

The next step in our project was to implement our findings from the exploratory analysis into our initial LSTM model.

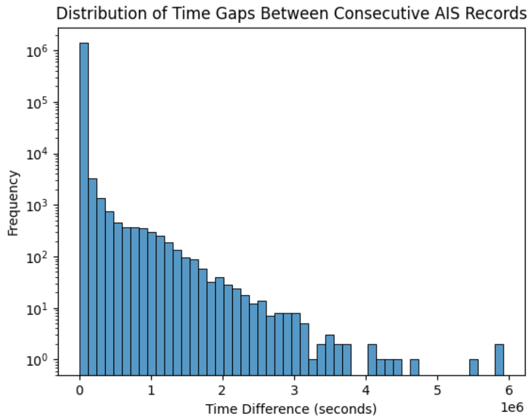

Figure 6: Distribution of Time Gaps Between Consecutive AIS Records

4.1 Feature Engineering

Blant others: This involved preprocessing the data:

- Temporal features were extracted from the AIS timestamps, including the elapsed time in seconds, day of the week, and hour of the day.
- Speed Over Ground (SOG) values were filtered to remove unrealistic speeds, and navigation status ('navstat') values were mapped to handle specific cases.
- 'cog' (course over ground) was transformed into cyclic features using sine and cosine encoding.
- One-hot encoding was applied to categorical features, and Min-Max scaling was used to normalize input and target features.

We also put much effort into transforming the raw AIS data into a structured format better suited for the model. First, we focused on extracting useful time-based features. We converted the `time` column into datetime objects and calculated `elapsed_time` as the number of seconds since the epoch to track vessel movements over time. We also pulled out `day_of_week` and `hour_of_day` from the timestamp and one-hot encoded them to capture weekly and daily patterns in vessel activity. For the COG feature, which is directional, we used sine and cosine transformations to avoid issues with its circular nature.

We mapped each to a unique index for the categorical variables, like **vesselId** and NAVSTAT. We used one-hot encoding to turn them into numbers that the model could understand. We cleaned up the data by filtering out combinations of NAVSTAT and SOG values that were unrealistic to co-occur. We also normalized all inputs and outputs with `MinMaxScaler` to help the model learn more effectively during training.

We added extra information by merging the dataset with external vessel data, including shippingLineId. Finally, we organized the data into sequential time steps for each vessel so the LSTM model could detect patterns over time.

To capture temporal dependencies, each vessel was assigned ten time-step sequences. This allowed the model to learn from sequences of consecutive historical data points. Each sequence was structured to predict the vessel's location at the next time step. The data was then split into training and validation sets.

4.2 The Model

The model consists of two bidirectional LSTM layers with 512 and 256 hidden units, respectively, followed by a dropout layer and a fully connected output layer. The bidirectional layers enable the model to learn patterns from forward and backward temporal data. The final output layer predicts the latitude and longitude coordinates.

- **Input Size:** Determined by the number of input features.

- **Hidden Layers:** Two bidirectional LSTM layers with 512 and 256 hidden units.
- **Output Size:** Set to 2, corresponding to latitude and longitude predictions.

We used a custom Haversine loss function, which calculates the distance between predicted and actual coordinates on the Earth's surface. This distance-based metric ensures that the model minimizes geodetic errors in location predictions, providing a more meaningful evaluation than traditional regression loss functions.

The model was trained for 100 epochs with early stopping to avoid overfitting. The Adam optimizer was used to update model weights based on the Haversine loss.

4.3 The Code

```python
#!/usr/bin/env python3

import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import copy

print("PyTorch version:", torch.__version__)

# Set device preference: MPS > CUDA > CPU
if torch.backends.mps.is_available():
    device = torch.device("mps")
    print("Using MPS device")
elif torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using CUDA device")
else:
    device = torch.device("cpu")
    print("Using CPU device")

"""
    Load and Preprocess Data
"""

# Read ais_train.csv
ais_train = pd.read_csv("ais_train.csv", sep='|')


vessel_mapping = {vessel: idx for idx, vessel in enumerate(ais_train['vesselId'].unique())}

# Temporal features
ais_train['time'] = pd.to_datetime(ais_train['time'])
ais_train['elapsed_time'] = (ais_train['time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')

# Filter out unrealistic speeds
ais_train = ais_train[ais_train['sog'] < 25]

# Map 'navstat' values
ais_train['navstat'] = ais_train['navstat'].replace(8, 0)  # Under way sailing -> Under way using
    engine
ais_train = ais_train[~((ais_train['navstat'].isin([1, 5])) & (ais_train['sog'] > 0))]
ais_train = ais_train[~((ais_train['navstat'] == 2) & (ais_train['sog'] > 5))]

# One-hot encode 'navstat'
ais_train = pd.get_dummies(ais_train, columns=['navstat'])

# Merge with vessel data
vessels = pd.read_csv("vessels.csv", sep='|')[['shippingLineId', 'vesselId']]
vessels['new_id'] = range(len(vessels))
vessel_id_to_new_id = dict(zip(vessels['vesselId'], vessels['new_id']))
ais_train = pd.merge(ais_train, vessels, on='vesselId', how='left')

# Temporal features
ais_train['day_of_week'] = ais_train['time'].dt.dayofweek
ais_train['hour_of_day'] = ais_train['time'].dt.hour
ais_train = pd.get_dummies(ais_train, columns=['day_of_week', 'hour_of_day'], drop_first=True)

# Handle cyclic features for 'cog'
ais_train['cog_sin'] = np.sin(np.radians(ais_train['cog']))
ais_train['cog_cos'] = np.cos(np.radians(ais_train['cog']))

# Merge with vessels and ports data
ais_train = pd.merge(ais_train, vessels, on='vesselId', how='left')
ais_train['vesselId'] = ais_train['vesselId'].map(vessel_mapping)
```

```python
# Define input and target features
input_features = [
    'latitude', 'longitude', 'sog', 'cog_sin', 'cog_cos', 'elapsed_time',
    'vesselId'
]

input_features.extend([col for col in ais_train.columns if 'day_of_week_' in col])
input_features.extend([col for col in ais_train.columns if 'hour_of_day_' in col])

target_columns = ['latitude', 'longitude']

# Initialize scalers
scaler_input = MinMaxScaler()
scaler_output = MinMaxScaler()

# Drop rows with NaN values in input features
ais_train = ais_train.dropna(subset=input_features + target_columns)

# Scale input and output features
input_data = scaler_input.fit_transform(ais_train[input_features])
output_data = scaler_output.fit_transform(ais_train[target_columns])

# Add scaled features back to DataFrame
ais_train[input_features] = input_data
ais_train[target_columns] = output_data

"""
    Create Sequences for Model Training
"""

# Function to create sequences per vessel
def create_sequences_per_vessel(df, time_steps):
    X, y = [], []
    vessel_ids = df['vesselId'].unique()
    for vessel_id in vessel_ids:
        vessel_data = df[df['vesselId'] == vessel_id].sort_values('elapsed_time')
        inputs = vessel_data[input_features].values
        targets = vessel_data[target_columns].values
        if len(inputs) < time_steps:
            continue  # Skip sequences shorter than time_steps
        for i in range(len(inputs) - time_steps):
            X.append(inputs[i:i + time_steps])
            y.append(targets[i + time_steps])
    return np.array(X), np.array(y)

# Create sequences
time_step = 10
X, y = create_sequences_per_vessel(ais_train, time_step)

# Split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, shuffle=True)

# Convert to PyTorch tensors
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).float()
X_val = torch.from_numpy(X_val).float()
y_val = torch.from_numpy(y_val).float()

# Create TensorDatasets and DataLoaders
batch_size = 128
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

"""
    Haversine Loss Function
"""

def haversine_loss(y_true, y_pred):
    R = 6371.0  # Earth radius in kilometers

    # Ensure constants are tensors of the same dtype and device as y_true
    pi_over_180 = torch.tensor(np.pi / 180.0, dtype=y_true.dtype, device=y_true.device)

    lat_true = y_true[:, 0] * pi_over_180
    lon_true = y_true[:, 1] * pi_over_180
    lat_pred = y_pred[:, 0] * pi_over_180
    lon_pred = y_pred[:, 1] * pi_over_180

    dlat = lat_pred - lat_true
    dlon = lon_pred - lon_true

    a = torch.sin(dlat / 2) ** 2 + torch.cos(lat_true) * torch.cos(lat_pred) * torch.sin(dlon / 2) ** 2
    c = 2 * torch.atan2(torch.sqrt(a), torch.sqrt(1 - a))
    distance = R * c
```

```python

    # Return mean distance over the batch
    return torch.mean(distance)

"""
    Define and Train the Model
"""

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(LSTMModel, self).__init__()
        # Set bidirectional=True and adjust hidden sizes accordingly
        self.lstm1 = nn.LSTM(
            input_size, hidden_size1, batch_first=True, bidirectional=True
        )
        self.lstm2 = nn.LSTM(
            hidden_size1 * 2, hidden_size2, batch_first=True, bidirectional=True
        )
        self.dropout = nn.Dropout(0.2)
        # Adjust the input size of the fully connected layer
        self.fc = nn.Linear(hidden_size2 * 2, output_size)

    def forward(self, x):
        out, _ = self.lstm1(x)
        out, _ = self.lstm2(out)
        out = self.dropout(out[:, -1, :])  # Get the last time step
        out = self.fc(out)
        return out

# Initialize the model
input_size = X_train.shape[2]
hidden_size1 = 512
hidden_size2 = 256
output_size = y_train.shape[1]
model = LSTMModel(input_size, hidden_size1, hidden_size2, output_size).to(device)

# Define optimizer
optimizer = optim.Adam(model.parameters())

# Training loop
num_epochs = 100
best_val_loss = float('inf')
patience = 5
counter = 0
best_model_wts = copy.deepcopy(model.state_dict())

for epoch in range(num_epochs):
    # Training
    model.train()
    train_losses = []

    for batch_X, batch_y in train_loader:
        batch_X = batch_X.to(device)
        batch_y = batch_y.to(device)

        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = haversine_loss(batch_y, outputs)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

    avg_train_loss = np.mean(train_losses)

    # Validation
    model.eval()
    val_losses = []
    with torch.no_grad():
        for batch_X, batch_y in val_loader:
            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)
            outputs = model(batch_X)
            loss = haversine_loss(batch_y, outputs)
            val_losses.append(loss.item())

    avg_val_loss = np.mean(val_losses)
    print(f"Epoch [{epoch + 1}/{num_epochs}], Train Loss: {avg_train_loss:.4f}, Val Loss:
    ↪ {avg_val_loss:.4f}")

    # Early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        best_model_wts = copy.deepcopy(model.state_dict())
        counter = 0
    else:
        counter += 1
        if counter >= patience:
```

```python
            print("Early stopping")
            break

# Load best model weights
model.load_state_dict(best_model_wts)

"""
    Prepare Test Data and Make Predictions
"""

# Load test data
ais_test = pd.read_csv("ais_test.csv")
ais_test['time'] = pd.to_datetime(ais_test['time'])
ais_test['elapsed_time'] = (ais_test['time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
ais_test['new_id'] = ais_test['vesselId'].map(vessel_id_to_new_id)

ais_test['vesselId'] = ais_test['vesselId'].map(vessel_mapping)
ais_test['day_of_week'] = ais_test['time'].dt.dayofweek
ais_test['hour_of_day'] = ais_test['time'].dt.hour

# One-hot encode
ais_test = pd.get_dummies(ais_test, columns=['day_of_week', 'hour_of_day'], drop_first=True)

# Merge with vessels and ports data
ais_test = pd.merge(ais_test, vessels, on='vesselId', how='left')

# Ensure all columns in ais_test match those in input_features
for col in input_features:
    if col not in ais_test.columns:
        ais_test[col] = 0

# Scale the test data using the same scaler
input_data_test = scaler_input.transform(ais_test[input_features])
ais_test_scaled = ais_test.copy()
ais_test_scaled[input_features] = input_data_test

# Prepare sequences for each vessel in the test set
def create_sequences_for_test(df_train, df_test, time_steps):
    X_test = []
    test_ids = []
    for idx, row in df_test.iterrows():
        vessel_id = row['vesselId']
        current_time = row['elapsed_time']

        # Get the historical data for this vessel up to the current_time
        vessel_train_data = df_train[df_train['vesselId'] == vessel_id]
        vessel_test_data = df_test[df_test['vesselId'] == vessel_id]

        # Combine and sort
        vessel_data = pd.concat([vessel_train_data, vessel_test_data], ignore_index=True)
        vessel_data = vessel_data.sort_values('elapsed_time')

        # Select data up to the current_time (excluding the current row)
        historical_data = vessel_data[vessel_data['elapsed_time'] < current_time]

        # Get the last 'time_steps' entries
        historical_sequence = historical_data.tail(time_steps)[input_features].values

        if len(historical_sequence) < time_steps:
            # Pad with zeros if not enough historical data
            padding = np.zeros((time_steps - len(historical_sequence), len(input_features)))
            historical_sequence = np.vstack([padding, historical_sequence])

        X_test.append(historical_sequence)
        test_ids.append(row['ID'])  # Assuming 'ID' is unique per test row

    return np.array(X_test), test_ids


# Create sequences for each test row
X_test, test_ids = create_sequences_for_test(ais_train, ais_test_scaled, time_step)

# Convert to PyTorch tensor
X_test = torch.from_numpy(X_test).float()

# Create a DataLoader for test data
test_dataset = TensorDataset(X_test)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

# Make predictions in batches
predictions = []

model.eval()
with torch.no_grad():
    for batch_X in test_loader:
        batch_X = batch_X[0].to(device)  # batch_X is a tuple
        outputs = model(batch_X)
```

```python
        predictions.append(outputs.cpu().numpy())

# Concatenate all batch predictions
y_pred = np.concatenate(predictions, axis=0)

# Inverse transform predictions
y_pred_inverse = scaler_output.inverse_transform(y_pred)


# Prepare submission
submission_df = pd.DataFrame({
    'ID': test_ids,
    'longitude_predicted': y_pred_inverse[:, target_columns.index('longitude')],
    'latitude_predicted': y_pred_inverse[:, target_columns.index('latitude')]
})

# Ensure the submission file has the required columns
submission_df = submission_df[['ID', 'longitude_predicted', 'latitude_predicted']]

# Save submission file
submission_df.to_csv("submission.csv", index=False)

# Display submission
print(submission_df.head())
print(f"Submission DataFrame shape: {submission_df.shape}")

print(f"Number of predictions: {len(y_pred_inverse)}")
print(f"Number of test IDs: {len(test_ids)}")
assert len(y_pred_inverse) == len(test_ids), "Mismatch between predictions and test IDs"
```

4.4 Result

After implementing the steps described in this section, we reduced our Kaggle score from 600 to 170.

4.5 Integrating Cubic Spline Interpolation into LSTM

As we discovered in our exploratory analysis, cubic spline interpolation could potentially smooth irregularities and fill in gaps within the dataset. Below is our implementation of the cubic spline interpolation integration.

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import copy
from scipy.interpolate import CubicSpline
from tqdm import tqdm

print("PyTorch version:", torch.__version__)

# Set device preference: MPS > CUDA > CPU
if torch.backends.mps.is_available():
    device = torch.device("mps")
    print("Using MPS device")
elif torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using CUDA device")
else:
    device = torch.device("cpu")
    print("Using CPU device")

"""
    Load and Preprocess Data
"""

# Read ais_train.csv
ais_train = pd.read_csv("ais_train.csv", sep='|')


vessel_mapping = {vessel: idx for idx, vessel in enumerate(ais_train['vesselId'].unique())}

# Temporal features
ais_train['time'] = pd.to_datetime(ais_train['time'])
ais_train['elapsed_time'] = (ais_train['time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')

# Filter out unrealistic speeds
ais_train = ais_train[ais_train['sog'] < 25]

```

```python
# Map 'navstat' values
ais_train['navstat'] = ais_train['navstat'].replace(8, 0)  # Under way sailing -> Under way using engine
ais_train = ais_train[~((ais_train['navstat'].isin([1, 5])) & (ais_train['sog'] > 0))]
ais_train = ais_train[~((ais_train['navstat'] == 2) & (ais_train['sog'] > 5))]

# One-hot encode 'navstat'
ais_train = pd.get_dummies(ais_train, columns=['navstat'])

# Merge with vessel data
vessels = pd.read_csv("vessels.csv", sep='|')[['shippingLineId', 'vesselId']]
vessels['new_id'] = range(len(vessels))
vessel_id_to_new_id = dict(zip(vessels['vesselId'], vessels['new_id']))
ais_train = pd.merge(ais_train, vessels, on='vesselId', how='left')

# Merge port data
ports = pd.read_csv("ports.csv", sep='|')[['portId', 'latitude', 'longitude']]
ports = ports.rename(columns={'latitude': 'port_latitude', 'longitude': 'port_longitude'})
ais_train = pd.merge(ais_train, ports, on='portId', how='left')
ais_train = ais_train[~ais_train['portId'].isnull()]  # Remove rows with null ports

def haversine_distance(lat1, lon1, lat2, lon2):
    # Earth radius in nautical miles
    R = 3440.065
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat / 2.0) ** 2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon / 2.0) ** 2
    return 2 * R * np.arcsin(np.sqrt(a))

def calculate_bearing(lat1, lon1, lat2, lon2):
    lat1_rad, lat2_rad = np.radians(lat1), np.radians(lat2)
    diff_long = np.radians(lon2 - lon1)
    x = np.sin(diff_long) * np.cos(lat2_rad)
    y = np.cos(lat1_rad) * np.sin(lat2_rad) - (np.sin(lat1_rad) * np.cos(lat2_rad) * np.cos(diff_long))
    initial_bearing = np.arctan2(x, y)
    return (np.degrees(initial_bearing) + 360) % 360

"""
    Cubic Spline Interpolation for Each Vessel
"""

# List to store processed trajectories
processed_trajectories = []

# Group data by vesselId
vessel_ids = ais_train['vesselId'].unique()
for vessel_id in tqdm(vessel_ids, desc="Interpolating Vessels"):

    vessel_data = ais_train[ais_train['vesselId'] == vessel_id].sort_values('elapsed_time')

    # Ensure at least two data points
    if len(vessel_data) < 2:
        continue

    # Prepare data for interpolation
    times = vessel_data['elapsed_time'].values
    latitudes = vessel_data['latitude'].values
    longitudes = vessel_data['longitude'].values

    # Remove duplicates in times
    times, unique_indices = np.unique(times, return_index=True)
    latitudes = latitudes[unique_indices]
    longitudes = longitudes[unique_indices]

    if len(times) < 2:
        continue

    # Convert lat/lon to radians
    lat_rad = np.radians(latitudes)
    lon_rad = np.radians(longitudes)

    # Convert to 3D Cartesian coordinates
    x = np.cos(lat_rad) * np.cos(lon_rad)
    y = np.cos(lat_rad) * np.sin(lon_rad)
    z = np.sin(lat_rad)

    # Create new time points for interpolation (every hour)
    start_time = times.min()
    end_time = times.max()
    new_times = np.arange(start_time, end_time + 1, 3600)  # Every hour in seconds
    # Ensure new_times are within the original times
    new_times = new_times[(new_times >= times.min()) & (new_times <= times.max())]

    # Interpolate x, y, z using cubic splines
    cs_x = CubicSpline(times, x)
    cs_y = CubicSpline(times, y)
```

```python
    cs_z = CubicSpline(times, z)

    x_interp = cs_x(new_times)
    y_interp = cs_y(new_times)
    z_interp = cs_z(new_times)

    # Normalize to unit sphere
    norm = np.sqrt(x_interp**2 + y_interp**2 + z_interp**2)
    x_interp /= norm
    y_interp /= norm
    z_interp /= norm

    # Convert back to lat/lon
    lat_interp = np.degrees(np.arcsin(z_interp))
    lon_interp = np.degrees(np.arctan2(y_interp, x_interp))

    # Handle longitude wrap-around
    lon_interp = (lon_interp + 360) % 360
    # Adjust longitudes > 180 to negative values (from -180 to 180)
    lon_interp[lon_interp > 180] -= 360

    # Create interpolated DataFrame
    interp_df = pd.DataFrame({
        'vesselId': vessel_id,
        'elapsed_time': new_times,
        'latitude': lat_interp,
        'longitude': lon_interp,
    })

    # Assign 'portId' using merge_asof
    vessel_port_data = vessel_data[['elapsed_time', 'portId']].drop_duplicates().sort_values('elapsed_time')
    interp_df = pd.merge_asof(
        interp_df.sort_values('elapsed_time'),
        vessel_port_data,
        on='elapsed_time',
        direction='nearest'
    )

    # Recalculate 'sog' and 'cog'
    lat_prev = np.roll(lat_interp, 1)
    lon_prev = np.roll(lon_interp, 1)
    time_prev = np.roll(new_times, 1)
    distances = haversine_distance(lat_prev, lon_prev, lat_interp, lon_interp)
    time_diffs = (new_times - time_prev) / 3600  # Convert time difference to hours
    time_diffs[0] = np.nan  # First element has no previous point
    sog = distances / time_diffs  # Speed in knots
    cog = calculate_bearing(lat_prev, lon_prev, lat_interp, lon_interp)
    cog[0] = np.nan  # First element has no previous point

    # Assign 'sog' and 'cog' to interp_df
    interp_df['sog'] = sog
    interp_df['cog'] = cog

    # Drop the first row as it has NaN values
    interp_df = interp_df.iloc[1:].reset_index(drop=True)

    # Convert 'elapsed_time' back to 'time'
    interp_df['time'] = pd.to_datetime(interp_df['elapsed_time'], unit='s')

    # Append to processed_trajectories
    processed_trajectories.append(interp_df)

# Combine all interpolated data
ais_train_interpolated = pd.concat(processed_trajectories, ignore_index=True)

print("ais_train", len(ais_train))
print("ais_train_interpolated", len(ais_train_interpolated))

"""
    Continue Preprocessing with Interpolated Data
"""

# Temporal features
ais_train_interpolated['day_of_week'] = ais_train_interpolated['time'].dt.dayofweek
ais_train_interpolated['hour_of_day'] = ais_train_interpolated['time'].dt.hour
ais_train_interpolated = pd.get_dummies(ais_train_interpolated, columns=['day_of_week', 'hour_of_day'], drop_first=True)

# Handle cyclic features for 'cog'
ais_train_interpolated['cog_sin'] = np.sin(np.radians(ais_train_interpolated['cog']))
ais_train_interpolated['cog_cos'] = np.cos(np.radians(ais_train_interpolated['cog']))

# Merge with vessels and ports data
ais_train_interpolated = pd.merge(ais_train_interpolated, vessels, on='vesselId', how='left')
ais_train_interpolated = pd.merge(ais_train_interpolated, ports, on='portId', how='left')

```

```python
# Calculate 'distance_to_port' and 'bearing_to_port'
ais_train_interpolated['distance_to_port'] = haversine_distance(
    ais_train_interpolated['latitude'], ais_train_interpolated['longitude'],
    ais_train_interpolated['port_latitude'], ais_train_interpolated['port_longitude']
)
ais_train_interpolated['bearing_to_port'] = calculate_bearing(
    ais_train_interpolated['latitude'], ais_train_interpolated['longitude'],
    ais_train_interpolated['port_latitude'], ais_train_interpolated['port_longitude']
)

ais_train_interpolated['vesselId'] = ais_train['vesselId'].map(vessel_mapping)

# Define input and target features
input_features = [
    'latitude', 'longitude', 'sog', 'cog_sin', 'cog_cos', 'elapsed_time',
    'distance_to_port', 'bearing_to_port', 'vesselId'
]
input_features.extend([col for col in ais_train_interpolated.columns if 'day_of_week_' in col])
input_features.extend([col for col in ais_train_interpolated.columns if 'hour_of_day_' in col])

target_columns = ['latitude', 'longitude']

# Initialize scalers
scaler_input = MinMaxScaler()
scaler_output = MinMaxScaler()

# Drop rows with NaN values in input features
ais_train_interpolated = ais_train_interpolated.dropna(subset=input_features + target_columns)

# Scale input and output features
input_data = scaler_input.fit_transform(ais_train_interpolated[input_features])
output_data = scaler_output.fit_transform(ais_train_interpolated[target_columns])

# Add scaled features back to DataFrame
ais_train_interpolated[input_features] = input_data
ais_train_interpolated[target_columns] = output_data

"""
    Create Sequences for Model Training
"""

# Function to create sequences per vessel
def create_sequences_per_vessel(df, time_steps):
    X, y = [], []
    vessel_ids = df['vesselId'].unique()
    for vessel_id in vessel_ids:
        vessel_data = df[df['vesselId'] == vessel_id].sort_values('elapsed_time')
        inputs = vessel_data[input_features].values
        targets = vessel_data[target_columns].values
        if len(inputs) < time_steps:
            continue  # Skip sequences shorter than time_steps
        for i in range(len(inputs) - time_steps):
            X.append(inputs[i:i + time_steps])
            y.append(targets[i + time_steps])
    return np.array(X), np.array(y)

# Create sequences
time_step = 10
X, y = create_sequences_per_vessel(ais_train_interpolated, time_step)

# Split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, shuffle=True)

# Convert to PyTorch tensors
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).float()
X_val = torch.from_numpy(X_val).float()
y_val = torch.from_numpy(y_val).float()

# Create TensorDatasets and DataLoaders
batch_size = 128
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

"""
    Haversine Loss Function
"""

def haversine_loss(y_true, y_pred):
    R = 6371.0  # Earth radius in kilometers

    # Ensure constants are tensors of the same dtype and device as y_true
    pi_over_180 = torch.tensor(np.pi / 180.0, dtype=y_true.dtype, device=y_true.device)

    lat_true = y_true[:, 0] * pi_over_180
```

```
    lon_true = y_true[:, 1] * pi_over_180
    lat_pred = y_pred[:, 0] * pi_over_180
    lon_pred = y_pred[:, 1] * pi_over_180

    dlat = lat_pred - lat_true
    dlon = lon_pred - lon_true

    a = torch.sin(dlat / 2) ** 2 + torch.cos(lat_true) * torch.cos(lat_pred) * torch.sin(dlon / 2) ** 2
    c = 2 * torch.atan2(torch.sqrt(a), torch.sqrt(1 - a))
    distance = R * c

    # Return mean distance over the batch
    return torch.mean(distance)

"""
    Define and Train the Model
"""

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(LSTMModel, self).__init__()
        # Set bidirectional=True and adjust hidden sizes accordingly
        self.lstm1 = nn.LSTM(
            input_size, hidden_size1, batch_first=True, bidirectional=True
        )
        self.lstm2 = nn.LSTM(
            hidden_size1 * 2, hidden_size2, batch_first=True, bidirectional=True
        )
        self.dropout = nn.Dropout(0.2)
        # Adjust the input size of the fully connected layer
        self.fc = nn.Linear(hidden_size2 * 2, output_size)

    def forward(self, x):
        out, _ = self.lstm1(x)
        out, _ = self.lstm2(out)
        out = self.dropout(out[:, -1, :])  # Get the last time step
        out = self.fc(out)
        return out

# Initialize the model
input_size = X_train.shape[2]
hidden_size1 = 512
hidden_size2 = 256
output_size = y_train.shape[1]
model = LSTMModel(input_size, hidden_size1, hidden_size2, output_size).to(device)

# Define optimizer
optimizer = optim.Adam(model.parameters())

# Training loop
num_epochs = 100
best_val_loss = float('inf')
patience = 5
counter = 0
best_model_wts = copy.deepcopy(model.state_dict())

for epoch in range(num_epochs):
    # Training
    model.train()
    train_losses = []

    for batch_X, batch_y in train_loader:
        batch_X = batch_X.to(device)
        batch_y = batch_y.to(device)

        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = haversine_loss(batch_y, outputs)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

    avg_train_loss = np.mean(train_losses)

    # Validation
    model.eval()
    val_losses = []
    with torch.no_grad():
        for batch_X, batch_y in val_loader:
            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)
            outputs = model(batch_X)
            loss = haversine_loss(batch_y, outputs)
            val_losses.append(loss.item())

    avg_val_loss = np.mean(val_losses)
```

```python
    print(f"Epoch [{epoch + 1}/{num_epochs}], Train Loss: {avg_train_loss:.4f}, Val Loss:
    ↪  {avg_val_loss:.4f}")

    # Early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        best_model_wts = copy.deepcopy(model.state_dict())
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping")
            break

# Load best model weights
model.load_state_dict(best_model_wts)

"""
    Prepare Test Data and Make Predictions
"""

# Load test data
ais_test = pd.read_csv("ais_test.csv")
ais_test['time'] = pd.to_datetime(ais_test['time'])
ais_test['elapsed_time'] = (ais_test['time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
ais_test['new_id'] = ais_test['vesselId'].map(vessel_id_to_new_id)

ais_test['vesselId'] = ais_test['vesselId'].map(vessel_mapping)
ais_test['day_of_week'] = ais_test['time'].dt.dayofweek
ais_test['hour_of_day'] = ais_test['time'].dt.hour

# One-hot encode
ais_test = pd.get_dummies(ais_test, columns=['day_of_week', 'hour_of_day'], drop_first=True)

# Merge with vessels and ports data
ais_test = pd.merge(ais_test, vessels, on='vesselId', how='left')

# Ensure all columns in ais_test match those in input_features
for col in input_features:
    if col not in ais_test.columns:
        ais_test[col] = 0

# Scale the test data using the same scaler
input_data_test = scaler_input.transform(ais_test[input_features])
ais_test_scaled = ais_test.copy()
ais_test_scaled[input_features] = input_data_test

# Prepare sequences for each vessel in the test set
def create_sequences_for_test(df_train, df_test, time_steps):
    X_test = []
    test_ids = []
    for idx, row in df_test.iterrows():
        vessel_id = row['vesselId']
        current_time = row['elapsed_time']

        # Get the historical data for this vessel up to the current_time
        vessel_train_data = df_train[df_train['vesselId'] == vessel_id]
        vessel_test_data = df_test[df_test['vesselId'] == vessel_id]

        # Combine and sort
        vessel_data = pd.concat([vessel_train_data, vessel_test_data], ignore_index=True)
        vessel_data = vessel_data.sort_values('elapsed_time')

        # Select data up to the current_time (excluding the current row)
        historical_data = vessel_data[vessel_data['elapsed_time'] < current_time]

        # Get the last 'time_steps' entries
        historical_sequence = historical_data.tail(time_steps)[input_features].values

        if len(historical_sequence) < time_steps:
            # Pad with zeros if not enough historical data
            padding = np.zeros((time_steps - len(historical_sequence), len(input_features)))
            historical_sequence = np.vstack([padding, historical_sequence])

        X_test.append(historical_sequence)
        test_ids.append(row['ID'])  # Assuming 'ID' is unique per test row

    return np.array(X_test), test_ids


# Create sequences for each test row
X_test, test_ids = create_sequences_for_test(ais_train_interpolated, ais_test_scaled, time_step)

# Convert to PyTorch tensor
X_test = torch.from_numpy(X_test).float()

# Create a DataLoader for test data
```

```python
test_dataset = TensorDataset(X_test)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

# Make predictions in batches
predictions = []

model.eval()
with torch.no_grad():
    for batch_X in test_loader:
        batch_X = batch_X[0].to(device)  # batch_X is a tuple
        outputs = model(batch_X)
        predictions.append(outputs.cpu().numpy())

# Concatenate all batch predictions
y_pred = np.concatenate(predictions, axis=0)

# Inverse transform predictions
y_pred_inverse = scaler_output.inverse_transform(y_pred)


# Prepare submission
submission_df = pd.DataFrame({
    'ID': test_ids,
    'longitude_predicted': y_pred_inverse[:, target_columns.index('longitude')],
    'latitude_predicted': y_pred_inverse[:, target_columns.index('latitude')]
})

# Ensure the submission file has the required columns
submission_df = submission_df[['ID', 'longitude_predicted', 'latitude_predicted']]

# Save submission file
submission_df.to_csv("submission.csv", index=False)

# Display submission
print(submission_df.head())
print(f"Submission DataFrame shape: {submission_df.shape}")

print(f"Number of predictions: {len(y_pred_inverse)}")
print(f"Number of test IDs: {len(test_ids)}")
assert len(y_pred_inverse) == len(test_ids), "Mismatch between predictions and test IDs"
```

This yielded a Kaggle score of 1347, which was quite a poor result. This indicated that we either needed a more advanced interpolation technique or that interpolation might not be the right approach for handling stationary intervals.

4.6 BiGRU

We also tried variations where we changed the LSTM layers with GRU in combinations with and without bidirectional layers. LSTM proved to be the best in all cases.

5 Darts and LightGBM

After spending considerable time on deep learning models, we explored time-series models. Specifically, we chose to try the Darts library with LightGBM.

5.1 Feature Engineering

For the Darts-model, a slighty different feature engineering-approach was used.

A critical step in feature engineering here is data resampling to an hourly frequency. By aggregating the `latitude` and `longitude` coordinates using the mean and applying cubic interpolation, the code addresses irregular time intervals and fills in missing values, resulting in uniformly spaced data points that enhance the model's ability to detect patterns. Creating Darts `TimeSeries` objects encapsulates these processed features, facilitating seamless integration with the `LightGBMModel` for time series forecasting.

To maintain consistency and relevance, the code filters the data to include only those vessel IDs in the training and testing sets, thereby focusing the model on vessels with available historical and predictive data. Grouping the training data by `vesselId` allows for vessel-specific processing, where each vessel's trajectory is sorted chronologically and set with `time` as the index.

Additionally, the code computes the forecasting horizon by calculating the time differences between the last training timestamp and each test timestamp in hours. This enables the model to predict future positions based on the temporal distance from the previously known data point. Using lag features, specifically setting `lags=48`, allows the LightGBM model to consider the past 24 hours of data when making predictions, capturing both short-term and longer-term dependencies in the vessel movements.

Moreover, by iterating over each vessel and fitting individual models, the feature engineering process ensures that vessel-specific characteristics and behaviors are accounted for, leading to more accurate and personalized predictions.

5.2 The Model

The primary model used for prediction was the XGBoost regressor, which was trained to predict longitude and latitude. Various hyperparameters were fine-tuned to improve the model's performance, including:

- **n_estimators:** Number of boosting rounds was 700.
- **learning_rate:** The learning rate was reduced to 0.02 to allow for more fine-grained updates to the model.
- **max_depth:** The maximum depth of trees was set to 9 to balance model complexity and overfitting.
- **regularization:** L1 (alpha) and L2 (lambda) regularization parameters were increased to 0.4 and 2.0, respectively, to control overfitting.

The model was evaluated using the Mean Squared Error (MSE) metric on the validation set. The following results were obtained:

- **XGBoost Validation MSE:** 1367.17 after hyperparameter tuning.
- **Shuffled Cross-Validation MSE:** 1488.46, indicating consistent performance across different data splits.
- **LightGBM Validation MSE:** 1824.10, showing that XGBoost outperformed LightGBM in this task.

5.3 Hyperparameter tuning

Here, we first tried `lags=24`, then `lags=48` and `lags=96`. We also played with setting `learning_rate` to 0.1 and 0.01 and `n_estimators` to 50 and 500. In the end, the default values for `learning_rate`, `n_estimators`, and `lags=48` worked out the best.

5.4 The Code

We decided to revisit using LightGBM in combination with Darts for our analysis. The central strategy involved creating individual time series for each vessel with minimal data preprocessing. This approach ultimately yielded a score of 159.8.

```python
import pandas as pd
import numpy as np
from darts import TimeSeries
from darts.models import LightGBMModel
import lightgbm as lgb

# Load ais_train.csv with separator '|'
train_df = pd.read_csv('ais_train.csv', sep='|')
train_df['time'] = pd.to_datetime(train_df['time'])

# Load ais_test.csv with separator ','
test_df = pd.read_csv('ais_test.csv', sep=',')
test_df['time'] = pd.to_datetime(test_df['time'])

# Use 'vesselId' instead of 'vessel_id'
# Select only vessel IDs that are in both train and test datasets
common_vessel_ids = set(train_df['vesselId']).intersection(set(test_df['vesselId']))
train_df = train_df[train_df['vesselId'].isin(common_vessel_ids)]

# Group the training data by vesselId
groups = train_df.groupby('vesselId')

# Initialize dictionaries to store TimeSeries objects and last training times
```

```python
timeseries_dict = {}
last_train_time = {}

# Process each vesselId group
for vessel_id, group_df in groups:
    # Sort on time
    group_df = group_df.sort_values('time')
    # Set index to time
    group_df = group_df.set_index('time')
    # Select features (latitude and longitude)
    features_df = group_df[['latitude', 'longitude']]
    # Resample data to hourly frequency with mean and linear interpolation
    features_df = features_df.resample('H').mean().interpolate(method='cubic')
    # Create Darts TimeSeries object
    ts = TimeSeries.from_dataframe(features_df, value_cols=['latitude', 'longitude'])
    # Store the TimeSeries object and last training time
    timeseries_dict[vessel_id] = ts
    last_train_time[vessel_id] = features_df.index.max()

# Initialize a dictionary to store predictions
predictions = {}

# Fit LightGBM models and predict for each TimeSeries object
for vessel_id, ts in timeseries_dict.items():
    # Get the last training time
    last_time = last_train_time[vessel_id]
    # Get test times for this vessel
    vessel_test_df = test_df[test_df['vesselId'] == vessel_id]
    test_times = vessel_test_df['time']
    # Compute the time differences in hours
    time_diffs = (test_times - last_time).dt.total_seconds() / 3600
    # Get the maximum forecast horizon needed
    max_n = int(np.ceil(time_diffs.max()))
    if max_n <= 0:
        continue  # Skip if no future times to predict
    # Initialize LightGBM model with lag parameters
    model = LightGBMModel(lags=48)
    # Fit the model
    model.fit(ts)
    # Predict up to the maximum horizon needed
    forecast = model.predict(max_n)
    # Store the forecast and last time
    predictions[vessel_id] = (forecast, last_time)

# Initialize a list to store submission rows
submission_rows = []

# Generate predictions for the submission file
for idx, row in test_df.iterrows():
    vessel_id = row['vesselId']
    test_time = row['time']
    test_id = row['ID']  # Assuming 'ID' column exists in test_df
    # Check if predictions are available for this vessel_id
    if vessel_id in predictions:
        forecast_ts, last_time = predictions[vessel_id]
        time_diff = (test_time - last_time).total_seconds() / 3600
        index = int(np.round(time_diff)) - 1  # Adjust index since forecast starts from last_time + 1 hour
        # Convert forecast_ts to DataFrame
        forecast_df = forecast_ts.pd_dataframe()
        # Check if index is within forecast horizon
        if 0 <= index < len(forecast_df):
            predicted_lat = forecast_df['latitude'].iloc[index]
            predicted_lon = forecast_df['longitude'].iloc[index]
        else:
            predicted_lat = np.nan
            predicted_lon = np.nan
    else:
        predicted_lat = np.nan
        predicted_lon = np.nan
    # Append the prediction to the submission list
    submission_rows.append({
        'ID': test_id,
        'longitude_predicted': predicted_lon,
        'latitude_predicted': predicted_lat
    })

# Create a submission DataFrame from the list
submission_df = pd.DataFrame(submission_rows)

# Save the submission file
submission_df.to_csv('submission.csv', index=False)

print(submission_df)
```

5.5 Result

The model performed reasonably well, representing our best results with LightGBM. However, we discovered that incorporating the additional preprocessing steps we found in our exploratory analysis worsened the model's performance. In particular, adding features such as speed over ground (SOG) and course over ground (COG) led to a decline in predictive accuracy. Furthermore, we found that cubic interpolation provided better results than linear interpolation when data processing. This was quite surprising.

6 Random Forest

After seeing an improvement in our score with a simpler model using Darts, we decided to experiment with a more traditional tree-based model: Random Forest.

6.1 Feature Engineering

We chose a slightly different approach for our feature engineering in our Random Forest-implmeentation. The new features we craeted to enhance the model's predictive capabilities were:

- `previous_lat`: Latitude at the previous timestamp.
- `previous_lon`: Longitude at the previous timestamp.
- `delta_time`: Time difference in seconds between the current and previous timestamps.

These features were generated using group-wise operations:

```
# Create 'previous_lat', 'previous_lon', and 'delta_time' in the training set
train['previous_lat'] = train.groupby('vesselId')['latitude'].shift(1)
train['previous_lon'] = train.groupby('vesselId')['longitude'].shift(1)
train['delta_time'] = train.groupby('vesselId')['time'].diff().dt.total_seconds()
```

We then removed any rows with missing values that result from the shift operation:

```
# Drop rows with missing values resulting from the shift operation
train = train.dropna(subset=['previous_lat', 'previous_lon', 'delta_time'])
```

For the test set, we initialized the new features with `NaN` values, which are populated during the prediction phase:

```
# Initialize 'previous_lat', 'previous_lon', and 'delta_time' in the test set
test['previous_lat'] = np.nan
test['previous_lon'] = np.nan
test['delta_time'] = np.nan
```

6.2 Feature Selection

To ensure the model focuses on the most relevant features, we performed feature selection by choosing the newly engineered features that capture the essential temporal and spatial information:

- `vesselId`: Encoded identifier for each vessel.
- `previous_lat` and `previous_lon`: Provide spatial context based on the vessel's last known position.
- `delta_time`: Captures the temporal interval between observations.

We then extracted each vessel's last known positions and timestamps from the training data, which serves as the starting point for making predictions in the test set:

```
# Retrieve last known positions from the training set
last_positions = train.groupby('vesselId').apply(lambda x: x.iloc[-1])[['vesselId', 'latitude',
↪  'longitude', 'time']]
last_positions = last_positions.set_index('vesselId')
```

6.3 The Model

Separate Random Forest Regressors are trained for latitude and longitude predictions using the engineered features and selected targets:

We iterate over each vessel in the test data to generate predictions. For each vessel, the process involves:

1. Verifying the vessel exists in the training data.
2. Retrieving the last known position and time.
3. For each timestamp in the test data:
   - Calculating the time difference from the last known time.
   - Preparing the feature vector for prediction.
   - Predicting the latitude and longitude.
   - Updating the previous position and time with the new predictions.
   - Storing the predictions.

6.4 Hyperparameter Tuning

We first tried with `lags=1`, `n_estimators` of 50. This proved to give us the highest score overall. We also tried `n_estimators` of 100 and `lags=2`, but these combinations worsened the model.

6.5 The Code

```python
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor

# Load training data
train = pd.read_csv('ais_train.csv', sep='|')

# Load test data
test = pd.read_csv('ais_test.csv', sep=',')

# Convert 'time' column to datetime
train['time'] = pd.to_datetime(train['time'])
test['time'] = pd.to_datetime(test['time'])

# Map 'vesselId' to unique integers
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
train['vesselId'] = le.fit_transform(train['vesselId'])
test['vesselId'] = le.transform(test['vesselId'])

# Sort datasets by 'vesselId' and 'time'
train = train.sort_values(by=['vesselId', 'time'])
test = test.sort_values(by=['vesselId', 'time'])

# Create 'previous_lat', 'previous_lon', and 'delta_time' in the training set
train['previous_lat'] = train.groupby('vesselId')['latitude'].shift(1)
train['previous_lon'] = train.groupby('vesselId')['longitude'].shift(1)
train['delta_time'] = train.groupby('vesselId')['time'].diff().dt.total_seconds()

# Drop rows with missing values resulting from the shift operation
train = train.dropna(subset=['previous_lat', 'previous_lon', 'delta_time'])

# Prepare training features and targets
X_train = train[['vesselId', 'previous_lat', 'previous_lon', 'delta_time']]
y_train_lat = train['latitude']
y_train_lon = train['longitude']

# Initialize 'previous_lat', 'previous_lon', and 'delta_time' in the test set
test['previous_lat'] = np.nan
test['previous_lon'] = np.nan
test['delta_time'] = np.nan

# Retrieve last known positions from the training set
last_positions = train.groupby('vesselId').apply(lambda x: x.iloc[-1])[['vesselId', 'latitude',
↪ 'longitude', 'time']]
last_positions = last_positions.set_index('vesselId')

# Train separate Random Forest models for latitude and longitude
model_lat = RandomForestRegressor(n_estimators=50, random_state=42)
model_lat.fit(X_train, y_train_lat)
```

```python

model_lon = RandomForestRegressor(n_estimators=50, random_state=42)
model_lon.fit(X_train, y_train_lon)

# Prepare a list to collect the prediction results
submission_rows = []

# Loop over each vessel in the test data
for vessel_id in test['vesselId'].unique():
    vessel_test_data = test[test['vesselId'] == vessel_id].copy()
    vessel_test_data = vessel_test_data.sort_values(by='time')

    # Check if the vessel_id exists in the last_positions
    if vessel_id in last_positions.index:
        prev_lat = last_positions.loc[vessel_id, 'latitude']
        prev_lon = last_positions.loc[vessel_id, 'longitude']
        last_time = last_positions.loc[vessel_id, 'time']
    else:
        # If vessel_id is not in the training data, skip prediction
        continue

    # Iterate over each record for the vessel
    for idx, row in vessel_test_data.iterrows():
        delta_time = (row['time'] - last_time).total_seconds()

        # Prepare the feature vector
        X_test_row = pd.DataFrame({
            'vesselId': [vessel_id],
            'previous_lat': [prev_lat],
            'previous_lon': [prev_lon],
            'delta_time': [delta_time]
        })

        # Predict latitude and longitude
        predicted_lat = model_lat.predict(X_test_row)[0]
        predicted_lon = model_lon.predict(X_test_row)[0]

        # Update previous values for the next iteration
        prev_lat = predicted_lat
        prev_lon = predicted_lon
        last_time = row['time']

        # Append the prediction to the submission list
        submission_rows.append({
            'ID': row['ID'],
            'longitude_predicted': predicted_lon,
            'latitude_predicted': predicted_lat
        })

# Create a submission DataFrame from the list
submission_df = pd.DataFrame(submission_rows)

# Merge the predictions with the test data based on 'ID'
final_submission = test[['ID']].merge(submission_df, on='ID', how='left')

# Save the submission file
final_submission.to_csv('submission.csv', index=False)
```

6.6 Result

This model proved to be our best-scoring model, with a Kaggle score of 115.8. Of course, this was nice, but we also think it is a bit strange that this very simple model would outperform our more elaborate models. However, as discussed in the following section, we recognize that this model may be overly simplistic, as it assumes that future vessel movements depend solely on the immediate previous position and time interval. We acknowledge that this simplification may overlook more complex navigational patterns.

6.7 Feature Importance For Random Forest Model

This section explores the importance of features in our Random Forest models used to predict latitude and longitude. Two separate Random Forest models were trained to predict each of these target variables individually, and the feature importance scores were computed for each model to identify the contribution of each input feature.

The tables below summarize the importance of the features for the models predicting latitude and longitude. The model prioritizes past latitude and longitude values (`previous_lat` and `previous_lon`) along with `delta_time`, which captures the time difference between observations. The `vesselId` feature, which identifies the unique vessel, has a lower importance score, indicating that the historical positional data and time intervals are the primary drivers of the model's predictive capabilities.

Feature Importance for Latitude Prediction Model		
Rank	**Feature**	**Importance**
1	previous_lat	0.995888
2	delta_time	0.003003
3	previous_lon	0.000875
4	vesselId	0.000234

Feature Importance for Longitude Prediction Model		
Rank	**Feature**	**Importance**
1	previous_lon	0.995079
2	delta_time	0.004414
3	previous_lat	0.000389
4	vesselId	0.000117

Table 1: Feature Importance for Latitude and Longitude Prediction Models

The results indicate that `previous_lat` and `previous_lon` are the most influential features for predicting latitude and longitude. This suggests that a vessel's previous positions strongly indicate its current location. Though less influential, the `delta_time` feature still contributes to the model's understanding by providing the time elapsed since the last recorded position, allowing the model to account for temporal changes in location.

In contrast, `vesselId`, which denotes the specific vessel, has the lowest importance score in both models. This outcome implies that while the vessel's unique identifier may provide some context, it does not significantly impact predictions. This finding aligns with the notion that a vessel's historical path, rather than its specific vessel ID, holds the most predictive power in forecasting its location.

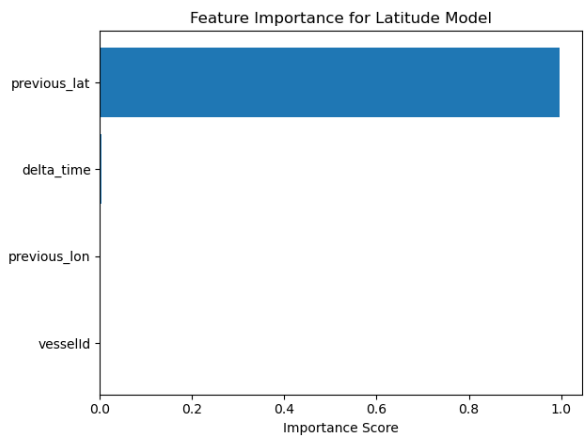

Figure 7: Feature importance for Latitude Prediction Model

Figures 7 and 8 visually illustrate the feature importance for each model. These bar charts confirm that 'previous_lat' and 'previous_lon' overwhelmingly dominate the feature contributions in their respective models, underscoring the importance of positional data in location prediction tasks. Future model improvements could focus on refining temporal and spatial feature representations and exploring additional factors that may influence the vessel's movement patterns.

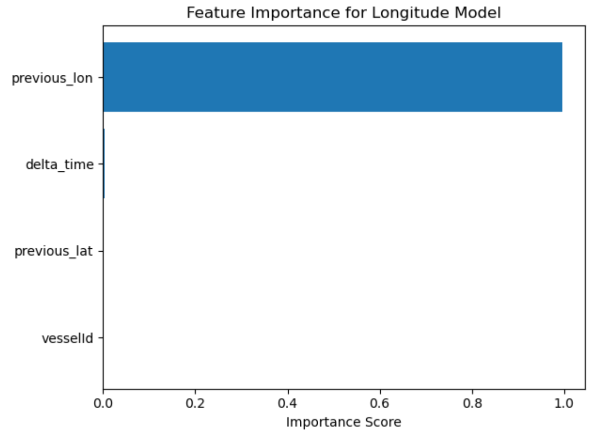

Figure 8: Feature importance for Longitude Prediction Model

7 Conclusions

In this project, we explored various models and feature engineering techniques to improve vessel position predictions. We achieved our best score of 115.8 on Kaggle with a Random Forest model. Our analysis highlighted the importance of recent positional data and time intervals, though more complex models incorporating additional features did not perform as well.

References

[1] "AIS Navigation Status - AIS Data - VT Explorer." [Online]. Available: https://api.vtexplorer.com/docs/ref-navstat.html

[2] Cristi, "How fast can large cargo ships go? What is the maximum speed that they can safely travel at?" Jun. 2024. [Online]. Available: https://medium.com/@crisri/how-fast-can-large-cargo-ships-go-what-is-the-maximum-speed-that-they-can-safely-travel-at-ae6b1f2c0155

[3] X. Liu, W. He, J. Xie, and X. Chu, "Predicting the Trajectories of Vessels Using Machine Learning," in *2020 5th International Conference on Control, Robotics and Cybernetics (CRC)*, Oct. 2020, pp. 66–70. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9253496/figures#figures