National College of Ireland

Configuration Manual

MSCDA_JANOL24_O

John Slattery
Student ID: x23265981

School of Computing
National College of Ireland

Supervisor: Mohammed Hasanuzzaman

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: John Slattery

Student ID: x23265981

Programme: MSCDA_JANOL24_O **Year:** 2025

Module: Research Project Report

Supervisor: Mohammed Hasanuzzaman

Submission Due Date: 11/12/2025

Project Title: What novel AI-driven approach can Dublin's traffic systems adopt to integrate weather data and improve daily congestion prediction beyond existing studies?

Word Count: 1365 **Page Count** 9

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: .....

Date: .....

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

John Slattery
Student ID: x23265981

1 Technology Setup

The project was implemented using Python 3.11.5 within a Jupyter Notebook environment, selected to ensure transparency, reproducibility, and facilitate experimentation. The modelling pipeline employs computationally efficient tree-based algorithms, specifically Random Forest and Gradient Boosting, which enable all experiments to be conducted successfully on a standard CPU environment.

1.1 Local Development Environment

- Operating System: Windows 11
- Python Version: 3.11.5
- Notebook: Jupyter Notebook version 7.3.2

1.2 Required Python Libraries

- Pandas
- Numpy
- Scikit-Learn
- Matplotlib
- Seaborn
- SHAP
- Joblib

1.3 Folder Structure

- Desktop → Traffic & Weather Analysis Code
- Traffic & Weather Analysis Code → .csv files
    - hly532.csv
    - Traffic_Flow_Data_Jan_to_June_2022_SDCC
    - Traffic_Flow_Data_June_to_December_2022_SDCC
    - Traffic_Flow_Data_Jan_to_June_2023_SDCC
    - Traffic_Flow_Data_April_to_September_2024_SDCC
- Traffic & Weather Analysis Code → .pkl files
    - gbr_Model.pkl
    - rf_model.pkl
- Traffic & Weather Analysis Code → plots
- Traffic & Weather Analysis Code → DA Code Traffic Weather.ipynb

2 The Code

2.1 Imports & Settings

The core Python libraries necessary for the analysis are imported in this section. Pandas and numpy are used for data manipulation, while joblib is employed for model serialisation. The RandomForestRegressor and GradientBoostingRegressor classes from sklearn.ensemble, along with the mean_absolute_error, mean_squared_error, and r2_score metrics from sklearn.metrics, are included for model development and evaluation. Additionally, matplotlib.pyplot and seaborn are imported to facilitate visualisation of exploratory analysis and model diagnostics. Collectively, these imports establish the computational environment required for the complete traffic–weather modelling pipeline.

Imports & Settings

```
import pandas as pd
import numpy as np
import joblib
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

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

Figure 1: Import & Settings Code

2.2 Load & Prepare Weather Data

The weather processing module involves the reading of the Met Éireann hourly climate data (hly532.csv) while ignoring the metadata entries and restricting the import to the targeted variables named corresponding to the columns of date, rainfall, temperature, wet bulb temperature, vapour pressure, relative humidity, and mean sea-level pressure. The date column is changed to datetime format, while the vapour pressure and humidity columns are forced to be numeric to handle any potential issues with data type. The next step involves the utilization of a map function to calculate the daily averages based on the hourly data by resampling based on the date index to produce the cleaned weather daily data that contains only one record per day.

Load & Prepare Weather Data

```
weather_raw = pd.read_csv(
    "hly532.csv",
    skiprows=23,
    low_memory=False,
    usecols=["date", "rain", "temp", "wetb", "vappr", "rhum", "msl"]
)

weather_raw["date"] = pd.to_datetime(weather_raw["date"], errors="coerce")

for col in ["vappr", "rhum"]:
    weather_raw[col] = pd.to_numeric(weather_raw[col], errors="coerce")

print("Weather (hourly) sample:")
print(weather_raw.head())
print(weather_raw.dtypes)

weather_agg_map = {
    "rain":  "sum",
    "temp":  "mean",
    "wetb":  "mean",
    "vappr": "mean",
    "rhum":  "mean",
    "msl":   "mean",
}

weather_daily = (
    weather_raw
    .set_index("date")
    .resample("D")
    .agg(weather_agg_map)
    .reset_index()
)

print("\nWeather (daily) sample:")
print(weather_daily.head())
```

Figure 2: Weather Preparation Code

2.3 Load & Prepare Traffic Data

The data was collected for the purpose of the South Dublin County Council from four different CSV files, each of which represented a different timeframe. Each CSV file was read into a DataFrame with low_memory parsing set to false to maintain the integrity of the data type, and a new column named source_file was added to each DataFrame to track the source of the information. The process involved checking for the presence of target column names to handle slight variations in the date and flow columns. The target columns of interest were then transformed into datetime formats. The DataFrames were then combined to create a master table, and the final dataframe, traffic_daily, was derived through the elimination of duplicate entries or application of relevant aggregations.

Load & Prepare Traffic Data

```
traffic_files = [
    "Traffic_Flow_Data_Jan_to_June_2022_SDCC.csv",
    "Traffic_Flow_Data_June_to_December_2022_SDCC.csv",
    "Traffic_Flow_Data_Jan_to_June_2023_SDCC.csv",
    "Traffic_Flow_Data_April_to_September_2024_SDCC.csv",
]

traffic_dfs = []

for f in traffic_files:
    print(f"Loading traffic file: {f}")
    df = pd.read_csv(f, low_memory=False)

    df["source_file"] = f

    print("  Columns:", list(df.columns)[:12], "...")

    if "date" not in df.columns:
        raise ValueError(f"'date' column not found in {f}")
    if "flow" not in df.columns:
        raise ValueError(f"'flow' column not found in {f}")

    df["date"] = pd.to_datetime(df["date"], errors="coerce")
    df["flow"] = pd.to_numeric(df["flow"], errors="coerce")

    traffic_dfs.append(df)

traffic_raw = pd.concat(traffic_dfs, ignore_index=True)

print("\nCombined traffic raw sample:")
print(traffic_raw.head())
print("Combined traffic rows (before dedupe):", len(traffic_raw))

candidate_subset = ["date", "site", "start_time", "end_time"]
subset_cols = [c for c in candidate_subset if c in traffic_raw.columns]

if subset_cols:
    print("Using duplicate subset columns:", subset_cols)
    traffic_raw = traffic_raw.drop_duplicates(subset=subset_cols, keep="first")
else:
    print("Warning: detailed columns not found; deduping on 'date' only.")
    traffic_raw = traffic_raw.drop_duplicates(subset=["date"], keep="first")

print("Combined traffic rows (after dedupe):", len(traffic_raw))

traffic_daily = (
    traffic_raw
    .groupby("date", as_index=False)["flow"]
    .sum()
    .rename(columns={"flow": "traffic"})
    .sort_values("date")
)
```

Figure 3: Traffic Preparation Code

2.4 Merge Datasets

The integration step normalizes the daily weather and traffic DataFrames to the shared temporal domain. This involves the following: The weather_daily DataFrame is transformed into a datetime format and arranged in chronological order. This involves finding the maximum value of the two start values and the minimum value of the two end values to establish the shared temporal domain, followed by clipping both the traffic_daily DataFrame and the weather_daily DataFrame to the shared domain. The clipped DataFrames are combined through an inner join based on the date column and then sorted to form a combined DataFrame (df) that has the traffic and weather attributes for each day.

```
weather_daily["date"] = pd.to_datetime(weather_daily["date"], errors="coerce")
weather_daily = weather_daily.sort_values("date")

print("Weather date range:", weather_daily["date"].min(), "→", weather_daily["date"].max())

start_date = max(traffic_daily["date"].min(), weather_daily["date"].min())
end_date   = min(traffic_daily["date"].max(), weather_daily["date"].max())

print("Overlapping date range:", start_date, "→", end_date)

traffic_clip = traffic_daily[
    (traffic_daily["date"] >= start_date) & (traffic_daily["date"] <= end_date)
].copy()

weather_clip = weather_daily[
    (weather_daily["date"] >= start_date) & (weather_daily["date"] <= end_date)
].copy()

df = (
    traffic_clip
    .merge(weather_clip, on="date", how="inner")
    .sort_values("date")
)

print("Merged rows:", len(df))
print(df.head())
print(df.tail())
```

Figure 4: Merge Datasets Code

2.5 Feature Engineering Sample

The feature engineering step is applied to the combined dataset through the duplication of the dataframe into df_fe, followed by the derivation of a series of temporal and historical traffic variables. The date column is then employed to derive the day of the week and the month, and a binary value is assigned to is_weekend to identify Saturday and Sunday. The following steps include the generation of lagged variables for traffic in the last day (traffic_lag1) and for the same day in the last week (traffic_lag7), and the calculation of the rolling averages for the last 3 days and 7 days (traffic_roll3 and traffic_roll7), respectively. This is followed by the elimination of rows with missing values created by the lag and rolling operations to create the cleaned model dataframe, stored in df_model with a reset index. 70/30 chronological split was added also.

```
df_fe = df.copy()

df_fe["dayofweek"] = df_fe["date"].dt.dayofweek
df_fe["month"] = df_fe["date"].dt.month
df_fe["is_weekend"] = df_fe["dayofweek"].isin([5, 6]).astype(int)

df_fe["traffic_lag1"] = df_fe["traffic"].shift(1)
df_fe["traffic_lag7"] = df_fe["traffic"].shift(7)

df_fe["traffic_roll3"] = df_fe["traffic"].rolling(3).mean()
df_fe["traffic_roll7"] = df_fe["traffic"].rolling(7).mean()

df_model = df_fe.dropna().reset_index(drop=True)

FEATURES = [
    "rain", "temp", "wetb", "vappr", "rhum", "msl",
    "dayofweek", "month", "is_weekend",
    "traffic_lag1", "traffic_lag7", "traffic_roll3", "traffic_roll7"
]

X = df_model[FEATURES]
y = df_model["traffic"]

split_idx = int(len(df_model) * 0.7)
X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

print("Train shape:", X_train.shape, "Test shape:", X_test.shape)

print("df_model created with rows:", len(df_model))
print(df_model.head())
print("Final modelling rows:", len(df))
```

Figure 5: Feature Engineering Code

2.6 Train Models

The modelling section trains the baseline and ensemble tree-based models on the engineered feature set. A simple mean-value baseline is created by filling an array for the test period with the mean of the training target values. A RandomForestRegressor is then instantiated with 500 trees, a maximum depth of 12, a minimum of five samples per leaf, a fixed random seed, and parallel execution enabled, and is fitted to the training data before generating predictions on the test set. In parallel, a GradientBoostingRegressor with 400 estimators, a learning rate of 0.05, and a maximum depth of 4 is trained and used to obtain its own predictions. A hybrid ensemble forecast is computed as the simple average of the Random Forest and Gradient Boosting predictions.

```
y_mean = np.full_like(y_test, fill_value=np.mean(y_train), dtype=float)

rf = RandomForestRegressor(
    n_estimators=500,
    max_depth=12,
    min_samples_leaf=5,
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)

gbr = GradientBoostingRegressor(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=4,
    random_state=42
)
gbr.fit(X_train, y_train)
y_pred_gbr = gbr.predict(X_test)

y_pred_hybrid = 0.5 * y_pred_rf + 0.5 * y_pred_gbr
```

Figure 6: Train Models Code

2.7 Metrics Output

Model performance is evaluated using a helper function print_metrics, which computes the mean absolute error, root mean squared error, and coefficient of determination between true and predicted values. The function prints a formatted line summarising these metrics for each model. It is called sequentially for the baseline mean predictor, the Random Forest model, the Gradient Boosting model, and the hybrid RF+GBR ensemble. This produces a concise comparative summary of forecasting accuracy across all approaches evaluated in the study.

Metrics Output

```
def print_metrics(name, y_true, y_hat):
    mae  = mean_absolute_error(y_true, y_hat)
    rmse = mean_squared_error(y_true, y_hat, squared=False)
    r2   = r2_score(y_true, y_hat)
    print(f"{name:25s} | MAE: {mae:,.2f} vehicles/day | "
          f"RMSE: {rmse:,.2f} | R²: {r2:.3f}")

print("\n============= MODEL COMPARISON =============")
print_metrics("Baseline (mean)", y_test, y_mean)
print_metrics("Random Forest",   y_test, y_pred_rf)
print_metrics("Gradient Boosting", y_test, y_pred_gbr)
print_metrics("Hybrid RF+GBR",   y_test, y_pred_hybrid)
print("============================================")
```

Figure 7: Metric Outputs Code

2.8 Plotting Sample

The plotting code provides a visual comparison between observed and predicted traffic volumes over the test period. A new figure with a fixed size is created, and the actual daily traffic values are plotted against their index as a line labelled “Actual”. The hybrid ensemble predictions are plotted on the same axes as a second line labelled “Predicted”. A legend is added to distinguish between both series. This simple diagnostic plot allows visual inspection of how closely the hybrid model tracks the true traffic-flow dynamics over time. This approach was used for the majority of plots.

Predicted vs Actual

```
plt.figure(figsize=(10,6))
plt.plot(y_test.index, y_test, label="Actual")
plt.plot(y_test.index, y_pred_hybrid, label="Predicted")
plt.legend()
```

Figure 8: Example Plot Code

2.9 SHAP

For better explainability, the notebook uses the SHAP tool to demonstrate the predictions made by the Random Forest model. The SHAP library is then imported, and a TreeExplainer object is initialized based on the trained Random Forest model estimator. A random sample of no more than 200 observations is taken from the test data to ensure a computationally feasible process in the explanation step. The results derived from the SHAP calculation are presented in a SHAP summary plot formed with the sampled feature matrix and the pre-specified feature names, clearly illustrating the contribution and effect of each feature in the prediction process, confirming the importance of the feature variables and lagged traffic in the prediction model.

SHAP Explainability

```
import shap
```

```
explainer = shap.TreeExplainer(rf)

X_test_sample = X_test.sample(n=min(200, len(X_test)), random_state=42)
```

```
shap_values = explainer.shap_values(X_test_sample)

shap.summary_plot(
    shap_values,
    X_test_sample,
    feature_names=feature_cols,
    show=True
)
```

Figure 9: SHAP interpretability & Plot Codes

2.10 Joblib Train Models

The final code block prepares the models for reuse by retraining and serialising them with joblib. New instances of the Random Forest and Gradient Boosting regressors are fitted on the full training dataset and saved to disk as rf_model.pkl and gbr_model.pkl. The models are then reloaded to verify the persistence process, and a convenience function hybrid_predict() is defined. This function takes a feature matrix as input, generates predictions using the reloaded Random Forest and Gradient Boosting models, and returns their average as a hybrid forecast. This step enables the forecasting framework to be integrated into future applications without rerunning the complete training pipeline.

Joblib Train Models

```
[52]: rf_model = RandomForestRegressor(
          n_estimators=500,
          max_depth=12,
          random_state=42
      )
      gbr_model = GradientBoostingRegressor(
          n_estimators=400,
          learning_rate=0.05,
          max_depth=4,
          random_state=42
      )

      rf_model.fit(X_train, y_train)
      gbr_model.fit(X_train, y_train)


      joblib.dump(rf_model, "rf_model.pkl")
      joblib.dump(gbr_model, "gbr_model.pkl")

      print("Models saved to disk.")


      rf_model = joblib.load("rf_model.pkl")
      gbr_model = joblib.load("gbr_model.pkl")

      print("Models loaded successfully.")


      def hybrid_predict(X):
          """
          Produces a hybrid forecast by averaging
          the Random Forest and Gradient Boosting predictions.
          """
          rf_pred = rf_model.predict(X)
          gbr_pred = gbr_model.predict(X)
          return (rf_pred + gbr_pred) / 2
```

Figure 10: Joblib Code

3 Plot Output Samples

Below are some sample plots to give a flavour of the overall thesis:

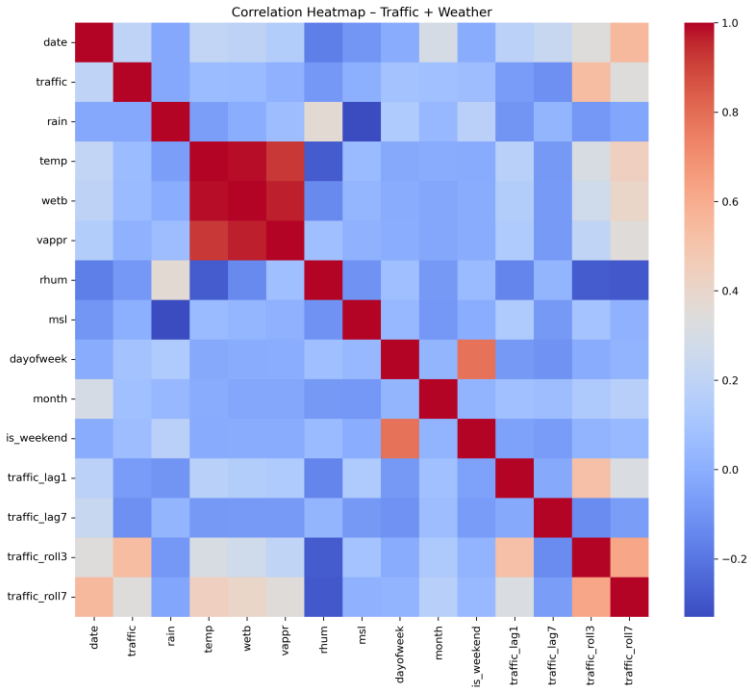

Figure 11: Correlation heatmap example

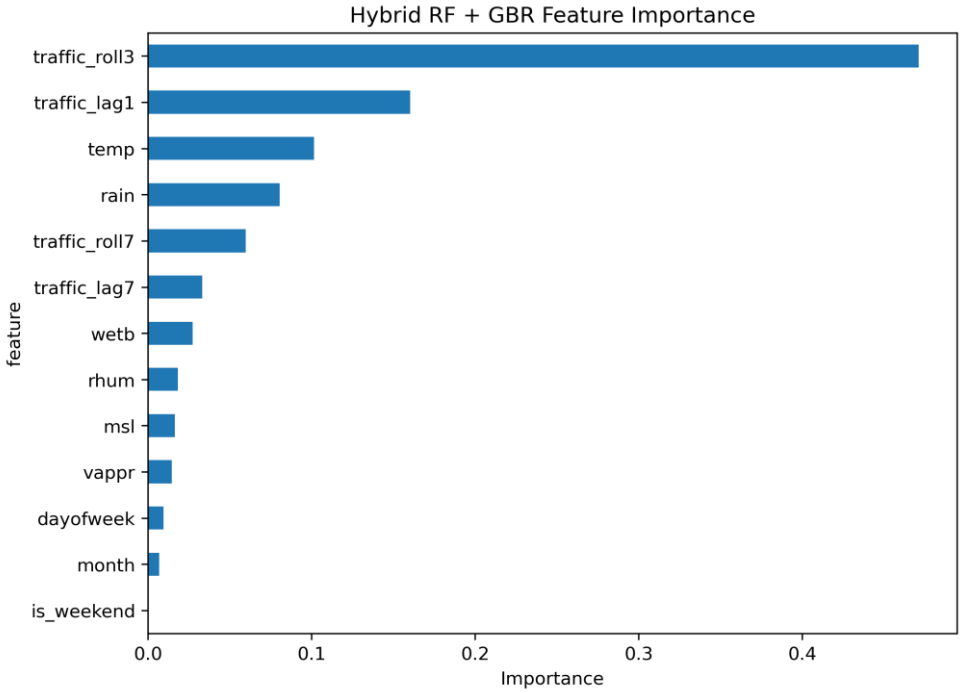

Figure 12: Feature Importance example

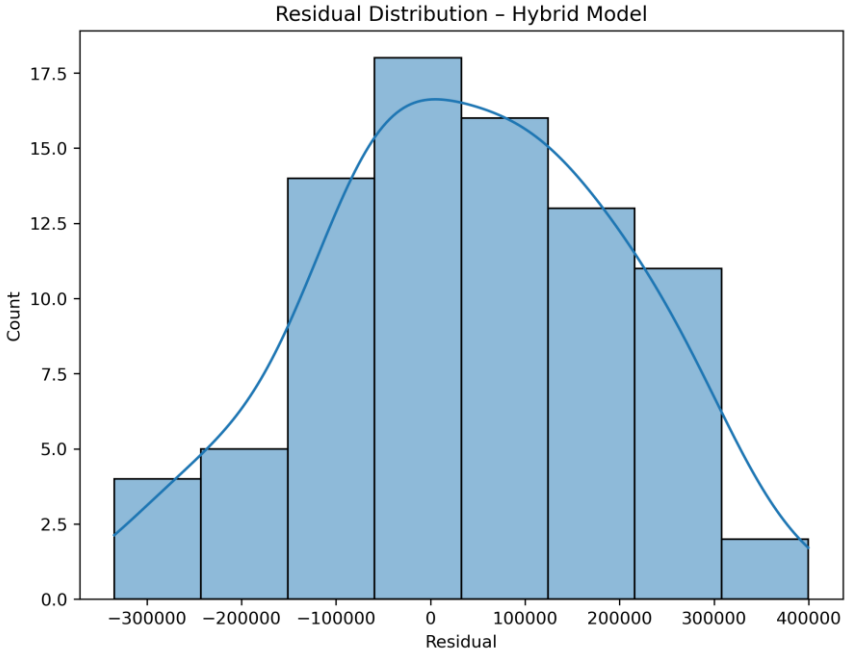

Figure 13: Residual plot sample

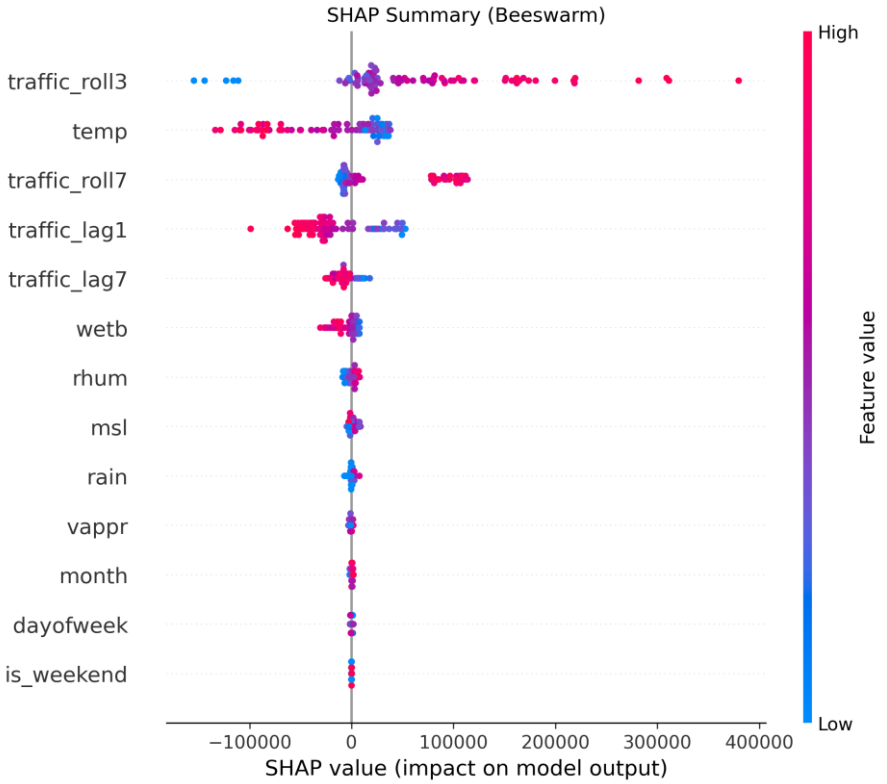

Figure 14: SHAP plot example

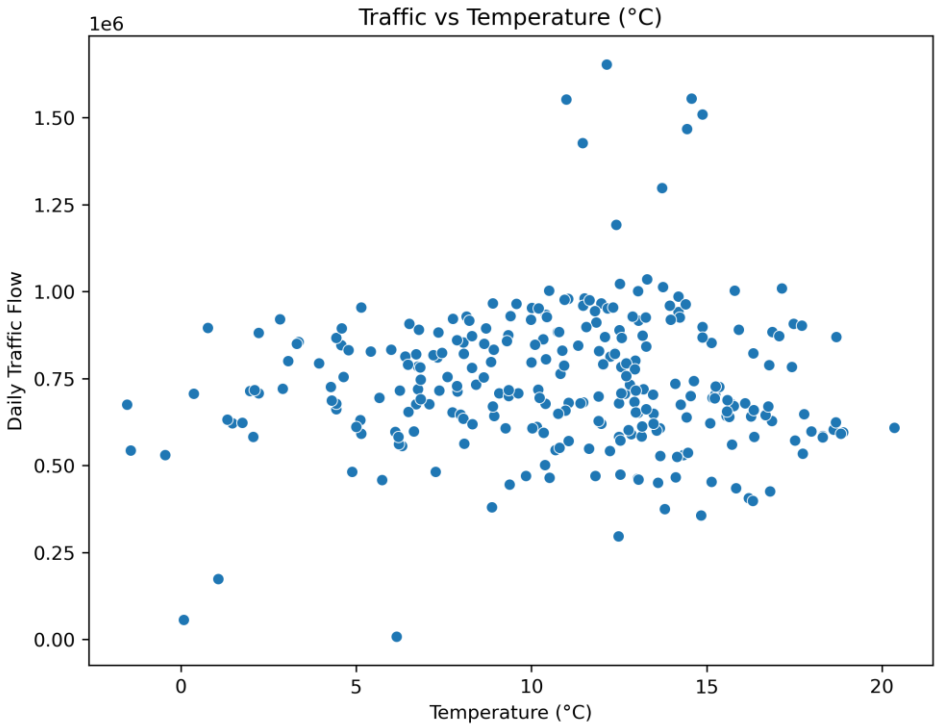

Figure 15: Scatterplot example