National College of Ireland

Configuration Manual

MSc Research Project
MSCCYBE_JANO24_O

Niamh Daly
Student ID: x19370553

School of Computing
National College of Ireland

Supervisor: Ross Spelman

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……Niamh Daly……

Student ID: ……x19370553……

Programme: ……MSCCYBE_JANO24_O …… **Year:** ……2……..

Module: ……MSc (Research) Practicum/Internship Part 2……

Supervisor: ……Ross Spelman……

Submission Due Date: ……11th December 2025 2pm……

Project Title: …… Synthetic Attack Evaluation of Anomaly Based Intrusion Detection System for Formula 1 Telemetry……

Word Count: ……5050…… **Page Count:** ……31……

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: ……Niamh Daly……

Date: ……8th December 2025……

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

Niamh Daly
Student ID: x19370553

1 Introduction

This document is to be used as the configuration manual. This document consists of the step by step instructions including screenshots along with versions of the tools and technologies that were utilised in order to make the artefact reproducible. The code artefact is executed by running the code. The steps below explain a step by step of what the code is doing, and what results are being produced.

2 System and Software Configuration and Requirements

All experiments were conducted on a workstation running Windows 11 (64-bit) with Python 3.9.13, using the following software ecosystem outlined in Table 1. The environment was held constant for all runs to avoid variation in results and ensure reproducibility.

Table 1: Summary of Tools and Technologies

Tool/ Library	Version	Purpose
Python	3.9.13	Programming language used to run data analysis, ML, and telemetry scripts
FastF1	3.6.1	Python Library for acquisition of real Formula 1 telemetry data
NumPy	2.0.2	Time series data handling and preprocessing
Pandas	2.3.3	Data handling and transformation[1]
Scikitlearn	1.6.1	ML library for modelling (PCA, IF) and data preprocessing (scaling)[2]
Matplotlib	3.9.2	Visualisation of telemetry and anomaly markers (line plots, etc.)
Seaborn	0.13.2	Built on Matplotlib, used for plotting (heatmaps, etc.)[3]
Jupyter Notebook	7.4.5	Interactive coding environment for Python used for experiment execution, reproducibility, and iterative development

[1] https://pandas.pydata.org/
[2] https://www.ibm.com/think/topics/scikit-learn
[3] https://seaborn.pydata.org/

3 Importing Relevant Libraries

A number of relevant libraries were used to produce the code artefact, these libraries are outlined in the table below, with the relevant code line, and a description of what the import is doing.

Table 2: Relevant Library Imports

Types of Imports	Code	Description
Standard library imports	import os	Provides functions for interacting with the operating system
	import warnings warnings.filterwarnings("ignore")	Allows filtering or suppressing warning messages and suppresses warnings for cleaner output
Data manipulation and analysis	import pandas as pd	Primary library for data manipulation and analysis
	import numpy as np	Numerical computing library, useful for arrays and calculations
Visualization libraries	import matplotlib.pyplot as plt	Core plotting library
	import seaborn as sns	Statistical data visualization built on top of matplotlib
Machine Learning utilities	from sklearn.model_selection import train_test_split	Splits data into training and test sets[4]
	from sklearn.preprocessing import StandardScaler	Standardizes features
	from sklearn.decomposition import PCA	Principal Component Analysis for dimensionality reduction
	from sklearn.ensemble import IsolationForest	Anomaly detection algorithm
FastF1 F1 telemetry & timing data	import fastf1	Library for accessing and analyzing Formula 1 telemetry and timing data[5]
	from fastf1.core import Laps	FastF1 class for working with lap timing datasets

4 Cache Setup

Creating a folder named 'f1_cache' to store cached Formula 1 session data if the folder doesn't already exist. FastF1 caching is enabled so previously downloaded session data is stored locally on the system. This method reduces the need to repeat API calls, and additionally speeds up any future data loading and analysis[6].

[4] https://github.com/scikit-learn/scikit-learn

[5] https://github.com/theOehrly/Fast-F1

[6] https://medium.com/formula-one-forever/formula-1-data-analysis-with-fastf1-%EF%B8%8F-d451b30f3a91

```
# Creating a directory for caching F1 session data
cache_dir = "f1_cache"
os.makedirs(cache_dir, exist_ok=True)

# Enabling FastF1 caching to speed up repeated data loading
fastf1.Cache.enable_cache(cache_dir)
```

Figure 1: Setting Up Cache

5 Dataset Compilation

The following subsections outline the steps taken to compile the chosen dataset from FastF1.

5.1 Drivers and Race Selection

Drivers from the 2024 season were selected based on their driver abbreviation, or three letter code for their surnames, e.g. 'VER' for 'Verstappen'. Race session was also selected, based on the year, grand prix, and session (i.e. Race, Practice, Qualifying, etc).

```
# List of F1 driver abbreviations to analyze
drivers = [
    'VER','PER','SAI','LEC','RUS','NOR','HAM','PIA','ALO','STR',
    'ALB','HUL','OCO','GAS','BOT','ZHO','MAG','LAW','TSU','SAR'
]

# List of race sessions to load
# Format: year, Grand Prix name, and session type (e.g., R = Race)
races = [
    {'year': 2024, 'gp': 'Bahrain', 'session': 'R'}
]
```

Figure 2: Drivers and Race Selection

5.2 Output File Configuration

The data is set to be saved to csv file. The processed telemetry data will be saved to this file so that it can be reused without reprocessing in future runs.

```
# File name where combined telemetry and lap data will be saved
output_file = "f1_bahrain_race_telemetry.csv"
```

Figure 3: Output File Configuration

5.3 Collecting the Telemetry

If the telemetry csv file has already been created, the file is loaded, instead of being reprocessed.

```python
# If we already created the telemetry CSV, load it instead of reprocessing
if os.path.exists(output_file):
    print(f"File '{output_file}' exists, loading...")
    df = pd.read_csv(output_file)
```

Figure 4: 'If' Statement checking for CSV file

Else the telemetry collection process is run. The script loops through each race and driver and loads the relevant session from FastF1 that were defined above, and extracts the valid laps that have occurred without pit stops. For every lap that is being collected, the chosen telemetry channels are being retrieved, and the driver and race info is being recorded for each lap into a list format. If no telemetry is able to be collected, an error is raised to prevent the script running, with empty results.

```python
else:
    print(f"File '{output_file}' not found, collecting telemetry...")
    # List to temporarily store telemetry data from each lap/session
    frames = []
    for race in races:
        year = race['year']; gp = race['gp']; sess = race['session']
        print(f"Loading {year} {gp} {sess}")
        session = fastf1.get_session(year, gp, sess)
        session.load()
        # Loop through each driver code in this race session
        for drv in drivers:
            laps: Laps = session.laps.pick_driver(drv).pick_wo_box()
            if laps.empty:
                continue
            # Loop through each lap and extract telemetry
            for _, lap in laps.iterlaps():
                tel = lap.get_car_data().add_distance()
                # Select relevant telemetry channels (if they exist in dataset)
                cols = [c for c in ['Time','Lap','Distance','Speed','Throttle','Brake',
                        'nGear','RPM','DRS'] if c in tel.columns]
                tel = tel[cols].copy()
                # Add metadata
                tel['Driver']  = drv
                tel['Year']    = year
                tel['Race']      = gp
                tel['Session'] = sess
                tel['Lap'] = int(lap['LapNumber'])
                # Append processed telemetry for this lap to list
                frames.append(tel)
    # If no data was collected, stop execution early
    if not frames:
        raise RuntimeError("No telemetry collected; check drivers/races/session types.")
```

Figure 5: 'Else' Statement collecting the telemetry data

The remainder of the ‘Else’ Statement combines all the telemetry data into a single data frame, and then saves the data frame to the CSV file.

```python
    # Combine all lap telemetry into a single DataFrame
    df = pd.concat(frames, ignore_index=True)

    # Save to CSV
    df.to_csv(output_file, index=False)
    print(f"Saved {output_file} with shape {df.shape}")
```

Figure 6: Remainder of 'Else' Statement

5.4 Data Inspection: Previewing the Collected Telemetry

The data frame is previewed by displaying the shape, columns and names within the data frame. Also shows the structural details and summary statistics which assisted in understanding the datasets contents[7] [8] [9].

```
display(df.head())

print(df.shape, df.columns.tolist())

df.info()

df.describe()
```

Figure 7: Data Frame Inspection

5.5 Data Preprocessing

Converted 'Time' column to time delta, for allowing proper time arithmetic for telemetry analysis[10] [11]. Created a new column for 'Time' in seconds, and dropped the original column 'Time'.

```
# Convert 'Time' column into a pandas timedelta
df['Time'] = pd.to_timedelta(df['Time'])
# Create a new column in seconds
df['t_s'] = df['Time'].dt.total_seconds().astype(float)

df.drop(columns=['Time'], inplace=True, errors='ignore')
```

Figure 8: 'Time' column conversions

Additionally 'Brake' and 'DRS' columns were converted into integers, as machine learning models and anomaly detection expect numeric types. Wanted to ensure 'Throttle' values were numeric, so ran 'to_numeric'.

```
# Converting features (Brake, DRS) into integers
for col in ['Brake', 'DRS']:
    if col in df.columns:
        df[col] = df[col].astype(int)

# Ensure throttle values are numeric
if 'Throttle' in df.columns:
    df['Throttle'] = pd.to_numeric(df['Throttle'], errors='coerce')
```

Figure 9: Converting values to numeric values

---

[7] https://realpython.com/pandas-python-explore-dataset/

[8] https://www.geeksforgeeks.org/python/python-pandas-dataframe-info/

[9] https://www.statology.org/pandas-describe/

[10] https://medium.com/@amit25173/how-to-convert-pandas-timedelta-to-seconds-b37f159fe2b4

[11] https://pandas.pydata.org/pandas-docs/version/2.1.2/reference/api/pandas.Series.dt.total_seconds.html#pandas.Series.dt.total_seconds

6 Exploratory Data Analysis - Visualisations

A number of visualisations were created to initially explore the data that had been selected for the analysis, these are outlined below in the following subsections.

6.1 Plotting Driver Lap Telemetry

Defined a function to visualise the telemetry for an individual driver during a chosen race session[12] [13]. The function plots speed, throttle, and brake as a function of the track distance[14] [15]. The parameters are as follows:

Table 3: Description of Function Variables

Parameter	Type	Description
df	DataFrame	The full telemetry dataset
driver	String	The driver code (e.g., 'VER', ‘'HAM')
race	String	Grand Prix name (e.g., 'Bahrain')
year	Integer	Year of season (e.g., 2024)
session_type	String	Session code: 'R' (Race), 'Q' (Quali), 'FP1' etc. (default 'R')
lap	String or Integer	Can get the ‘'fastest' or specific lap number

The dataset is filtered to the selected driver and session. A number of ‘If’ statements are used in the function:

- If no data is found, the function is exited.
- If ‘Distance’ doesn’t exist, the function is exited.
- If chosen lap is ‘fastest’ then will get the fastest lap,
    - Else will plot the specific lap that was chosen.

```
def plot_driver_telemetry(df, driver, race, year, session_type='R', lap='fastest'):

    data = df[
        (df['Driver'] == driver) &
        (df['Race'] == race) &
        (df['Year'] == year) &
        (df['Session'] == session_type)].copy()
    if data.empty:
        print(f"No data for {driver} {race} {year} {session_type}.")
        return
    if 'Distance' not in data.columns:
        print("No 'Distance' column in the data.")
        return
    if lap == 'fastest':
        lap_to_plot = data.groupby('Lap')['Speed'].max().idxmax()
    else:
        lap_to_plot = int(lap)
```

Figure 10: Beginning of Visualisation Function

[12] https://medium.com/formula-one-forever/formula-1-data-analysis-with-fastf1-%EF%B8%8F-d451b30f3a91

[13] https://python.plainenglish.io/telemetry-of-the-fastest-lap-of-a-f1-grand-prix-fastf1-tutorials-4211a48f5eac

[14] https://realpython.com/pandas-plot-python/

[15] https://pandas.pydata.org/docs/user_guide/visualization.html

The chosen lap data is then saved into the 'telemetry' variable. The selected driver, race, year, session type, and lap are then plotted. The speed, throttle, and brake, are plotted against the distance[16].

```
telemetry = data[data['Lap'] == lap_to_plot]

print(f"Plotting {driver} {race} {year} {session_type}, Lap {lap_to_plot}")

telemetry.plot(
    x='Distance',
    y=['Speed', 'Throttle', 'Brake'],
    figsize=(12, 6)
)

plt.title(f"{driver} {race} {year} {session_type} Lap {lap_to_plot}")
plt.xlabel("Distance [m]")
plt.show()
```

Figure 11: Remainder of the Function for Plotting the Graph

Selecting the specific driver, race, year, session type and lap. These can be altered if more data was loaded to the data frame for different races and years.

```
plot_driver_telemetry(
    df,
    driver='VER',
    race ='Bahrain',
    year=2024,
    session_type='R',
    lap='fastest'
)

plot_driver_telemetry(
    df,
    driver='HAM',
    race ='Bahrain',
    year=2024,
    session_type='R',
    lap=10
)
```

Figure 12: Selecting what will be Plotted

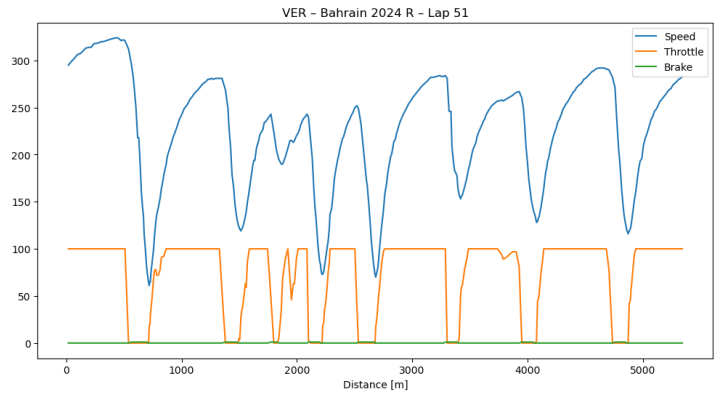

Figure 13: Visualisation that is produced

[16] https://medium.com/@samyak.bhansali.201559/data-analysis-using-f1-telemetry-462da3af6b17

6.2 Correlation Analysis of All Numeric Telemetry Features

The numeric features were selected for the correlation analysis, columns such as 'Driver', 'GP', 'Session', etc, were excluded as they are string and categorical values. The Pearson correlation matrix was computed between the telemetry features. The visualisation was created as a heatmap[17] [18] [19]. The heatmap shows all the correlation values formatted to two decimal places. The blue values are the negative correlation values and the red are the positively correlated values[20].

```
# Select only numeric features for correlation analysis
numeric_df = df.select_dtypes(include='number')
corr = numeric_df.corr()

plt.figure(figsize=(20, 16), dpi=130)
sns.heatmap(
    corr,
    annot=True,
    fmt='.2f',
    cmap='coolwarm',
    square=True
)

plt.title('Telemetry Feature Correlation Heatmap')
plt.show()
```

Figure 14: Correlation Analysis of All Numeric Telemetry Features

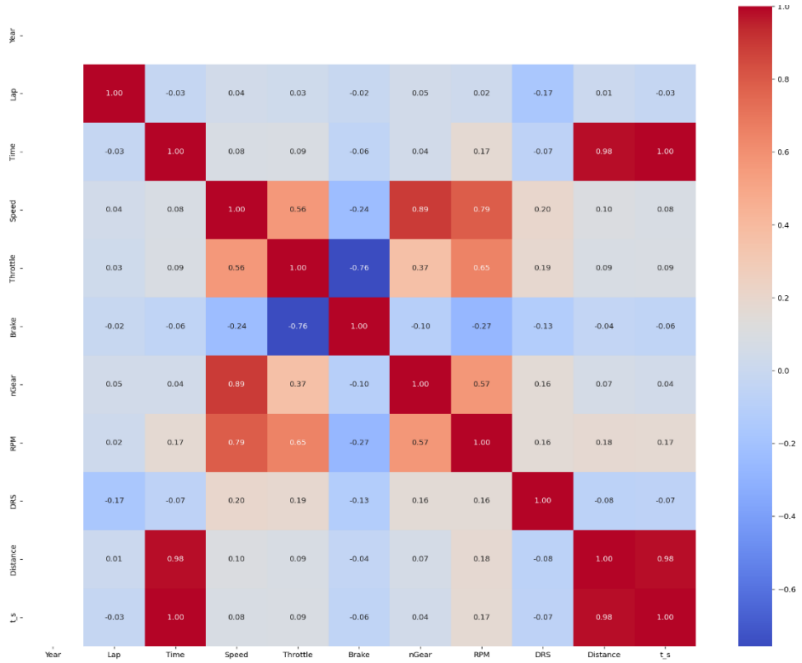

Figure 15: Resulting Correlation Heatmap

[17] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html#pandas.DataFrame.select_dtypes

[18] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html#pandas.DataFrame.corr

[19] https://realpython.com/numpy-scipy-pandas-correlation-python/

[20] https://seaborn.pydata.org/generated/seaborn.heatmap.html

6.3 Per Driver Telemetry Correlation Maps

To get individual driver correlation maps, created a for loop for the unique drivers that are present within the ‘Driver’ column in the data frame. The data frame is then filtered to only have the values of the specific driver. The correlation is computed on the ‘Speed’, ‘Throttle’, ‘nGear’, ‘RPM’, ‘DRS’ subset.

```
# Loop through each driver present in the dataset
for drv in df['Driver'].unique():
    subset = df[df['Driver'] == drv]
    corr = subset[['Speed', 'Throttle', 'nGear', 'RPM', 'DRS']].corr()

    plt.figure(figsize=(6, 5))
    sns.heatmap(
        corr,
        annot=True,
        cmap='coolwarm',
        vmin=-1, vmax=1
    )
    plt.title(f'Correlation Map - {drv}')
    plt.show()
```

Figure 16: Per Driver Telemetry Correlation Maps

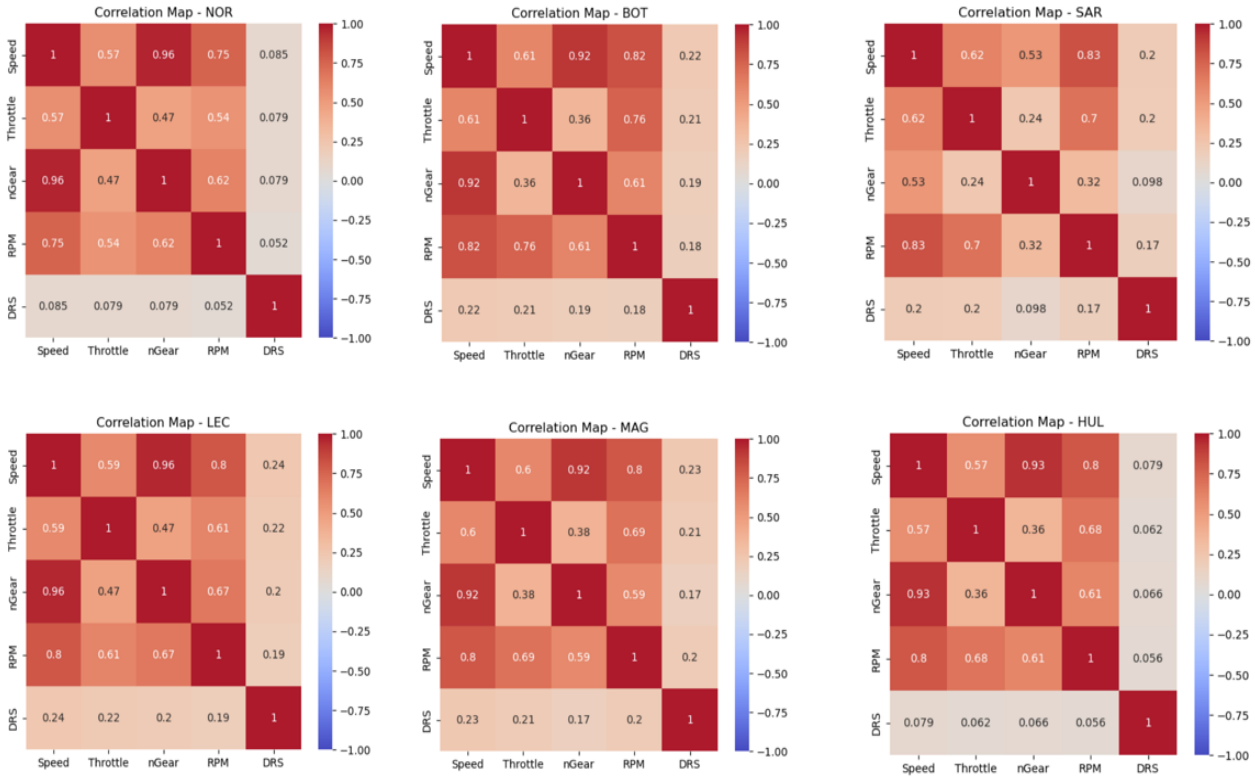

Figure 17: Correlation Heatmaps of 6 Drivers from Bahrain Race

7 Machine Learning Modelling

To run the machine learning models the data had to be appropriately prepared, these steps are outlined in the section below.

7.1 Feature Selection

The core telemetry features were selected that were to be used for the modelling. These features included 'Speed', 'Throttle', 'Brake', 'nGear', 'RPM', 'Distance', and 'DRS'. Each of these features were kept, if they were available within the data frame. A derivative feature was created for the rate of change of speed 'dSpeed', from the 'Speed' column[21] [22]. This derivative feature represents the acceleration or deceleration behaviour. This new column was added to the data frame.

```
feature_cols = ['Speed', 'Throttle', 'Brake', 'nGear', 'RPM', 'Distance', 'DRS']

feature_cols = [c for c in feature_cols if c in df.columns]

if 'Speed' in df.columns:
    df['dSpeed'] = df['Speed'].diff().fillna(0)

feature_cols = [c for c in feature_cols + ['dSpeed'] if c in df.columns]
```

Figure 18: Feature Selection

7.2 Train Test Split by Driver

The data was split into an 80% train and 20% test split per driver[23]. This is to allow the model to learn baseline behaviour per driver. The evaluation is carried out on the unseen samples from the same driver.

```
train_list = []
test_list = []

test_ratio = 0.2
```

Figure 19: Defining the Train Test Split

A 'For' loop is created to loop through the unique drivers within the data frame. A subset of the telemetry data is created for the driver and the columns that were defined in the feature selection are defined into a feature matrix. The data is split into the train and test splits based on the specific driver. Random state 42 was defined to make the split reproducible. Train and test splits are combined into final train and test splits data frames[24]. The shape of the train and test data frames are printed for reference.

---

[21] https://pandas.pydata.org/docs/user_guide/indexing.html

[22] https://www.geeksforgeeks.org/pandas/indexing-and-selecting-data-with-pandas/

[23] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[24] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.assign.html

```
for driver in df['Driver'].unique():
    df_d = df[df['Driver'] == driver].dropna(subset=feature_cols)
    if df_d.empty:
        continue

    X = df_d[feature_cols]

    X_train_d, X_test_d = train_test_split(
        X,
        test_size=test_ratio,
        shuffle=True,
        random_state=42
    )

    X_train_d = X_train_d.assign(Driver=driver)
    X_test_d  = X_test_d.assign(Driver=driver)

    train_list.append(X_train_d)
    test_list.append(X_test_d)

train_df = pd.concat(train_list).reset_index(drop=True)
test_df  = pd.concat(test_list).reset_index(drop=True)

print("Train shape:", train_df.shape)
print("Test shape:", test_df.shape)
train_df.head()
```

Figure 20: Train Test Per Driver Loop

7.3 Per Driver PCA and Isolation Forest

The per driver anomaly detection pipeline is described below, starting with standard scaler, PCA, and Isolation Forest[25]. Per driver telemetry allows for capturing the drivers baseline behaviour and flags deviations on their own scale.

Table 4: PCA IF Function Parameters

Parameter	Type	Description
train_df	pd.DataFrame	The combined training rows for all drivers. Must include 'Driver' and `feature_cols`
test_df	pd.DataFrame	The combined test rows for all drivers. Must include 'Driver' and `feature_cols`
feature_cols	List[string]	Numeric feature columns to use (e.g., 'Speed', 'Throttle', 'Brake', 'nGear', 'RPM', 'Distance', 'DRS', 'dSpeed')
n_components	Integer default=2	The number of PCA components. 2 is assigned as default as is convenient for plotting
contamination	Float default=0.03	The expected outlier fraction for IF. Will be tuned per dataset
random_state	Integer default=42	Allows for reproducibility for PCA & IF

[25] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Only the drivers that are present in both the train and test splits are used in the PCA IF model. The 'for' loop works through the drivers within the drivers set, and extracts the drivers train test split partitions. If either the train or test splits are empty it skips that drivers data. Additional checks include ensuring the features exist and are numeric, and dropping any rows with missing values from the data.

```
def run_pca_if_per_driver(
    train_df,
    test_df,
    feature_cols,
    n_components=2,
    contamination=0.03,
    random_state=42
):
    out = []
    drivers = sorted(set(train_df['Driver']).intersection(set(test_df['Driver'])))

    for d in drivers:
        tr = train_df[train_df['Driver'] == d]
        te = test_df[test_df['Driver'] == d]
        if tr.empty or te.empty:
            continue

        avail_feats = [c for c in feature_cols if c in tr.columns and c in te.columns]
        if not avail_feats:
            continue

        Xtr = tr[avail_feats].astype(float).dropna()
        Xte = te[avail_feats].astype(float).dropna()

        if Xtr.empty or Xte.empty:
            continue
```

Figure 21: Beginning of Per Driver PCA IF Function

The data present in the train and test datasets are then standardised using 'StandardScaler'. PCA will then be fit on the train, and transformed on the test data. Isolation Forest is fir on the PCA space and scored on the test[26].

```
# Standardised
scaler = StandardScaler()
Xtr_s = scaler.fit_transform(Xtr)
Xte_s = scaler.transform(Xte)

# PCA
pca = PCA(n_components=n_components, random_state=random_state)
Xtr_p = pca.fit_transform(Xtr_s)
Xte_p = pca.transform(Xte_s)

# Isolation Forest
iso = IsolationForest(
    n_estimators=200,
    contamination=contamination,
    random_state=random_state
)
iso.fit(Xtr_p)
scores = iso.score_samples(Xte_p)
```

Figure 22: Standardising, PCA, and IF

The predictions are attached back to the test set rows that align with the PCA transformed data and adds the Isolation Forest anomaly scores. This results in a clean aligned dataset that contains test rows that contain PCA coordinates and model scores.

[26] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

```
# Attach predictions back to the aligned subset of test rows
tmp = te.loc[Xte.index].copy()
tmp['pca_1'] = Xte_p[:, 0]
if n_components >= 2:
    tmp['pca_2'] = Xte_p[:, 1]
else:
    tmp['pca_2'] = np.nan
tmp['if_score'] = scores

out.append(tmp)
```

Figure 23: Adding PCA Features and Anomaly Scores for Plotting

The function returns the subset of the test data frame across the drivers that are present in both train and test splits, along with the 'pca_1', 'pca_2', and 'if_score'.

```
# Return combined augmented test rows
return pd.concat(out).reset_index(drop=True) if out else pd.DataFrame()
```

Figure 24: Return for the PCA IF Function

Table 5: PCA IF Function Returns

Returns	Variables	Description
test_df	pca_1	The PCA transformed coordinates of test rows
	pca_2	The PCA transformed coordinates of test rows
	if_score	Isolation Forest score, the higher the score the more normal the data, the lower the score, the more anomalous the data

8 Synthetic Anomaly Injection (Per Driver) Function

The synthetic anomalies are injected into a copy of the provided test data frame per driver. The function creates labelled faults for validating the PCA and IF pipeline. All the injections preserve the original index of the data frame. Each injected window is tagged with the following:

- ***is_anomaly***: will be 1 if window is anomalous
- ***anomaly_type***: a categorical code relating to the anomaly (e.g., 'A1_speed_spike')
- ***anomaly_params***: selected parameters (e.g., '+80kmh')
- ***anomaly_group***: an integer id to group contiguous samples from the same injection

The table below outlines the anomaly injection parameters for the synthetic anomaly injection function.

Table 6: Anomaly Injection Parameters

Parameter	Type	Description
test_df	pd.DataFrame	The telemetry subset to modify. The expected columns include 'Driver', 'Speed', 'Throttle', 'Brake', 'nGear', 'RPM', 'Distance'
min_dist	Float default=200	The minimum distance 'm' threshold to avoid injecting into formation lap or pit segments
types	Tuple[str], default = 'A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7'	Described which anomaly families to inject, any subset is allowed

A series of for loops are present within the function to ensure the data within the data frame is format in the correct types, and to ensure the anomaly label columns exist. Throttle scale is also detected in order to inject the appropriate values based on the scale. A group id is instantiated which increments per each injected window.

```
def inject_anomalies_per_driver(
    test_df,
    min_dist=200,
    types=('A1','A2','A3','A4','A5','A6','A7')
):

    rng = np.random.default_rng(42)
    df_out = test_df.copy()

    # Checking types for columns
    for col in ['Brake', 'DRS']:
        if col in df_out.columns:
            df_out[col] = df_out[col].astype(int)

    # Converting to numeric if missed in other steps
    for col in ['Speed','Throttle','nGear','RPM','Distance']:
        if col in df_out.columns:
            df_out[col] = pd.to_numeric(df_out[col], errors='coerce')

    # Checking anomly label columns exist
    for col in ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']:
        if col not in df_out.columns:
            df_out[col] = 0 if col == 'is_anomaly' else ''

    # Increments per each injected window
    group_id = 0

    # Detecting 'Throttle' Scale, (1-0, vs 1-100)
    thr_max = df_out['Throttle'].max() if 'Throttle' in df_out else np.nan
    throttle_vals = [85, 95, 100] if (pd.notna(thr_max) and thr_max > 1.5) else [0.85, 0.95, 1.00]
```

Figure 25: Beginning of Anomaly Injection Function

Next the drivers are iterated over to inject the anomalies per driver, to avoid cross driver leakage in the patterns. And the injections are going to only be injected where the distance is greater than the minimum distance that was predefined.

```
# Iterate per driver
for drv in df_out['Driver'].unique():
    dmask = df_out['Driver'] == drv
    dfd = df_out.loc[dmask].copy()

    # Only inject where there is enough distance covered
    valid = (dfd['Distance'] > min_dist) if 'Distance' in dfd else pd.Series(True, index=dfd.index)
    idx = dfd.index[valid]
    if len(idx) < 200:
        # Skip drivers with too few valid points for meaningful windows
        continue
```

Figure 26: Iterating Per Driver for Anomaly Injection Function

An inner function within the anomaly injection function will pick windows for the anomaly injection, based on valid indices[27].

```
# Function for picking windows from valid indices
def pick_window(length: int):
    length = int(length)
    start_i = int(rng.integers(0, max(1, len(idx) - length)))
    return idx[start_i:start_i + length]
```

Figure 27: Nested Function for Picking Windows

[27] https://www.geeksforgeeks.org/python/python-inner-functions/

8.1 Anomaly 1: Speed Spike

The ‘If’ statement run if the speed spike anomaly is to be injected into the ‘Speed’. If it is being injected, it will iterate over three spike amplitudes. For each of the amplitudes (40, 80, 120), it selects a random window of 15, and will increase the speed values by the chosen amounts, ensuring they stay within realistic speed bounds. The corresponding rows are updated with the relevant anomaly flag and descriptive labels.

```
# Anomaly 1: Speed Spike
if 'A1' in types and 'Speed' in dfd:
    for amp in [40, 80, 120]:  # km/h bumps
        win = pick_window(15); group_id += 1
        base = df_out.loc[win, 'Speed']
        df_out.loc[win, 'Speed'] = np.clip(base + amp, 0, 350)
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1,'A1_speed_spike',f'+{amp}kmh',group_id]
```

Figure 28: Anomaly 1 - Speed Spike

8.2 Anomaly 2: Speed Drop Out or Severe Sensor Failure

The second ‘If’ statement runs if the speed drop out anomaly is to be injected into the ‘Speed’. If the anomaly is being injected, for each window length (10, 20, or 40 rows), a random segment od that window size is selected, and the respective ‘Speed’ column is assigned extremely low values drawn from a uniform 0-5 km/h range. These rows with the injected anomalies are then marked as anomalous with the relevant anomaly flag and descriptive labels.

```
# Anomaly 2: Speed Drop Out or Severe Sensor Failure
if 'A2' in types and 'Speed' in dfd:
    for L in [10, 20, 40]:
        win = pick_window(L); group_id += 1
        df_out.loc[win, 'Speed'] = rng.uniform(0, 5, size=len(win))
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1,'A2_speed_zero',f'len={len(win)}',group_id]
```

Figure 29: Anomaly 2: Speed Drop Out or Severe Sensor Failure

8.3 Anomaly 3: Throttle Stuck During Deceleration

The Throttle stuck during deceleration anomaly is the next ‘If’ statement within the anomaly insertion function. This anomaly simulates scenarios where the throttle sensor remains fixed at unnaturally high values during vehicle deceleration. The block identifies deceleration periods based on segments where ‘Speed’ is dropping quickly. Potential early laps or pit lane regions were filtered out of the potential data pool, by using a distance threshold. If there are enough valid deceleration points within the data, then the anomaly placement is restricted to those points. Otherwise the anomaly placement will look at all available indices.

For each of the chosen throttle values, a 40 row window within the region is selected, and the ‘Throttle’ readings are overwritten with the fixed value, and the rows are labelled with the relevant anomaly flag and detailed metadata.

```
# Anomaly 3: Throttle Stuck During Deceleration
if 'A3' in types and {'Throttle','Speed'}.issubset(dfd.columns):
    dsp = df_out['Speed'].diff().fillna(0)
    decel_idx = df_out.index[(dsp < -2) & (df_out['Distance'] > 500)]
    cand = decel_idx.intersection(idx) if len(decel_idx) > 60 else idx
    for val in throttle_vals:
        start = int(rng.integers(0, max(1, len(cand) - 40)))
        win = cand[start:start + 40]; group_id += 1
        df_out.loc[win, 'Throttle'] = val
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1, 'A3_throttle_stuck', f'val={val}', group_id]
```

Figure 30: Anomaly 3 - Throttle Stuck During Deceleration

8.4 Anomaly 4: Brake Loss During Deceleration

Anomaly 4 is the brake loss during deceleration anomaly. This anomaly injects simulations of brake loss situations where the brake signal drops to zero during strong deceleration. This anomaly mimics hydraulic failure or sensor dropout. Deceleration zones are again calculated this time by computing the gradient of the 'Speed' signal and selecting the related indices where the speed is seen to decrease sharply. If enough deceleration points exist, then these become the chosen regions for placing the anomalies. If no then the full index range of the data is used. The appropriate zero value is determined in the code for the 'Brake' column, based on whether at this stage the value is stored as a boolean, integer, or float.

For each of window lengths (20, 30, 40 rows), the random deceleration segment is selected of that respective size. The brake values are substituted with the appropriate zero equivalent, and the rows are annotated to reflect the modification of the rows. The respective anomaly flag and the parameters are updated to clearly indicate each synthetic brake loss event.

```
# Anomaly 4: Brake Loss During Deceleration
if 'A4' in types and {'Brake','Speed'}.issubset(dfd.columns):
    sp = dfd['Speed'].astype(float).to_numpy()
    grad = np.diff(sp, prepend=sp[0])
    decel_idx = dfd.index[grad < -5]
    cand = decel_idx if len(decel_idx) >= 60 else idx

    # Determine zero equivalent for the brake column type
    brake_series = df_out['Brake']
    brake_loss_value = 0 if (brake_series.dtype == 'int64' or brake_series.dtype == 'bool') else 0.0

    for L in [20, 30, 40]:
        s = int(rng.integers(0, max(1, len(cand) - L)))
        win = cand[s:s + L]; group_id += 1
        df_out.loc[win, 'Brake'] = brake_loss_value
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1,'A4_brake_loss',f'len={len(win)}',group_id]
```

Figure 31: Anomaly 4 - Brake Loss During Deceleration

8.5 Anomaly 5: Gear Flicker

The gear flicker anomaly is the fifth anomaly that was defined within the anomaly injection function. This anomaly simulates noisy or unstable gear sensors that rapidly fluctuate between their adjacent gear values. For each of the specified jump magnitudes of 1, 2, or 3 gears, a windows of 20 rows are selected, to maintain some realistic behaviour. The current gear values are extracted from the data, and the missing entries, if any, are replaced with a default gear value. Random jitter is added to the gears from the predefined jump range. The resulting gear values are clipped to stay within the valid range of 1-8. All the modified rows are marked as anomalies and the recorded metadata describing the flicker intensity and anomaly group are tracked.

```
# Anomaly 5: Gear Flicker
if 'A5' in types and 'nGear' in dfd:
    for jumps in [1, 2, 3]:
        win = pick_window(20); group_id += 1
        base = df_out.loc[win, 'nGear'].fillna(1).astype(int)
        jitter = rng.integers(-jumps, jumps + 1, size=len(win))
        df_out.loc[win, 'nGear'] = np.clip(base + jitter, 1, 8)
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1, 'A5_gear_flicker', f'±{jumps}', group_id]
```

Figure 32: 8.5 Anomaly 5 - Gear Flicker

8.6 Anomaly 6: Packet Loss/ Frozen Values

The packet loss anomaly is emulating simulations where telemetry stops updating and the telemetry system simply repeats the previous sample. The columns that these packet loss anomalies can be simulated on are listed ('Speed', 'Throttle', 'Brake', 'nGear', 'RPM'), and the short windows are selected. Windows of 10 or 20 rows were selected to simulate a few seconds of stale data. The current values within the window are replaced with the previous rows value, which produces a realistic hold on the last sample effect, without needing to remove rows or disturb the indexes. All the affected rows are flagged with the relevant anomaly tags, thus documenting the nature and duration of the synthetic packet loss event.

```
# Anomaly 6: Packet Loss/ Frozen Values
if 'A6' in types:
    cols = [c for c in ['Speed','Throttle','Brake','nGear','RPM'] if c in df_out.columns]
    for L in [10, 20]:
        win = pick_window(L); group_id += 1
        for c in cols:
            prev = df_out[c].shift(1)
            df_out.loc[win, c] = prev.loc[win].values
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1, 'A6_packet_loss_hold', f'len={len(win)}', group_id]
```

Figure 33: 8.6 Anomaly 6 - Packet Loss/ Frozen Values

8.7 Anomaly 7: Delayed Data/ Sensor Lag

The final injected anomaly introduces delay/ jitter anomalies, where the telemetry appears to be temporally misaligned. This would be as a result of telemetry updates arriving late. Like anomaly 6, the relevant key telemetry columns are considered for this type anomaly ('Speed', 'Throttle', 'Brake', 'nGear', 'RPM'). For each chosen shift in the data, a 30 row window is selected, in which the values within the window are replaced with samples from earlier timestamps. A positive shift is utilised to create a realistic delayed update effect again without having to alter the row indices. The modified rows are then flagged as anomalies and annotated with metadata describing the shift applied and the anomaly group.

```
# Anomaly 7: Delayed Data/ Sensor Lag
if 'A7' in types:
    cols = [c for c in ['Speed','Throttle','Brake','nGear','RPM'] if c in df_out.columns]
    for shift in [2, 5]:
        win = pick_window(30); group_id += 1
        for c in cols:
            shifted_vals = df_out[c].shift(shift)
            df_out.loc[win, c] = shifted_vals.loc[win].values
        df_out.loc[win, ['is_anomaly','anomaly_type','anomaly_params','anomaly_group']] = \
            [1, 'A7_delay_jitter', f'shift={shift}', group_id]
```

Figure 34: 8.7 Anomaly 7 - Delayed Data/ Sensor Lag

8.8 Anomaly Injection Function Returns

The anomaly injection function returns a pandas data frame, which is a copy of test_df but with the anomalies injected, and the label columns added to the data. These columns as mentioned above are: ‘is_anomaly’. ‘anomaly_type’, ‘anomaly_params’, and ‘anomaly_group’.

```
    return df_out
```

Figure 35: Anomaly Injection Function Returns

The fault injected version of the test dataset is created so the anomaly detection pipeline can be validated against the known synthetic faults. All anomalies (A1 – A7) are applied to the data frame using the anomaly injection function. This ‘test_df_inj’ is now a test set that mixes the normal telemetry with the controlled labelled faults. The injected data is then passed through the PCA and Isolation Forest function, using three PCA components to capture more variance and improve the models ability to separate artificial anomalies from the defined normal behaviour.

```
# Injecting all the anomalies into the data frame using the anomaly injection function
test_df_inj = inject_anomalies_per_driver(
    test_df.copy(),
    types=['A1','A2','A3','A4','A5','A6','A7']
)

test_scores_df_inj = run_pca_if_per_driver(
    train_df,
    test_df_inj,
    feature_cols,
    n_components=3
)
```

Figure 36: Injecting Synthetic Anomalies into the Test Set

9 Physics Based Consistency Checks

The table below outlines the physics based consistency checks parameters for the ‘add_physics_checks’ function:

Table 7: Physics Based Consistency Checks Parameters

Parameter	Type	Description
df	pd.DataFrame	The data frame that needs the physics based checks to be conducted on
max_speed	Float default=360	The km/h upper bound for speed, with the typical maximum being between the range of 335-355km/h
freeze_window	Integer default=10	Defining the consecutive identical samples to flag as frozen, set to approximately 1 second windows
throttle_hi	Integer default=80	Defining the throttle threshold on a scale of 0-100

The physics based consistency checks are defined in a function, with their purpose being to catch obviously invalid data. The flags such as 'phys_high_accel' and 'phys_high_jerk' use robust data driven thresholds to make them session adaptive, and not hard coded to specific limits.

Table 8: Summary of Physics Based Consistency Checks Flags

Flag Column	What it detects	How its Computed	Parameters/ Thresholds
phys_out_of_range	Speed that is outside the physically plausible bounds	Flagging if 'Speed' < 0 or 'Speed' > 'max_speed'	'max_speed' Default is set to 360 km/h
phys_high_accel	Implausibly high acceleration magnitude	Converting 'Speed' (km/h) to m/s, and checking if finite difference with the 'dt' column	Threshold is taken from values greater than the 99.5th percentile in the session
phys_high_jerk	Implausibly high jerk (rate of change of acceleration)	Calculating the finite difference of acceleration, using the 'dt', and flagging if true	Threshold is taken from values greater than the 99.5th percentile in the session
phys_tb_conflict	Throttle brake inconsistency (hard throttle while braking)	Flag when 'Throttle' > 'throttle_hi'* and Brake = 1	*'throttle_hi' = 80, which is a throttle threshold on a 0–100 scale
phys_speed_freeze	Frozen speed (no updates across many samples)	Groups consecutive True runs and counts their lengths, producing a running counter of how long the speed has remained constant	flags runs with length ≥ 'freeze_window'
phys_any	Row wise aggregate of all above	Logical 'OR' of the five flags	Any physics issue present convenience mask

```python
def add_physics_checks(
    df,
    max_speed=360.0,
    freeze_window=10,
    throttle_hi=80
):
    out = df.copy()

    # Bounds on Speed
    out['phys_out_of_range'] = (out['Speed'] < 0) | (out['Speed'] > max_speed)

    # Time step estimation
    if 'Time' in out.columns and np.issubdtype(out['Time'].dtype, np.timedelta64):
        dt = out['Time'].diff().dt.total_seconds().replace(0, np.nan)
        dt = dt.fillna(dt.median() if pd.notna(dt.median()) else 0.1)
    else:
        dt = pd.Series(0.1, index=out.index)

    # Acceleration and jerk
    v_ms = out['Speed'] * (1000/3600)
    a = (v_ms.diff() / dt).replace([np.inf, -np.inf], np.nan).fillna(0)
    j = (a.diff() / dt).replace([np.inf, -np.inf], np.nan).fillna(0)

    # Session Adaptive Thresholds
    a_thr = np.nanpercentile(a.abs(), 99.5)
    j_thr = np.nanpercentile(j.abs(), 99.5)
    out['phys_high_accel'] = a.abs() > a_thr
    out['phys_high_jerk']  = j.abs() > j_thr

    # Throttle Brake Conflict
    if {'Throttle', 'Brake'}.issubset(out.columns):
        out['phys_tb_conflict'] = (out['Throttle'] > throttle_hi) & (out['Brake'] == 1)
    else:
        out['phys_tb_conflict'] = False

    # Frozen speed detection
    same = out['Speed'].diff().abs().fillna(0) <= 1e-6
    run = (same.groupby((~same).cumsum()).cumcount() + 1) * same
    out['phys_speed_freeze'] = run >= freeze_window

    # Aggregate flag
    phys_cols = [
        'phys_out_of_range', 'phys_high_accel', 'phys_high_jerk',
        'phys_tb_conflict', 'phys_speed_freeze'
    ]
    out['phys_any'] = out[phys_cols].any(axis=1)

    return out, phys_cols
```

Figure 37: Physics Based Consistency Checks

10 Fusing Physics Based Checks with IF

For fusing the physics based checks with the IF pipeline, started by setting a fixed FPR for the per driver data scores. The IF thresholds were calibrated per driver first on the clean data, with no injections. The threshold FPR was set at 5, i.e., the 5th percentile of defined 'normality'.

```
# Fixed FPR Per Driver
operating_q = 5
```

Figure 38: Fixing a Target False Positive Rate (FPR)

The 'test_scores_df_clean' runs PCA and IF on the clean test set with no injected anomalies, to obtain the baseline anomaly scores by driver.

```
# Calibrating IF thresholds per driver on clean scores
test_scores_df_clean = run_pca_if_per_driver(train_df, test_df, feature_cols)
```

Figure 39: Calibrating IF Thresholds on Clean Data

For each driver, all their clean IF scores are taken, the 5th percentile is computed, so that only 5% of the clean sample were being flagged as anomalies for each driver, and then these thresholds are stored in a dictionary. Each driver then has a personalised threshold based on their normal driving behaviour.

```
# Per driver threshold at the operating percenticle
calib = (
    test_scores_df_clean
    .groupby('Driver')['if_score']
    .apply(lambda s: np.percentile(s, operating_q))
    .to_dict()
)
```

Figure 40: Computing Per Driver Thresholds at the Desired Percentile

The physics flags are then added to the injected dataset to produce the original IF scores along with the additional boolean physics rule violations.

```
# Physics flags added to the injected score set
scored = test_scores_df_inj.copy()
scored, phys_cols = add_physics_checks(scored)
```

Figure 41: Adding Physics Based Flags to the Injected Test Set

Each driver is looped to retrieve the calibrated thresholds. Anomalies are flagged using the IF results alone 'det_if', then the anomalies are flagged using a fused method where the IF OR the physics based rules are detected 'det_fused'. In the instances where a driver did not appear in the calibration, a percentile fallback is deemed to be used. All per driver decisions are then merged into the final scored dataset.

```python
# IF OR physics per driver
rows = []
for d, g in scored.groupby('Driver'):
    thr = calib.get(d, np.percentile(g['if_score'], operating_q))
    gg = g.copy()
    gg['det_if']    = gg['if_score'] < thr          # IF only decision
    gg['det_fused'] = gg['det_if'] | gg['phys_any'] # Fuse with physics OR rule
    rows.append(gg)

scored = pd.concat(rows, ignore_index=True)
```

Figure 42: Applying IF-Only and Combined (IF OR Physics) Decisions

A function for computing per anomaly type true positive rates (TPR) at the event level is defined. Any sample in a given injected window is treated as a single detection event if it is correctly detected. The scored dataset is grouped by anomaly type and anomaly group. For each event, it is checked to see whether any row within that window had been flagged as anomalous. This is done by using the 'detected' column. The results are scored via 1 indicating a detection and 0 meaning the anomaly was missed. The average detection rate per anomaly is then computed, and a table of event level TPRs is produced.

```python
# Event Level TPR
def event_tpr(df, col_name='det_if'):
    events = []
    for (atype, gid), g in df.groupby(['anomaly_type', 'anomaly_group']):
        if atype == '' or gid == 0:
            continue  # skip non-injected / unlabeled rows
        events.append({'anomaly_type': atype, 'detected': int(g[col_name].any())})
    return (
        pd.DataFrame(events)
        .groupby('anomaly_type')['detected']
        .mean()
        .reset_index()
    )

# Compating IF only VS fused detector
tpr_if    = event_tpr(scored, 'det_if').rename(columns={'detected': f'TPR@{operating_q}%FPR'})
tpr_fused = event_tpr(scored, 'det_fused').rename(columns={'detected': 'TPR_fused'})

tpr_compare = tpr_if.merge(tpr_fused, on='anomaly_type')

# Sortig by the best performing anomaly types
display(tpr_compare.sort_values('TPR_fused', ascending=False))
```

Figure 43: Event Level TPR and Visualisation

Table 9: Resulting Event Level TPR Table

	anomaly_type	TPR@5%FPR	TPR_fused
2	A3_throttle_stuck	0.550000	1.000000
1	A2_speed_zero	0.650000	0.833333
6	A7_delay_jitter	0.700000	0.800000
4	A5_gear_flicker	0.633333	0.750000
0	A1_speed_spike	0.542373	0.559322
3	A4_brake_loss	0.433333	0.450000
5	A6_packet_loss_hold	0.282051	0.384615

10.1 Visual Comparison of Detection Performance by Anomaly Type

The comparison between the detection performance of each anomaly type is format in tabular format above, but has also been graphed for evaluation.

```python
# x-axis positions for each anomaly type
x = np.arange(len(tpr_compare))
w = 0.38  # bar width

plt.figure(figsize=(8, 4))

# IF only
plt.bar(
    x - w/2,
    tpr_compare[f'TPR@{operating_q}%FPR'],
    width=w,
    label='IF only'
)

# IF OR Physics Rules
plt.bar(
    x + w/2,
    tpr_compare['TPR_fused'],
    width=w,
    label='IF OR physics'
)

# Formatting
plt.xticks(x, tpr_compare['anomaly_type'], rotation=15)
plt.ylim(0, 1)
plt.ylabel('Detection rate')
plt.title('Detection by anomaly type')
plt.grid(axis='y', alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 44: 10.1 Visual Comparison of Detection Performance by Anomaly Type

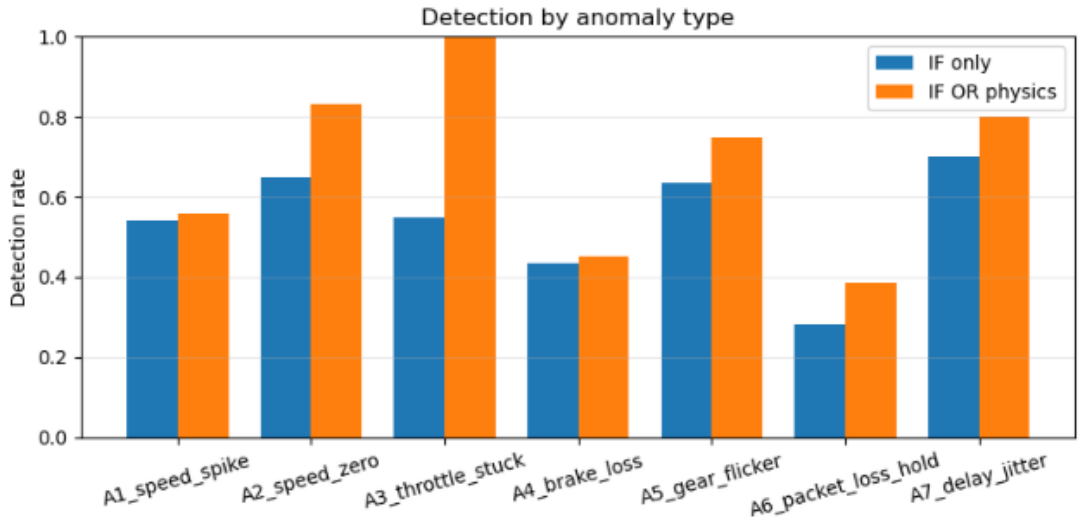

Figure 45: Resulting Bar Chart

11 Evaluation

The evaluation steps within the code will be addressed in the sections below.

11.1 Evaluation of Event Level Detection

The evaluation function for the event level detection evaluates the anomaly detection performance across multiple IF false positive operating points. For each driver, and each

injected anomaly window ('anomaly_group'), the function checks if any of the rows in that window crossed the IF threshold, and checks the threshold that were computed per driver to control the FPR (i.e., the percentiles of 'if_score'). Below are the parameters that are used for the function.

Table 10: Event Level Detection Evaluation Parameters

Parameter	Type	Description
df_scored	pd.DataFrame	The scored test set that contains the 'if_score', 'is_anomaly', 'anomaly_type', 'anomaly_group', 'Driver'
q_list	Tuple[int] default=(5, 2)	The percentile thresholds of IF scores for detection,, e.g. (10, 5, 2). Used for evaluating the detection performance at 10%, 5%, and 2% FPR operating points

As mentioned above, each driver is iterated over in a for loop, computing the driver specific IF score threshold corresponding to the requested operating percentiles. Within each drivers subset, the function processes each anomaly type, and separates the data into their individual anomaly events. These events windows record descriptive metrics for the anomaly detection, including summary statistics of the IF scores. The detection is noted at each operating point, by determining if any sample within the event window was to fall below the threshold, thus the event would be classified as detected.

```
def evaluate_detection(df_scored, q_list=(5, 2)):

    rows = []

    # Processing Per Driver
    for driver, ddrv in df_scored.groupby('Driver'):

        # Compute per driver threshold for each FPR operating point
        thresh = {q: np.percentile(ddrv['if_score'], q) for q in q_list}

        # Loop over anomaly types
        for atype, dat in ddrv.groupby('anomaly_type', dropna=False):
            if atype == '' or dat['is_anomaly'].sum() == 0:
                continue  # skip clean rows or no injected events

            # Evaluate event windows (grouped by anomaly_group)
            for gid, g in dat.groupby('anomaly_group'):
                if gid == 0:
                    continue

                row = {
                    'Driver': driver,
                    'anomaly_type': atype,
                    'group_id': int(gid),
                    'n_rows': len(g),
                    'avg_if_score': g['if_score'].mean(),
                    'min_if_score': g['if_score'].min(),
                }

                # Checking detection at each operating point
                for q in q_list:
                    row[f'detected@{q}%'] = int((g['if_score'] < thresh[q]).any())

                rows.append(row)
```

Figure 46: Evaluation of Event Level Detection

All the event level results are then aggregated into a consolidated data frame and are sorted by driver, anomaly type and event identifier.

```
    return (
        pd.DataFrame(rows)
        .sort_values(['Driver', 'anomaly_type', 'group_id'])
        .reset_index(drop=True)
    )
```

Figure 47: Evaluation Function Returns

The event level detection function was run on the fault injected test set and the resulting first 20 rows were produced.

```
results_table = evaluate_detection(test_scores_df_inj, q_list=(10, 5, 2, 0.2, 0.1, 0.05))
display(results_table.head(20))
```

Figure 48: Running the Evaluation Function and Display Summary of Table

Table 11: First 20 Rows of Table Printed

	Driver	anomaly_type	group_id	n_rows	avg_if_score	min_if_score	detected@10%	detected@5%	detected@2%	detected@0.2%	detected@0.1%	detected@0.05%
0	ALB	A1_speed_spike	267	15	-0.486245	-0.556413	1	0	0	0	0	0
1	ALB	A1_speed_spike	268	15	-0.493082	-0.609295	1	1	0	0	0	0
2	ALB	A1_speed_spike	269	15	-0.508213	-0.648909	1	1	1	1	0	0
3	ALB	A2_speed_zero	270	10	-0.512354	-0.567606	1	0	0	0	0	0
4	ALB	A2_speed_zero	271	17	-0.506129	-0.636170	1	1	1	0	0	0
5	ALB	A2_speed_zero	272	40	-0.496447	-0.625148	1	1	1	0	0	0
6	ALB	A3_throttle_stuck	273	40	-0.461733	-0.578903	1	0	0	0	0	0
7	ALB	A3_throttle_stuck	274	36	-0.466515	-0.615581	1	1	0	0	0	0
8	ALB	A3_throttle_stuck	275	40	-0.466014	-0.567835	1	0	0	0	0	0
9	ALB	A4_brake_loss	276	20	-0.476931	-0.593218	1	1	0	0	0	0
10	ALB	A4_brake_loss	277	30	-0.477112	-0.549674	1	0	0	0	0	0
11	ALB	A4_brake_loss	278	26	-0.459966	-0.533214	0	0	0	0	0	0
12	ALB	A5_gear_flicker	279	20	-0.479489	-0.583619	1	1	0	0	0	0
13	ALB	A5_gear_flicker	280	20	-0.482927	-0.579245	1	0	0	0	0	0
14	ALB	A5_gear_flicker	281	20	-0.470703	-0.600988	1	1	0	0	0	0
15	ALB	A6_packet_loss_hold	282	10	-0.488529	-0.514656	0	0	0	0	0	0
16	ALB	A6_packet_loss_hold	283	20	-0.479651	-0.549735	1	0	0	0	0	0
17	ALB	A7_delay_jitter	284	30	-0.461995	-0.616475	1	1	0	0	0	0
18	ALB	A7_delay_jitter	285	30	-0.482846	-0.596171	1	1	0	0	0	0
19	ALO	A1_speed_spike	153	8	-0.459984	-0.561432	1	0	0	0	0	0

11.2 Per Anomaly Type Evaluation

Anomaly groups (from A1-A7) are tested individually, to measure how well the models detects each anomaly group individually. This is done by injecting and evaluating each anomaly type one at a time. The anomaly groups are injected one at a time in a for loop, the clean test set is copied and only the chosen anomaly type is injected into the data, using the anomaly injection function.

The injected dataset is then passed through the PCA and IF pipeline to compute the relevant anomaly scores per driver. The evaluation function is then utilised to check how well the model is able to detect the injected events at the three FPR operating points.

The results are labelled with the type of experiment that was carried out, allowing for the comparison of anomaly type detections. All the results are aggregated into ‘all_results’, and concatenated into a single combined table.

```python
# List of groups to evaluate individually
types = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
all_results = []

for t in types:
    inj = inject_anomalies_per_driver(test_df.copy(), types=[t])

    scored = run_pca_if_per_driver(train_df, inj, feature_cols)

    res = evaluate_detection(scored, q_list=(10,5,2, 0.2, 0.1, 0.05))

    res['experiment'] = t

    all_results.append(res)

# Combining results from all experiments into one table
results_per_type = pd.concat(all_results, ignore_index=True)
```

Figure 49: Per Anomaly Type Evaluation

11.3 Combined Anomaly Injection Experiment Evaluation

After computing the per anomaly type evaluation, the combined anomaly injection experiment is evaluated. The model performance is evaluated when all anomaly families are injected simultaneously rather than previously where it was one anomaly family at a time. The fault injection dataset is passed into the anomaly detection pipeline, to give per driver anomaly scores. The detection performance is again measures at the specified FPR thresholds.

```python
# Injecting every anomaly type into the test set at once
inj_all = inject_anomalies_per_driver(test_df.copy(), types=types)

# Scoring the injected telemetry per driver
scored_all = run_pca_if_per_driver(train_df, inj_all, feature_cols)

# Evaluating the event level detection at the different operating points
results_all = evaluate_detection(scored_all, q_list=(10,5,2, 0.2, 0.1, 0.05))

# Tagging the experiement as 'ALL'
results_all['experiment'] = 'ALL'
```

Figure 50: Combined Anomaly Injection Evaluation

11.4 Final Aggregate Results Table

The results of the per anomaly type evaluation, and the combined anomaly injection experiment are aggregated into a table. All the results are concatenated, i.e., where the anomalies were tested individually, and where all the anomalies types were injected together.

```python
final_results = pd.concat(
    [results_per_type, results_all],
    ignore_index=True
)

#Preview of the merged results table
display(final_results.head())
```

Figure 51: Final Aggregate Results Table

Table 12: Final Aggregate Results Table Preview

	Driver	anomaly_type	group_id	n_rows	avg_if_score	min_if_score	detected@10%	detected@5%	detected@2%	detected@0.2%	detected@0.1%	detected@0.05%	experiment
0	ALB	A1_speed_spike	43	15	-0.500250	-0.648336	1	1	1	0	0	0	A1
1	ALB	A1_speed_spike	44	15	-0.502151	-0.579653	1	1	0	0	0	0	A1
2	ALB	A1_speed_spike	45	15	-0.489866	-0.668672	1	1	1	0	0	0	A1
3	ALO	A1_speed_spike	25	15	-0.482743	-0.572626	1	0	0	0	0	0	A1
4	ALO	A1_speed_spike	26	15	-0.490767	-0.542333	0	0	0	0	0	0	A1

11.5 Per Anomaly Detection Performance Summary

The per anomaly detection performance summary for the individual anomaly experiments were computed.

```
# Computing mean event level detection rates for each anomaly group at the different operating rates
summary =(results_per_type
    .groupby('experiment')[['detected@10%', 'detected@5%', 'detected@2%', 'detected@0.2%', 'detected@0.1%', 'detected@0.05%']]
    .mean()
)

# Display the summary table
summary
```

Figure 52: Per Anomaly Detection Performance Summary

Table 13: Results Table for Per Anomaly Detection Performance Summary

experiment	detected@10%	detected@5%	detected@2%	detected@0.2%	detected@0.1%	detected@0.05%
A1	0.866667	0.733333	0.533333	0.100000	0.016667	0.000000
A2	0.983333	0.916667	0.583333	0.033333	0.016667	0.016667
A3	0.983333	0.833333	0.316667	0.000000	0.000000	0.000000
A4	0.850000	0.466667	0.216667	0.033333	0.000000	0.000000
A5	0.816667	0.633333	0.300000	0.016667	0.000000	0.000000
A6	0.625000	0.350000	0.150000	0.000000	0.000000	0.000000
A7	0.950000	0.650000	0.250000	0.050000	0.050000	0.050000

11.6 Visualising all the Operating Points and their Detection Rates

The results for the per anomaly detection performance were also graphed in a grouped bar chart to better visualise the results from the table. The values were sorted by their FPR, and grouped in the Figure presented below.

```python
# Visualising all the Operating Points and their Detection Rates
df = summary.rename(columns={
    'detected@10%':   'TPR_10',
    'detected@5%':    'TPR_5',
    'detected@2%':    'TPR_2',
    'detected@0.2%':  'TPR_0_2',
    'detected@0.1%':  'TPR_0_1',
    'detected@0.05%': 'TPR_0_05'
}).copy()

# Mapping Columns to the numerical FPR
fpr_map = {
    'TPR_10':   0.10,
    'TPR_5':    0.05,
    'TPR_2':    0.02,
    'TPR_0_2':  0.002,
    'TPR_0_1':  0.001,
    'TPR_0_05': 0.0005
}

# Sort by FPR
ordered_cols = sorted(fpr_map.keys(), key=lambda c: fpr_map[c], reverse=True)

# Creating a Grouped Bar Chart
fig, ax = plt.subplots(figsize=(12,6))

x = range(len(df))
w = 0.12

for i, col in enumerate(ordered_cols):
    ax.bar([xi + (i - len(ordered_cols)/2) * w for xi in x],
           df[col],
           width=w,
           label=f"TPR @ {fpr_map[col]*100:.3g}% FPR")

ax.set_xticks(x)
ax.set_xticklabels(df.index)
ax.set_ylim(0,1)
ax.set_ylabel("Detection Rate")
ax.set_title("Detection Comparison Across Operating Points")
ax.legend(ncol=3)
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

Figure 53: Visualising all the Operating Points and their Detection Rates

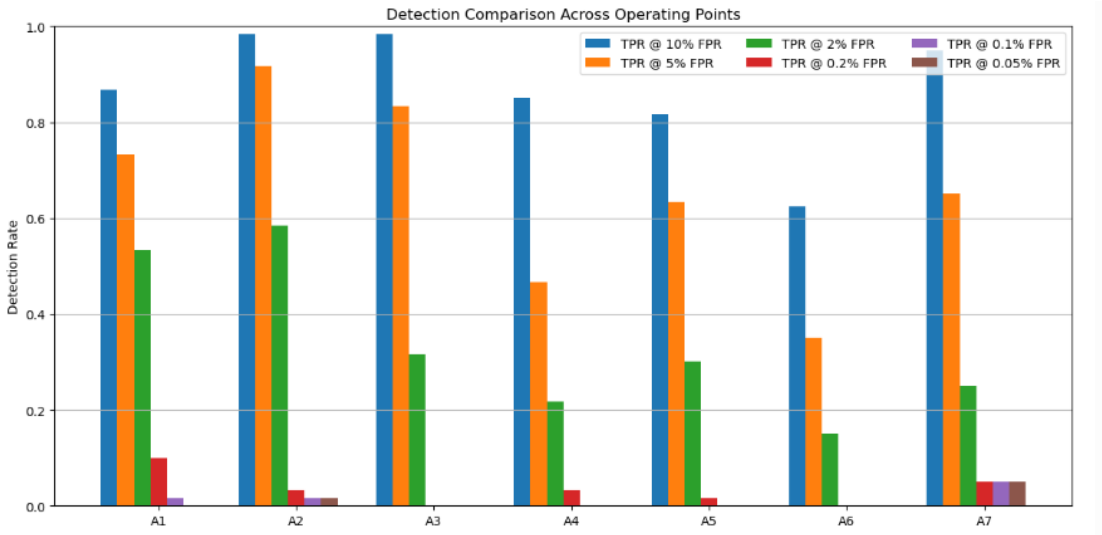

Figure 54: Resulting Grouped Bar Chart

Based on the Figure above it was decided to graph the results against only the TPRs at 10%, 5%, and 2%. Additionally a ROC Style curve for each anomaly type was also created.

```python
# Visualising all the Operating Points and their Detection Rates
df = summary.rename(columns={
    'detected@10%':   'TPR_10',
    'detected@5%':    'TPR_5',
    'detected@2%':    'TPR_2',
}).copy()

# Mapping Columns to the numerical FPR
fpr_map = {
    'TPR_10':   0.10,
    'TPR_5':    0.05,
    'TPR_2':    0.02,
}

# Sort by FPR
ordered_cols = sorted(fpr_map.keys(), key=lambda c: fpr_map[c], reverse=True)

# Creating a Grouped Bar Chart
fig, ax = plt.subplots(figsize=(10,5))

x = range(len(df))
w = 0.25

for i, col in enumerate(ordered_cols):
    ax.bar([xi + (i - len(ordered_cols)/2) * w for xi in x],
           df[col],
           width=w,
           label=f"TPR @ {fpr_map[col]*100:.3g}% FPR")

ax.set_xticks(x)
ax.set_xticklabels(df.index)
ax.set_ylim(0,1)
ax.set_ylabel("Detection Rate")
ax.set_title("Detection Comparison Across Operating Points")
ax.legend(ncol=3)
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

Figure 55: Visualiasation for 10%, 5%, and 2% Operating Points

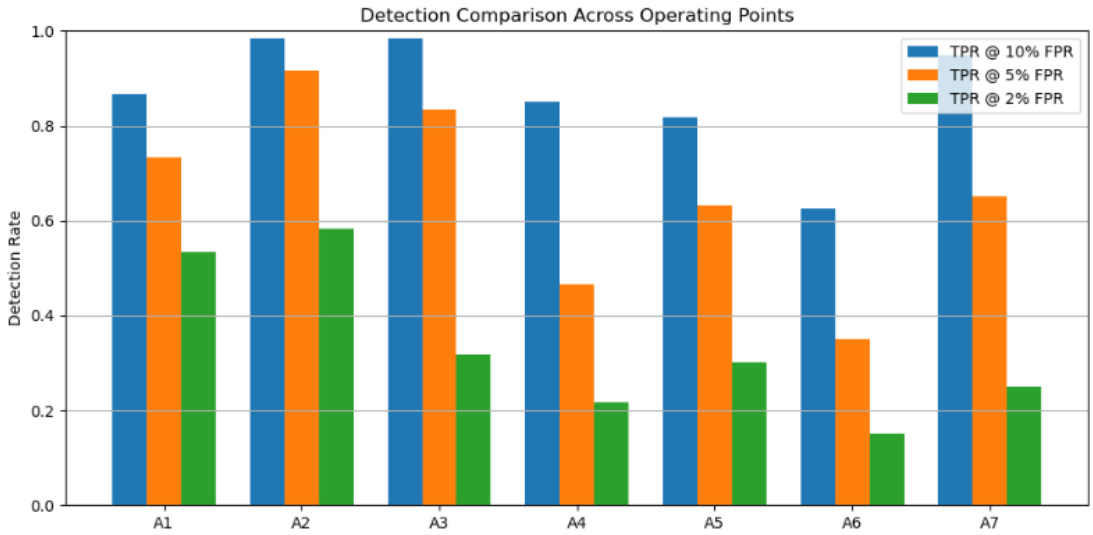

Figure 56: Resulting Grouped Bar Chart for 10%, 5%, and 2%

```python
# ROC Style Curve for each anomaly type
fig, ax = plt.subplots(figsize=(10,5))

FPRs = [0.10, 0.05, 0.02]

for anomaly_type, row in df.iterrows():
    ax.plot(FPRs, [row['TPR_10'], row['TPR_5'], row['TPR_2']],
            marker='o', label=anomaly_type)

ax.set_xlabel('False Positive Rate (FPR)')
ax.set_ylabel('True Positive Rate (TPR)')
ax.set_title('ROC Style Visualisation per Anomaly Type')
ax.set_ylim(0,1)
plt.grid()
plt.tight_layout()
plt.show()
```

Figure 57: ROC Style Curve for Each Anomaly Type

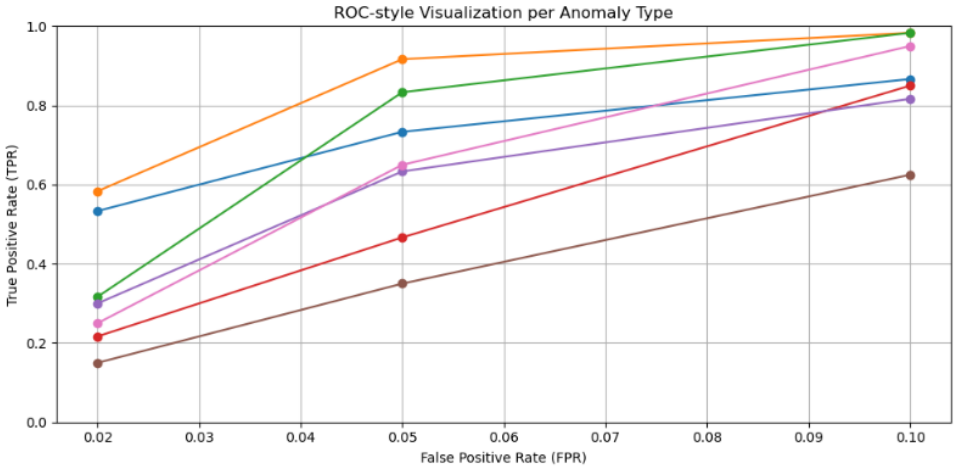

Figure 58: Resulting ROC Style Visualisation

11.7 IF Thresholds on Clean Test Set

The per driver anomaly score thresholds can be collected by analysing the clean test set, so that future detection can be calibrated fairly for each driver. The baseline anomaly scores are collected and stored in a dictionary. Can be compared with the results obtained in Table 9.

```python
# Passing clean test set through the PCA IF pipeline
test_scores_df_clean = run_pca_if_per_driver(train_df, test_df, feature_cols)

# False positive operating point
operating_q = 5

# Computing per driver IF threshold at this percentile
calib = (
    test_scores_df_clean
        .groupby('Driver')['if_score']
        .apply(lambda s: np.percentile(s, operating_q))
        .to_dict()
)

# This dict gives the anomaly score threshold for each driver
calib
```

Figure 59: IF Thresholds on Clean Test Set Per Driver

```
{'ALB': -0.5750624585772784,
 'ALO': -0.5747128655360392,
 'BOT': -0.5874752430574509,
 'GAS': -0.5839624905417492,
 'HAM': -0.5781384122463797,
 'HUL': -0.5802459278682196,
 'LEC': -0.5864980452925876,
 'MAG': -0.5876653028956428,
 'NOR': -0.5841132455240468,
 'OCO': -0.5799416321653724,
 'PER': -0.5897539388460874,
 'PIA': -0.575865534635248,
 'RIC': -0.581468297953661,
 'RUS': -0.5791761450902877,
 'SAI': -0.5837112477001793,
 'SAR': -0.5820305131308904,
 'STR': -0.5796413687968018,
 'TSU': -0.5769362681134066,
 'VER': -0.5786810634812174,
 'ZHO': -0.5882316588893033}
```

11.8 Visualising Injected Packet Loss and Delay/Jitter Anomalies

A visualisation was created to attempt to visualise two types of anomalies for a driver from the injected data. For this example, using driver ‘VER’. The data was sorted by distance, and the rows marked as anomalies were isolated, and the base speed was graphed. Then the two chosen anomalies were highlighted on the visualisation.

```python
driver = "VER"  # Change to inspect other drivers

dfp = test_df_inj[test_df_inj["Driver"] == driver].copy()

if "Distance" in dfp.columns:
    dfp = dfp.sort_values("Distance")

anom = dfp[dfp["is_anomaly"] == 1]

plt.figure(figsize=(10, 4))

plt.plot(
    dfp["Distance"],
    dfp["Speed"],
    label="Speed",
    linewidth=1
)

# Highlight A6
a6 = anom[anom["anomaly_type"].str.contains("A6", na=False)]
plt.scatter(
    a6["Distance"],
    a6["Speed"],
    label="A6 Packet-Loss/Freeze",
    marker="s",   # square marker
    s=40,
    edgecolors="green"
)

# Hinglight A7
a7 = anom[anom["anomaly_type"].str.contains("A7", na=False)]
plt.scatter(
    a7["Distance"],
    a7["Speed"],
    label="A7 Delay/Jitter",
    marker="D",   # diamond marker
    s=50,
    edgecolors="red"
)

# Formatting
plt.xlabel("Distance (m)")
plt.ylabel("Speed (km/h)")
plt.title(f"{driver}  Speed with Packet-Loss + Jitter Anomalies Highlighted")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

Figure 60: Visualising Injected Packet Loss and Delay/Jitter Anomalies

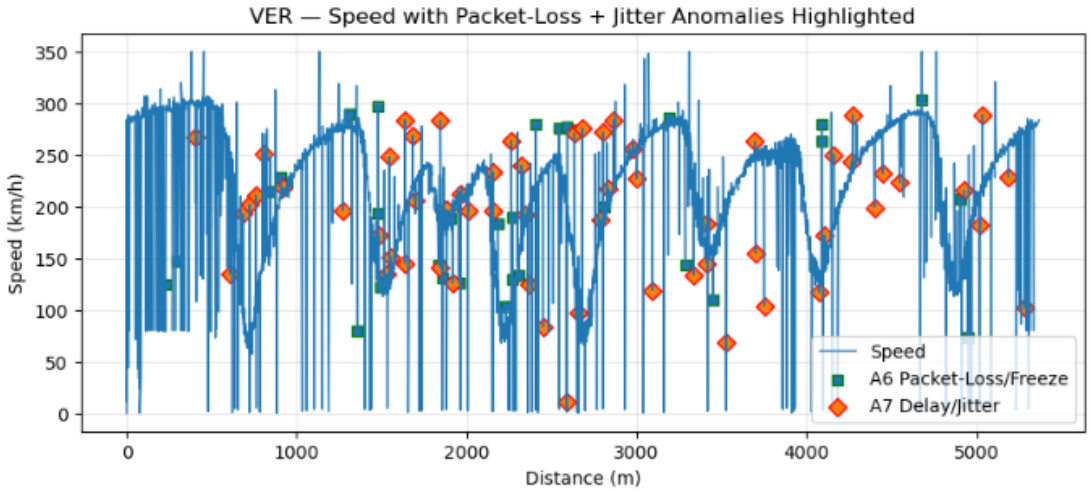

Figure 61: A6 and A7 Anomalies for 'VER'

12 References

Bhansali, S. (2023, June 23). *Data analysis using F1 Telemetry*. Retrieved from Medium: https://medium.com/@samyak.bhansali.201559/data-analysis-using-f1-telemetry-462da3af6b17

Bobbitt, Z. (2021, August 9). *How to Use describe() Function in Pandas (With Examples)*. Retrieved from Statology: https://www.statology.org/pandas-describe/

Chiuman, N. (2025, July 17). *Formula 1 Data Analysis with FastF1*. Retrieved from Medium: https://medium.com/formula-one-forever/formula-1-data-analysis-with-fastf1-%EF%B8%8F-d451b30f3a91

Clark, B. (2025, November 17). *What is Scikit-Learn (Sklearn)?* Retrieved from IBM: https://www.ibm.com/think/topics/scikit-learn

Garcia, R. (2023, October 11). *Telemetry of the fastest lap of a F1 Grand Prix | FastF1 Tutorials*. Retrieved from Medium: https://python.plainenglish.io/telemetry-of-the-fastest-lap-of-a-f1-grand-prix-fastf1-tutorials-4211a48f5eac

GeeksforGeeks. (2025, July 26). *Indexing and Selecting Data with Pandas*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/pandas/indexing-and-selecting-data-with-pandas/

GeeksforGeeks. (2025, July 26). *Python | Pandas dataframe.info()*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/python/python-pandas-dataframe-info/

GeeksforGeeks. (2025, September 1). *Python Inner Functions*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/python/python-inner-functions/

Horvath, R. (2020, September 14). *Plot With pandas: Python Data Visualization for Beginners*. Retrieved from Real Python: https://realpython.com/pandas-plot-python/

Horvath, R. (2020, January 6). *Using pandas and Python to Explore Your Dataset*. Retrieved from Real Python: https://realpython.com/pandas-python-explore-dataset/

Nithurshen. (n.d.). *Scikit-learn: machine learning in Python*. Retrieved from GitHub: https://github.com/scikit-learn/scikit-learn

Pandas. (2025). *Pandas*. Retrieved from Pandas: https://pandas.pydata.org/

Pandas. (n.d.). *Chart visualization*. Retrieved from Pandas:

https://pandas.pydata.org/docs/user_guide/visualization.html

Pandas. (n.d.). *Indexing and selecting data*. Retrieved from Pandas: https://pandas.pydata.org/docs/user_guide/indexing.html

Pandas. (n.d.). *pandas.DataFrame.assign*. Retrieved from Pandas: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.assign.html

Pandas. (n.d.). *pandas.DataFrame.corr*. Retrieved from Pandas: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html#pandas.DataFrame.corr

Pandas. (n.d.). *pandas.DataFrame.select_dtypes*. Retrieved from Pandas: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html#pandas.DataFrame.select_dtypes

Pandas. (n.d.). *pandas.Series.dt.total_seconds*. Retrieved from Pandas: https://pandas.pydata.org/pandas-docs/version/2.1.2/reference/api/pandas.Series.dt.total_seconds.html#pandas.Series.dt.total_seconds

ScikitLearn. (n.d.). *IsolationForest*. Retrieved from ScikitLearn: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

ScikitLearn. (n.d.). *PCA*. Retrieved from ScikitLearn: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

ScikitLearn. (n.d.). *train_test_split*. Retrieved from ScikitLearn: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Seaborn. (n.d.). *seaborn.heatmap*. Retrieved from Seaborn: https://seaborn.pydata.org/generated/seaborn.heatmap.html

Seaborn. (n.d.). *seaborn: statistical data visualization*. Retrieved from https://seaborn.pydata.org/

Stojiljković , M. (2019, December 23). *NumPy, SciPy, and pandas: Correlation With Python*. Retrieved from Real Python: https://realpython.com/numpy-scipy-pandas-correlation-python/

theOehrly. (n.d.). *theOehrly/Fast-F1: FastF1 is a python package for accessing and analyzing Formula 1 results, schedules, timing data and telemetry*. Retrieved from GitHub: https://github.com/theOehrly/Fast-F1

Yadav, A. (2025, February 19). *How to Convert pandas Timedelta to Seconds?* Retrieved from Medium: https://medium.com/@amit25173/how-to-convert-pandas-timedelta-to-seconds-b37f159fe2b4