Practical AI CA2

Team Members:

- Jeyakumar Sriram
- Shawn Lim Jun Jie
- Yeoh Wei Zheng Benjamin
- Aarron Loke Ruixuan

Libraries

In [6]:
```
### Data Manipulation and Analysis:
import pandas as pd
import numpy as np
import sqlalchemy as sal

### Data Visualization:
import matplotlib.pyplot as plt
import seaborn as sns

### Date and Time:
from datetime import datetime

### Machine Learning:
from sklearn.metrics import accuracy_score, make_scorer, recall_score, fbeta_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler, Ordina
from sklearn.ensemble import IsolationForest, RandomForestClassifier, AdaBoostClassif
from sklearn.neighbors import LocalOutlierFactor, KNeighborsClassifier
from sklearn.linear_model import LogisticRegression, Perceptron, RidgeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, OneClassSVM
from sklearn.tree import DecisionTreeClassifier
from sklearn.compose import ColumnTransformer
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
from sklearn.pipeline import Pipeline

# MLFlow
import mlflow
from mlflow.models import infer_signature
mlflow.set_tracking_uri(uri="http://127.0.0.1:8080")

### Other:
import math
from time import time
import pickle
import warnings; warnings.filterwarnings("ignore")
```

Data Loading

Later replace with SQL Server. Use CA1 code.

In [5]:
```
query_sensor = "SELECT * FROM sensor"
query_safety = "SELECT * FROM safety_status"
query_driver = "SELECT * FROM drivers"
```

```
engine = sal.create_engine(
    "mssql+pyodbc://DESKTOP-V0N62F3\SQLEXPRESS/cab_data?driver=ODBC+Driver+17+for+SQL
)

drivers = pd.read_sql(query_driver, engine)
safety = pd.read_sql(query_safety, engine)

sensors = pd.read_sql(query_sensor, engine, chunksize=100000)
sensors = pd.concat([chunk for chunk in sensors])

# Use if database not available
# drivers = pd.read_hdf("./h5/driversRaw.h5")
# sensors = pd.read_hdf("./h5/sensorsRaw.h5")
# safety = pd.read_hdf("./h5/safetyRaw.h5")
```

Feature Engineering

Sensor Data

Derived Features:

- New columns are introduced to provide valuable insights into the sensor data:
  - **Net Acceleration ((net_acc)):** Represents the net acceleration magnitude calculated as follows:

$$\sqrt{acceleration_x^2 + acceleration_y^2 + acceleration_z^2}$$

  - **Net Gyroscope ((net_gyro)):** Represents the net gyroscope magnitude calculated as follows:

$$\sqrt{gyro_x^2 + gyro_y^2 + gyro_z^2}$$

  - **Roll ((roll)):** Represents the roll angle, indicating the tilt of the sensor around the x-axis, calculated as follows:

$$\frac{180}{\pi} \cdot \arctan\left(\frac{acceleration_y}{\sqrt{acceleration_x^2 + acceleration_z^2}}\right)$$

  - **Pitch ((pitch)):** Represents the pitch angle, indicating the tilt of the sensor around the y-axis, calculated as follows:

$$\frac{180}{\pi} \cdot \arctan\left(\frac{acceleration_x}{\sqrt{acceleration_y^2 + acceleration_z^2}}\right)$$

  - **Yaw ((yaw)):** Represents the yaw angle, indicating the rotation of the sensor around the z-axis, calculated as follows:

$$\frac{180}{\pi} \cdot \arctan\left(\frac{acceleration_z}{\sqrt{acceleration_x^2 + acceleration_z^2}}\right)$$

In [67]:
```
net_acc = lambda x: math.sqrt(
    x["acceleration_x"] ** 2 + (x["acceleration_y"]) ** 2 + (x["acceleration_z"]) **
)
net_gyro = lambda x: math.sqrt(x["gyro_x"] ** 2 + x["gyro_y"] ** 2 + x["gyro_z"] ** 2
roll = (
    lambda x: 180
    * math.atan(
        x["acceleration_y"]
        / math.sqrt(
            x["acceleration_x"] * x["acceleration_x"]
            + x["acceleration_z"] * x["acceleration_z"]
        )
    )
    / math.pi
)
pitch = (
    lambda x: 180
    * math.atan(
        x["acceleration_x"]
        / math.sqrt(
            x["acceleration_y"] * x["acceleration_y"]
            + x["acceleration_z"] * x["acceleration_z"]
        )
    )
    / math.pi
)
yaw = (
    lambda x: 180
    * math.atan(
        x["acceleration_z"]
        / math.sqrt(
            x["acceleration_x"] * x["acceleration_x"]
            + x["acceleration_z"] * x["acceleration_z"]
        )
    )
    / math.pi
)

sensors.loc[:, "net_acc"] = sensors.apply(net_acc, axis=1)
sensors.loc[:, "net_gyro"] = sensors.apply(net_gyro, axis=1)
sensors.loc[:, "roll"] = sensors.apply(roll, axis=1)
sensors.loc[:, "pitch"] = sensors.apply(pitch, axis=1)
sensors.loc[:, "yaw"] = sensors.apply(yaw, axis=1)
```

```
/var/folders/j8/cpvjf5791k5cyvj2dbrz0_mr0000gn/T/ipykernel_24175/1274286828.py:3: Runt
imeWarning: divide by zero encountered in scalar divide
  roll = lambda x: 180 * math.atan(x['acceleration_y'] / math.sqrt(x['acceleration_x']
* x['acceleration_x'] + x['acceleration_z'] * x['acceleration_z'])) / math.pi
/var/folders/j8/cpvjf5791k5cyvj2dbrz0_mr0000gn/T/ipykernel_24175/1274286828.py:5: Runt
imeWarning: invalid value encountered in scalar divide
  yaw = lambda x: 180 * math.atan(x['acceleration_z'] / math.sqrt(x['acceleration_x']*
x['acceleration_x'] + x['acceleration_z']*x['acceleration_z']))/math.pi
```

In [68]:
```
shape_initial = sensors.shape
```

Driver Data

1. **Label Encoding:**

   - Categorical variables 'gender' and 'car_brand' are encoded using the `LabelEncoder` to convert them into numerical representations.

2. **Data Type Conversion:**

   - 'driver_name' is converted to a string.
   - 'date_of_birth' is converted to a datetime object.
   - 'driver_age' is calculated based on the difference between the current year and 'date_of_birth'.

- 'car_age' is calculated based on the difference between the current year and 'car_model_year'.

3. **Memory Optimization:**

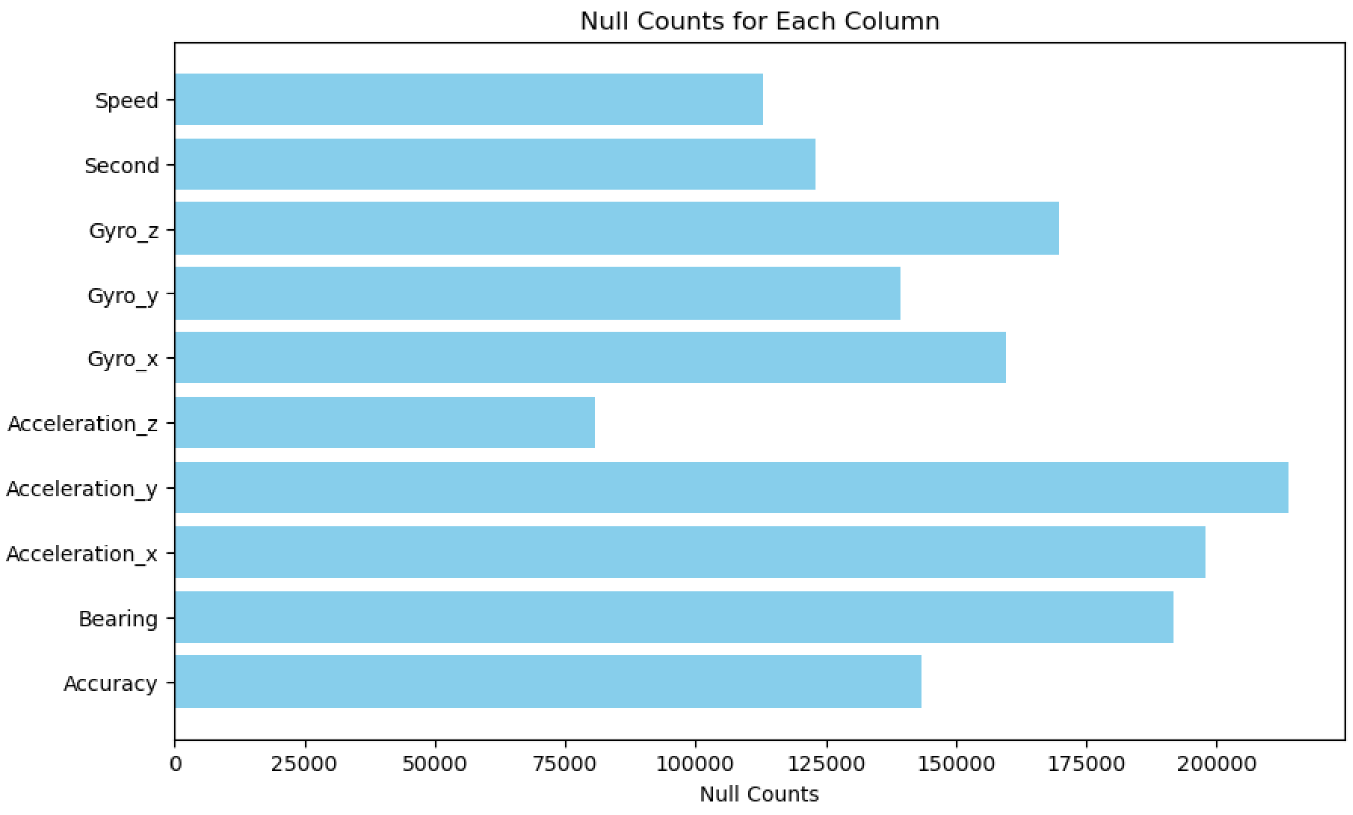
   - Column data types are optimized for memory usage. Specific data types are assigned to columns, such as 'car_age' (int8), 'driver_age' (int8), 'car_brand' (int8), 'car_model_year' (int16), 'gender' (bool), 'driver_rating' (float32), and 'no_of_years_driving_exp' (int8).

In [70]:
```python
drivers["driver_name"] = drivers["driver_name"].astype(str)

label_encoder = LabelEncoder()
drivers["gender"] = label_encoder.fit_transform(drivers["gender"])
drivers["car_brand"] = label_encoder.fit_transform(drivers["car_brand"])

current_date = datetime.now().year
drivers["data_of_birth"] = pd.to_datetime(drivers["data_of_birth"])
drivers["driver_age"] = (current_date - drivers["data_of_birth"].dt.year).astype(int)
drivers["car_age"] = current_date - drivers["car_model_year"]

column_types = {
    "car_age": np.int8,
    "driver_age": np.int8,
    "car_brand": np.int8,
    "car_model_year": np.int16,
    "gender": bool,
    "driver_rating": np.float32,
}
drivers = drivers.astype(column_types)
```

Imputation & Outlier Detection

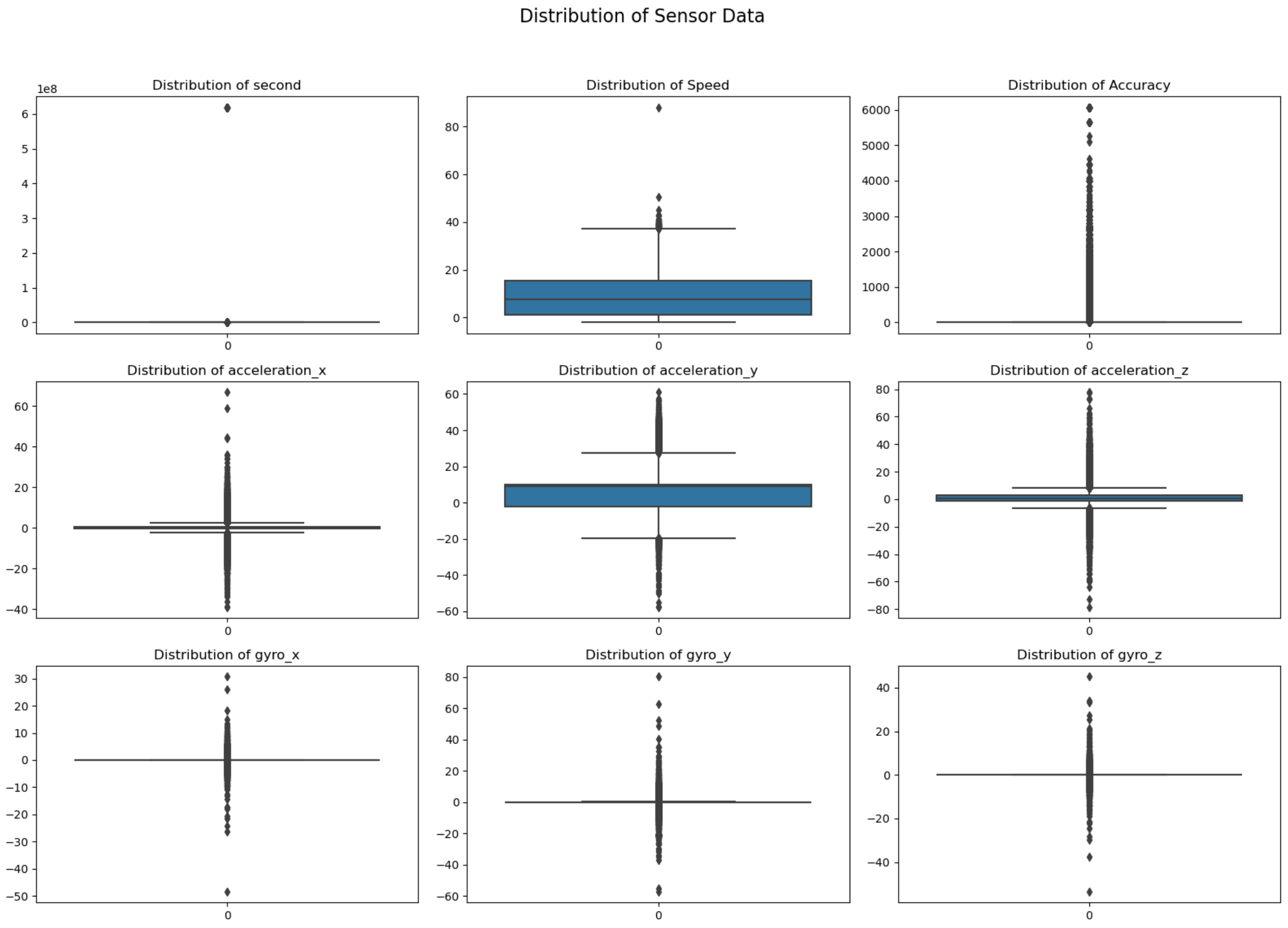
The following code generates a horizontal bar graph illustrating the null counts for each column in the `sensors` DataFrame.

In [71]:
```python
columns = [
    "Accuracy",
    "Bearing",
    "acceleration_x",
    "acceleration_y",
    "acceleration_z",
    "gyro_x",
    "gyro_y",
    "gyro_z",
    "second",
    "Speed",
]
null_counts = [sensors[col].isnull().sum() for col in columns]

# Create a bar graph
plt.figure(figsize=(10, 6))
plt.barh([i.capitalize() for i in columns], null_counts, color="skyblue")
plt.xlabel("Null Counts")
plt.title("Null Counts for Each Column")
plt.show()
```



The following visualizations represent the distribution of various sensor data using boxplots. Each subplot corresponds to a specific sensor data attribute.

In [72]:
```python
# Create a subplot with 3 rows and 4 columns
fig, axes = plt.subplots(3, 3, figsize=(16, 12))
fig.suptitle("Distribution of Sensor Data", fontsize=16)

# Flatten the axes for easy indexing
axes = axes.flatten()

# Define the columns for boxplots
columns = [
    "second",
    "Speed",
    "Accuracy",
    "acceleration_x",
    "acceleration_y",
    "acceleration_z",
    "gyro_x",
    "gyro_y",
    "gyro_z",
]

# Plot each boxplot in a loop
for i, column in enumerate(columns):
    sns.boxplot(sensors[column], ax=axes[i])
    axes[i].set_title(f"Distribution of {column}")

# Adjust layout for better spacing
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Null Values:

- **Accuracy:**
  - Interpolated using the polynomial method to fill missing values.
- **Speed:**
  - Outliers above 40 are set to `None` and then interpolated using the linear method.
- **Second:**
  - Rows where 'second' is less than or equal to 100,000 are removed to address missing values.
- **Acceleration_x, Acceleration_y, Acceleration_z, Gyro_x:**
  - Outliers above 40 for 'acceleration_x' are set to `None` and then interpolated using the polynomial method.
  - 'acceleration_y', 'acceleration_z', and 'gyro_x' are also interpolated using the polynomial method.
- **Gyro_x, Gyro_y, Gyro_z:**
  - Outliers below -40 and above 40 are set to `None`.
  - Interpolation is performed using the linear method.

In [73]:
```
# Interpolate Accuracy using polynomial method
sensors["Accuracy"].interpolate(method="polynomial", order=5, inplace=True)

# Remove missing seconds where 'second' is less than or equal to 100000
sensors = sensors[sensors["second"] < 100000]

# Interpolate Speed using linear method for handling outliers
outliers_speed = sensors["Speed"] > 40
```

```
sensors.loc[outliers_speed, "Speed"] = None
sensors["Speed"].interpolate(method="linear", inplace=True)

# Interpolate Bearing
bearing_rad = np.radians(sensors["Bearing"])
sin_component = np.sin(bearing_rad)
cos_component = np.cos(bearing_rad)

components = pd.DataFrame({"sin": sin_component, "cos": cos_component})
components_filled = components.fillna(method="ffill").fillna(method="bfill")
components_interpolated = components_filled.interpolate()
interpolated_bearings = np.degrees(
    np.arctan2(components_interpolated["sin"], components_interpolated["cos"])
)
sensors["Bearing"] = interpolated_bearings % 360

# Interpolate acceleration_x, acceleration_y, acceleration_z, gyro_x for handling nul
outliers_x = sensors["acceleration_x"] > 40
sensors.loc[outliers_x, "acceleration_x"] = None

sensors["acceleration_x"].interpolate(method="polynomial", order=5, inplace=True)
sensors["acceleration_y"].interpolate(method="polynomial", order=5, inplace=True)
sensors["acceleration_z"].interpolate(method="polynomial", order=5, inplace=True)
sensors["gyro_x"].interpolate(method="polynomial", order=5, inplace=True)
```

```
/var/folders/j8/cpvjf5791k5cyvj2dbrz0_mr0000gn/T/ipykernel_24175/1197173152.py:18: Fut
ureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future ve
rsion. Use obj.ffill() or obj.bfill() instead.
  components_filled = components.fillna(method='ffill').fillna(method='bfill')
```

Outliers:

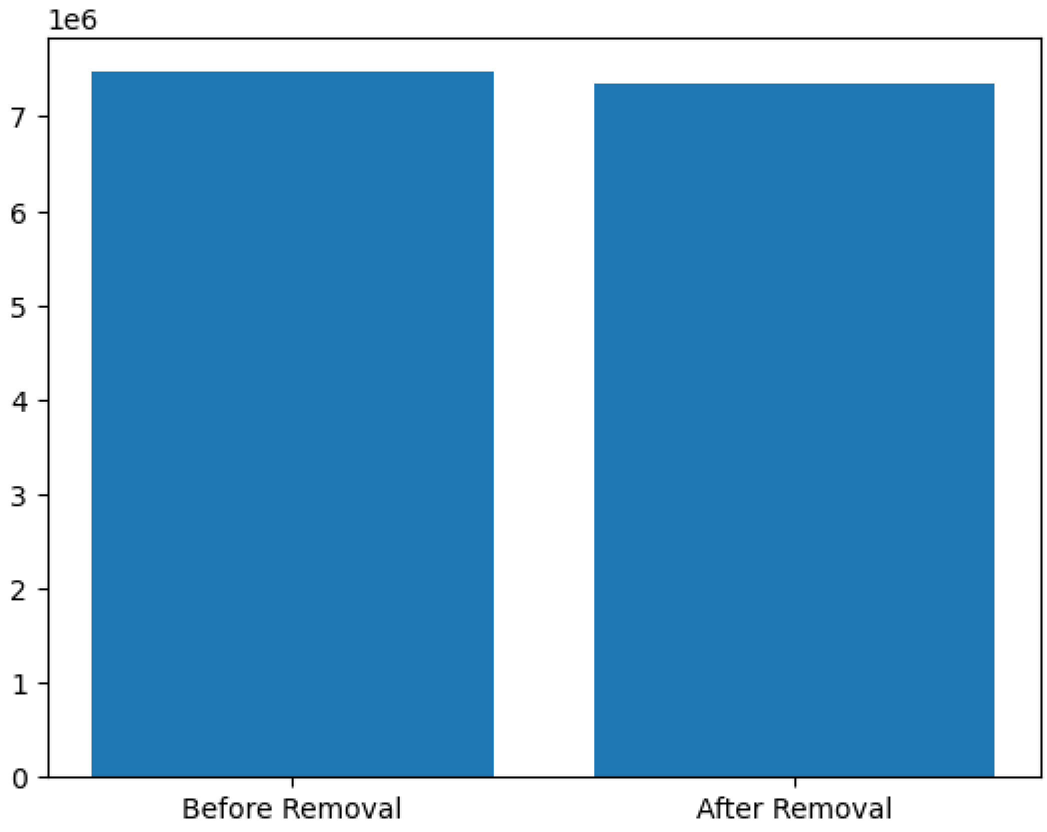
- **BookingID:**
  - Rows with 'bookingID' equal to 163208757335 or 858993459333 are removed.
- **Speed:**
  - Outliers above 40 are identified and handled by setting them to `None`.
- **Gyro_x, Gyro_y, Gyro_z:**
  - Outliers below -40 and above 40 are identified and set to `None`.
- **Note:**
  - Outlier handling involves setting values to `None` before interpolation.
  - Data types are adjusted accordingly to accommodate interpolated values.

In [75]:
```
# Remove outliers based on bookingID
sensors = sensors[sensors["bookingID"] != 163208757335]
sensors = sensors[sensors["bookingID"] != 858993459333]

# Handle outliers for gyro_x, gyro_y, gyro_z
gyro_columns = ["gyro_x", "gyro_y", "gyro_z"]

for column in gyro_columns:
    outliers_low = sensors[column] < -40
    outliers_high = sensors[column] > 40

    sensors.loc[outliers_low, column] = None
    sensors.loc[outliers_high, column] = None

    sensors[column].interpolate(method="linear", inplace=True)
```

In [77]:
```
post_removal_shape = sensors.shape
```

In [83]:
```
before_after = [shape_initial[0], post_removal_shape[0]]
print(f"Total Number of Values Removed: {shape_initial[0] - post_removal_shape[0]}")
print(
    f"Percentage of Values Removed: {round((shape_initial[0] - post_removal_shape[0])
)
plt.bar(["Before Removal", "After Removal"], before_after)
plt.show()
```

```
Total Number of Values Removed: 123703
Percentage of Values Removed: 1.66%
```



Data Aggregation

The function `count_harsh_acc` is designed to count the number of occurrences of harsh acceleration. It considers accelerations greater than 15.

In [84]:
```
# create custom functions for aggregate, which in this case is used to count the numb
def count_harsh_acc(x):
    return x[x > 15].count()


def count_harsh_gyro(x):
    return x[x > 1].count()
```

Data aggregation is applied to sensor data for the purpose of analyzing driver safety. The key steps involve summarizing relevant information from the sensor dataset, joining it with driver information, and creating a consolidated DataFrame for further analysis.

1. Sensor Data Aggregation:

- The sensor data is first filtered to include only relevant columns.
- Grouping by 'bookingID', various aggregation functions are applied to different columns, such as calculating maximum values and counting occurrences of harsh acceleration and gyroscope

events.
- The resulting DataFrame is assigned appropriate column names for clarity

2. Joining with Driver Information:

- Driver information, including safety-related data, is joined with the aggregated sensor data based on 'driver_id' and 'bookingID'.
- The resulting DataFrame 'df' contains both aggregated sensor information and driver-related safety data.

3. Data Cleaning:

- Unnecessary columns ('bookingID', 'driver_id', 'driver_name', 'data_of_birth', 'car_model_year') are dropped, leaving a clean and focused dataset for further analysis.

In [85]:
```
agg_sensors = sensors[
    ["bookingID", "second", "Speed", "net_acc", "net_gyro", "roll", "yaw", "pitch"]
]
agg_sensors = (
    agg_sensors.groupby("bookingID")
    .agg(
        {
            "second": "max",
            "Speed": "mean",
            "net_acc": ["max", count_harsh_acc],
            "net_gyro": ["max", count_harsh_gyro],
            "roll": "mean",
            "yaw": "mean",
            "pitch": "mean",
        }
    )
    .reset_index()
)
agg_sensors.columns = [
    f"{col[0]}_{col[1]}" if col[1] != "" else f"{col[0]}" for col in agg_sensors.colu
]

driver_safety = drivers.join(safety.set_index("driver_id"), on="driver_id", how="inne
df = agg_sensors.join(driver_safety.set_index("bookingID"), on="bookingID", how="inne
df = df.drop(
    columns=["bookingID", "driver_id", "driver_name", "data_of_birth", "car_model_yea
)
```

We will be saving the code in HDF5 format.

In [86]:
```
df.to_hdf("./h5/cleaned_dataset.h5", "w")
```

In [87]:
```
df.shape
```

Out[87]:
```
(19998, 16)
```

Outlier Detection (Post-Aggregation)

After aggregatin our data for the ML model, we will also perform outlier detecion on this data to find multi-dimensional outliers which may hinder our model's performance.

In [43]:
```
df = pd.read_hdf("./h5/cleaned_dataset.h5")
```

In [44]:
```
df.head()
```

Out[44]:

	second_max	Speed_mean	net_acc_max	net_acc_count_harsh_acc	net_gyro_max	net_gyro_
0	1589.0	9.162333	12.258996	0	0.538088	
1	1034.0	7.715448	11.844354	0	0.608687	
2	822.0	2.851901	11.051480	0	0.420290	
3	1092.0	6.027042	13.493185	0	0.661675	
4	1092.0	4.799132	13.393135	0	0.626294	

I will be applying some basic transformations to pass in the data into the outlier detection models. This will include one hot encoding and label encoding.

In [45]:
```
one_hot_encoder = OneHotEncoder(sparse=False, drop="first")
df["gender"] = df["gender"].replace({True: 1, False: 0})
one_hot_car = one_hot_encoder.fit_transform(df["car_brand"].to_numpy().reshape(-1, 1)
encoded_columns = one_hot_encoder.get_feature_names_out(["car_brand"])
encoded_df = pd.DataFrame(one_hot_car, columns=encoded_columns)
df = df.drop("car_brand", axis=1).join(encoded_df)
```

Spliting Dataset into Danger & Non-Danger

When detecting the outliers you might think we should do it on the same dataset. But doing so would lead to singificantly higher outlier counts in the minority class compared to the majority class. This can lead to useful information in the minority class be classified as Outliers. To combat this, we will perform outlier detection on the Dangerous and Non-Dangerous datasets separately.

In [46]:
```
danger_df = df.loc[df["label"] == 1]
nondanger_df = df.loc[df["label"] == 0]
```

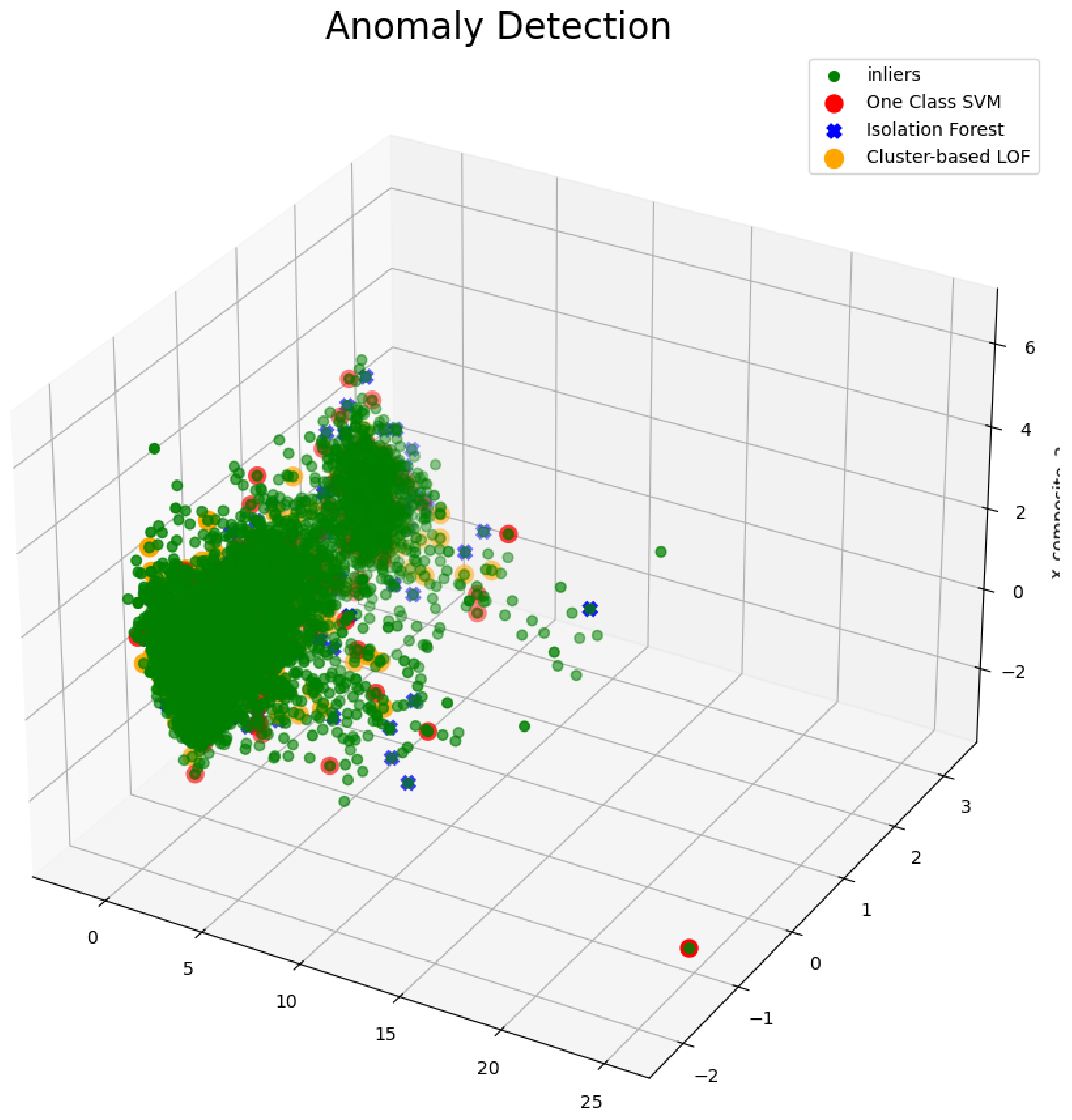
Algorithms Used

- **One-Class SVM:** SVM-based method learning a decision function to separate normal data from outliers.
- **Isolation Forest:** Random forest-based technique isolating anomalies by randomly partitioning features.
- **Cluster-based Local Outlier Factor (CBLOF):** Identifies outliers based on their deviation from cluster centroids in high-dimensional data.

To visualise the difference between these methods, we will be plotting a 3D plot comparing which points were classified as outliers by which methods.

In [47]:
```
def detectOutlier(current_df):
    classifier_dict = {
        "One-Class SVM": OneClassSVM(nu=0.05),
        "Isolation Forest": IsolationForest(contamination=0.05),
        "Cluster-based LOF": LocalOutlierFactor(contamination=0.05),
    }
    Outliers = pd.DataFrame()
    for i, (clf_name, clf) in enumerate(classifier_dict.items()):
        clf.fit(current_df)
        current_df["anomaly"] = clf.fit_predict(current_df)
        outliers_subset = current_df[current_df.anomaly == -1]
        outliers_subset = outliers_subset.assign(model=clf_name)
        Outliers = pd.concat([Outliers, outliers_subset], axis=0, ignore_index=False)
        Outliers.reset_index(drop=True, inplace=True)

    pca = PCA(n_components=3)  # reduce to k=3 dimensions
    scaler = StandardScaler()
    X = scaler.fit_transform(current_df)
    X_reduce = pca.fit_transform(X)
    fig = plt.figure(figsize=(11, 9))
    ax = fig.add_subplot(111, projection="3d")
    ax.set_zlabel("x_composite_3")
    ax.scatter(
        X_reduce[:, 0],
        X_reduce[:, 1],
        zs=X_reduce[:, 2],
        s=30,
        lw=1,
        label="inliers",
        c="green",
    )
    ax.scatter(
        X_reduce[Outliers[Outliers["model"] == "One-Class SVM"].index, 0],
        X_reduce[Outliers[Outliers["model"] == "One-Class SVM"].index, 1],
        X_reduce[Outliers[Outliers["model"] == "One-Class SVM"].index, 2],
        lw=4,
        s=40,
        c="red",
        label="One Class SVM",
    )
    ax.scatter(
        X_reduce[Outliers[Outliers["model"] == "Isolation Forest"].index, 0],
        X_reduce[Outliers[Outliers["model"] == "Isolation Forest"].index, 1],
        X_reduce[Outliers[Outliers["model"] == "Isolation Forest"].index, 2],
        lw=4,
        s=40,
        marker="x",
        c="blue",
        label="Isolation Forest",
    )
    ax.scatter(
        X_reduce[Outliers[Outliers["model"] == "Cluster-based LOF"].index, 0],
        X_reduce[Outliers[Outliers["model"] == "Cluster-based LOF"].index, 1],
        X_reduce[Outliers[Outliers["model"] == "Cluster-based LOF"].index, 2],
        lw=4,
        s=50,
        c="orange",
        label="Cluster-based LOF",
    )
    plt.title("Anomaly Detection", fontsize=20)
    plt.legend()
    plt.tight_layout()
```

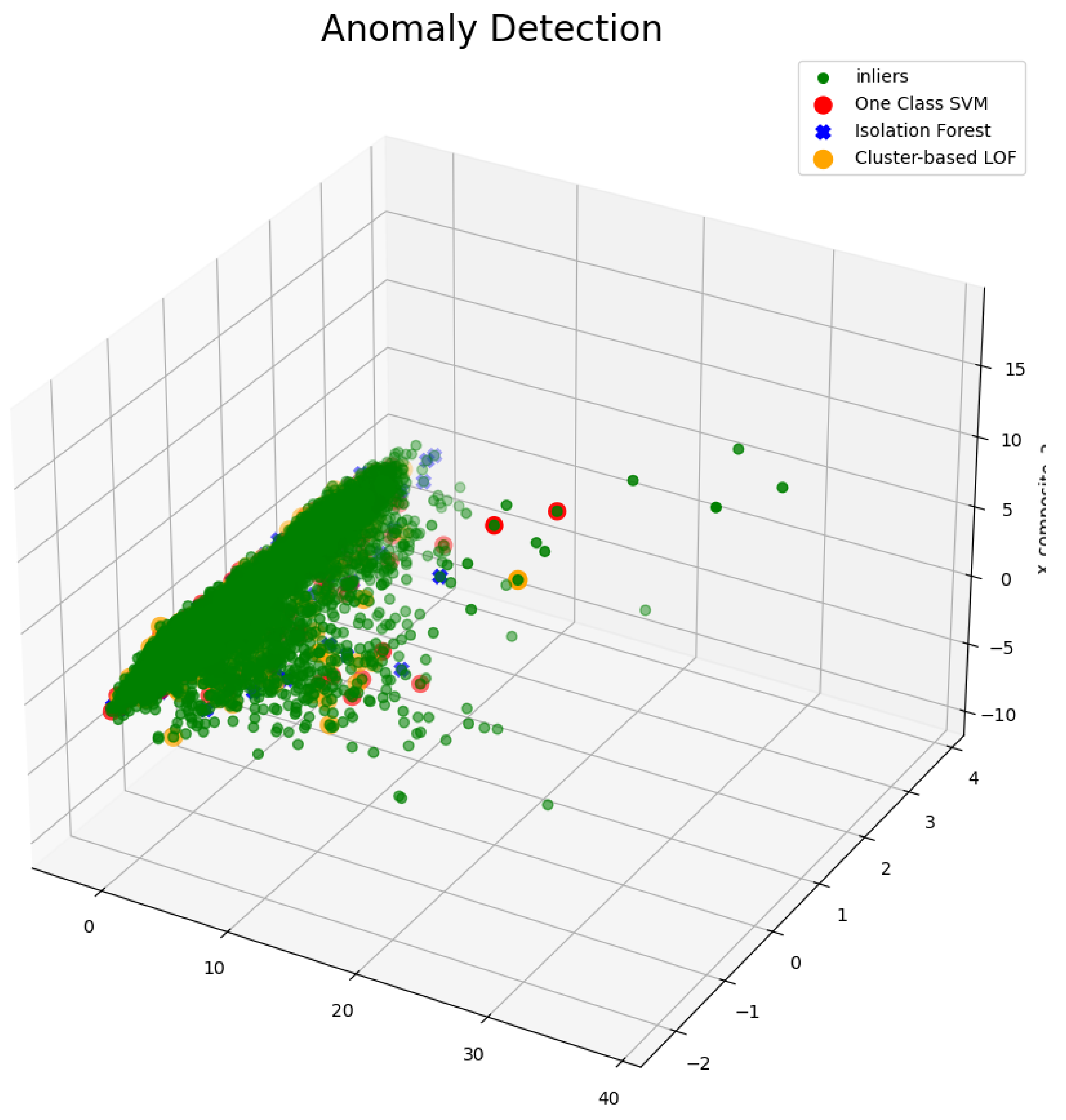
In [37]:
```
detectOutlier(danger_df.copy(True))
```



Interpretation:

- One Class SVM seems to be more successful in detect very obvious outliers which are far from most other points.
- Random Forest is able to find outliers even near the main cluster.

In [38]:
```
detectOutlier(nondanger_df.copy(True))
```



Interpretation:

- This dataset seems less clustered than the previous one as these are non-dangerous trips
- One Class SVM seems to be more successful in detect very obvious outliers which are far from most other points.
- Random Forest is able to find outliers even outside the main cluster.

Outlier Methods Comparison

To compare the effectivenes of the differnt outlier removal models, I will be trying out different combinations of models and fit them using a simple RandomForest model. I will then compare the score of the model to find the best outlier model.

To score the model, I will be using the following metric caled f_{beta} score which is use tot take the harmonic mean of precision and recal.

$$F_{\beta\text{-Score}} = \frac{(1+\beta^2)\cdot(precision \cdot recall)}{\beta^2 \cdot precision + recall}$$

This formula will ensure that the model takes into account both the recall and precision as well. For this project we will set $beta$ = 2 and use the f_2 score.

In [48]:
```
njobs = 8
algs_d = [('None', None),
        ('Isolation forest', IsolationForest(contamination=0.05, n_jobs=njobs)),
        ('Local outlier factor', LocalOutlierFactor(contamination=0.05)),
        ('One class SVM', OneClassSVM(nu=0.05))]

# copy paste, not 100% sure if even using .copy() will imply duplicating objects insi
algs_nd = [('None', None),
        ('Isolation forest', IsolationForest(contamination=0.05, n_jobs=njobs)),
        ('Local outlier factor', LocalOutlierFactor(contamination=0.05)),
        ('One class SVM', OneClassSVM(nu=0.05))]
```

In [50]:
```
y = df['label']
X = df.drop('label', axis=1)
outlier_df = pd.DataFrame(columns=['Method Dangerous', 'Method not dangerous','outlie
mlflow.set_experiment("Outlier Models Comparison")
for alg_d_name, alg_d in algs_d:
    for alg_nd_name, alg_nd in algs_nd:

        ls_accuracy = []
        ls_f2 =[]
        ls_recall = []
        outliers_removed = 0

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, ran
        with mlflow.start_run():

            if alg_d is not None and algs_nd is not None:

                X_train_d = X_train.loc[y_train == 1]
                y_train_d = y_train[y_train == 1]
                X_train_nd = X_train.loc[y_train == 0]
                y_train_nd = y_train[y_train == 0]

                # if trying to
                if alg_d is not None:

                    X_out_d = alg_d.fit_predict(X_train_d)
                    mask = (X_out_d != -1)

                    outliers_removed += np.sum(mask == False)

                    X_train_d, y_train_d = X_train_d.iloc[mask, :], y_train_d.iloc[ma

                if alg_nd is not None:

                    X_out_nd = alg_nd.fit_predict(X_train_nd)
                    mask = (X_out_nd != -1)

                    outliers_removed += np.sum(mask == False)

                    X_train_nd, y_train_nd = X_train_nd.iloc[mask, :], y_train_nd.ilo

                X_train = pd.concat([X_train_d, X_train_nd])
                y_train = pd.concat([y_train_d, y_train_nd])

                # perform a shuffle on both X_train and y_train
                idx = np.random.permutation(X_train.index)
                X_train = X_train.reindex(idx)
                y_train = y_train.reindex(idx)
            #print(f'Shape of the training dataset after removing outliers with {alg_

            clf = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=nj
            clf.fit(X_train, y_train)
            y_pred = clf.predict(X_test)

            accuracy = accuracy_score(y_test, y_pred)
            recall = recall_score(y_test, y_pred)
            f2 = fbeta_score(y_test, y_pred, beta=2)

            ls_accuracy.append(accuracy)
            ls_f2.append(f2)
            ls_recall.append(recall)

            avg_accuracy = sum(ls_accuracy)/ len(ls_accuracy)
            avg_recall = sum(ls_recall)/ len(ls_recall)
            avg_f2 = sum(ls_f2)/ len(ls_f2)


            mlflow.log_metric("Accuracy", accuracy)
            mlflow.log_metric("Recall", recall)
            mlflow.log_metric("F2 Score", f2)
            mlflow.sklearn.log_model(clf, "Random Forest")
            mlflow.set_tag('mlflow.runName', f"{alg_d_name}+{alg_nd_name}")

            row_to_append = {'Method Dangerous': alg_d_name,
                            'Method not dangerous' : alg_nd_name,
                            'outliers_removed' : outliers_removed,
                            'Accuracy': avg_accuracy,
                            'Recall': avg_recall,
                            'F2_Score' : avg_f2}
            outlier_df = pd.concat([outlier_df, pd.DataFrame([row_to_append])], ignor

outlier_df
```

```
2024/02/08 00:19:43 INFO mlflow.tracking.fluent: Experiment with name 'Outlier Models
Comparison' does not exist. Creating a new experiment.
```

Out[50]:

	Method Dangerous	Method not dangerous	outliers_removed	Accuracy	Recall	F2_Score
0	None	None	0	0.784545	0.203202	0.237274
1	None	Isolation forest	0	0.784545	0.203202	0.237274
2	None	Local outlier factor	0	0.784545	0.203202	0.237274
3	None	One class SVM	0	0.784545	0.203202	0.237274
4	Isolation forest	None	169	0.782424	0.187808	0.220439
5	Isolation forest	Isolation forest	671	0.774848	0.245074	0.278166
6	Isolation forest	Local outlier factor	671	0.773939	0.225985	0.258524
7	Isolation forest	One class SVM	671	0.784242	0.217980	0.252713
8	Local outlier factor	None	169	0.783485	0.189655	0.222640
9	Local outlier factor	Isolation forest	671	0.772576	0.236453	0.268870
10	Local outlier factor	Local outlier factor	671	0.777727	0.222906	0.256265
11	Local outlier factor	One class SVM	671	0.783939	0.211823	0.246207
12	One class SVM	None	168	0.783636	0.193350	0.226617
13	One class SVM	Isolation forest	670	0.773333	0.240764	0.273427
14	One class SVM	Local outlier factor	670	0.773636	0.234606	0.267256
15	One class SVM	One class SVM	670	0.784242	0.225985	0.261024

Observations

Method Dangerous	Method not dangerous	Outliers Removed	Accuracy	Recall	F2 Score
None	None	0	0.7845	0.2032	0.2373
None	Isolation Forest	0	0.7845	0.2032	0.2373
None	Local Outlier Factor	0	0.7845	0.2032	0.2373
None	One-Class SVM	0	0.7845	0.2032	0.2373
Isolation Forest	None	169	0.7809	0.1786	0.2102
Isolation Forest	Isolation Forest	671	0.7721	0.2549	0.2873
Isolation Forest	Local Outlier Factor	671	0.7724	0.2284	0.2607
Isolation Forest	One-Class SVM	671	0.7833	0.2180	0.2525
Local Outlier Factor	None	169	0.7835	0.1934	0.2266
Local Outlier Factor	Isolation Forest	671	0.7748	0.2297	0.2625
Local Outlier Factor	Local Outlier Factor	671	0.7758	0.2180	0.2507
Local Outlier Factor	One-Class SVM	671	0.7842	0.2094	0.2437
One-Class SVM	None	168	0.7803	0.1817	0.2134
One-Class SVM	Isolation Forest	670	0.7736	0.2445	0.2772
One-Class SVM	Local Outlier Factor	670	0.7742	0.2309	0.2636
One-Class SVM	One-Class SVM	670	0.7838	0.2198	0.2545

- Isolation Forest and Local Outlier Factor achieve relatively higher accuracies compared to One-Class SVM in most scenarios.
- When used alone, Isolation Forest tends to detect more outliers compared to the other methods.
- Combining Isolation Forest with itself or with other methods generally improves recall and F2 score, but may slightly reduce accuracy.
- One-Class SVM performs well in certain scenarios, particularly when combined with Isolation Forest or Local Outlier Factor.

Final Outlier Removal

Based on the provided results, Isolation Forest for both datasets seem to be best method for detecting anomalies. We will be applying it to the dataset which is freshly loaded and saved it once more.

In [47]:
```python
 # import the data again. This time the data will not be transformed, so used the tra
df_pure = pd.read_hdf("./h5/cleaned_dataset.h5", index_col = 0)

danger_df_pure = df_pure.loc[df_pure['label'] == 1]
nondanger_df_pure = df_pure.loc[df_pure['label'] == 0]

outlier_alg_d = IsolationForest(contamination=0.05, n_jobs=njobs)
outlier_alg_nd = IsolationForest(contamination=0.05, n_jobs=njobs)

# separate the features from the labels first
X_d = danger_df.drop('label', axis=1)
y_d = danger_df['label']
X_nd = nondanger_df.drop('label', axis=1)
y_nd = nondanger_df['label']

# fit and predict
X_out_d = outlier_alg_d.fit_predict(X_d)
X_out_nd = outlier_alg_nd.fit_predict(X_nd)

# create the masks, which will be used to filter out outliers
mask_d = (X_out_d != -1)
mask_nd = (X_out_nd != -1)

# count the number removed
outliers_removed = np.sum(mask_d == False) + np.sum(mask_nd == False)
print(f'Number of outliers removed: {outliers_removed}')

# do the filtering
danger_df_pure = danger_df_pure.iloc[mask_d, :]
nondanger_df_pure = nondanger_df_pure.iloc[mask_nd, :]

# recombine the danger and non danger
df_final = pd.concat([danger_df_pure, nondanger_df_pure])

# perform a shuffle
idx = np.random.permutation(df_final.index)
df_final = df_final.reindex(idx)

# save it
df_final.to_hdf('./h5/very_cleaned_dataset.h5', 'w')
```

```
Number of outliers removed: 1001
```

The number of outliers removed is higher here as the train test split wasn't applied and the model is fitting on the whole dataset.

Exploratory Data Analysis (Aggregated Data)

Before passing the data into a model, we will perform some basic EDA to find patterns in the data and create the preprocessing pipeline accordingly.

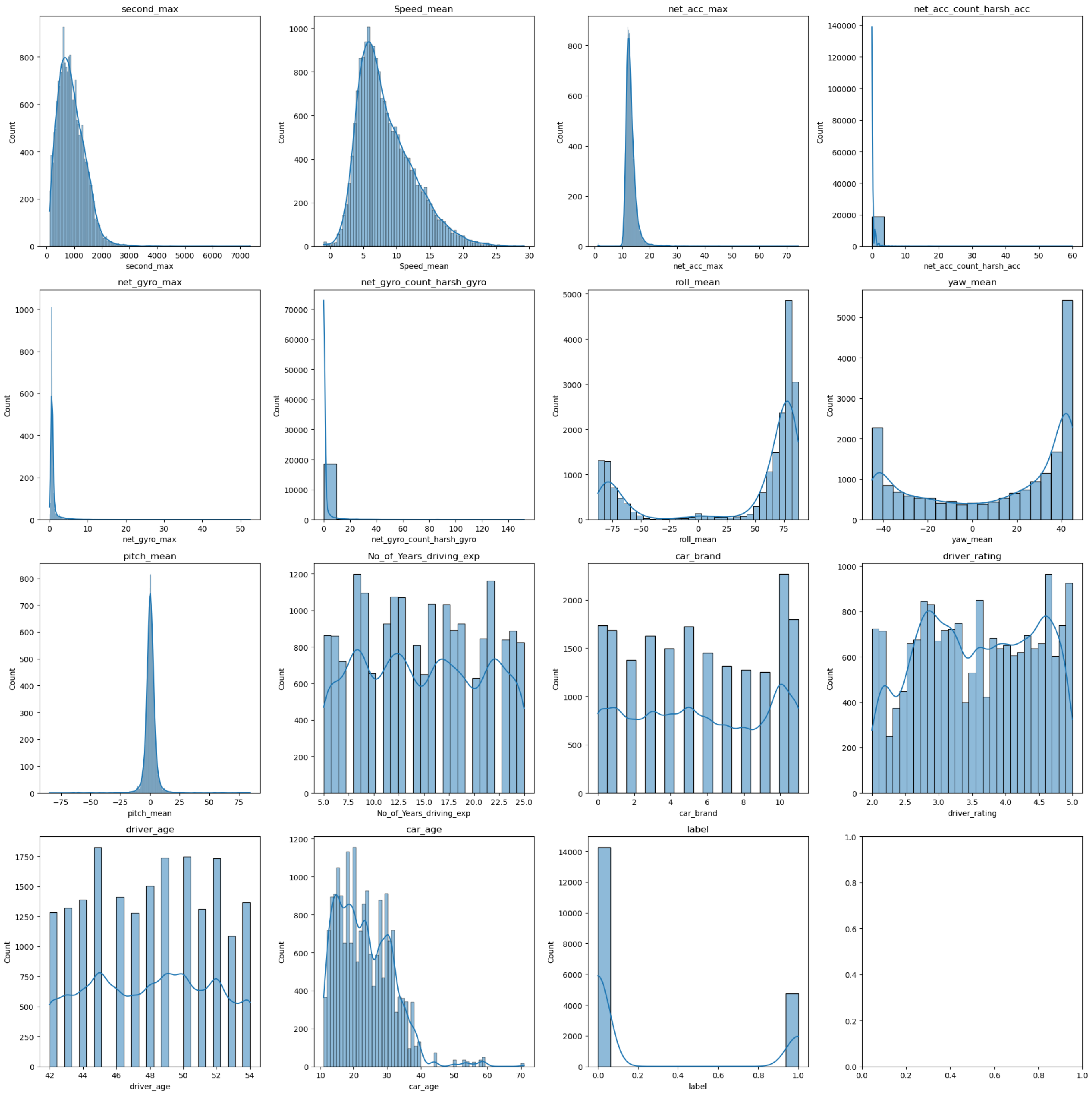
In [87]:
```
df = pd.read_hdf("./h5/very_cleaned_dataset.h5", index_col = 0)
print(df.shape)
df.head()
```

```
(18997, 16)
```

Out[87]:

	second_max	Speed_mean	net_acc_max	net_acc_count_harsh_acc	net_gyro_max	net_
8036	434.0	7.085810	13.978516	0	0.462158	
19521	1142.0	12.224400	14.094879	0	0.727868	
7811	643.0	6.596449	11.897546	0	0.496676	
5113	326.0	8.818926	11.400488	0	0.481220	
2051	701.0	11.456132	11.778945	0	0.282603	

In [89]:
```
pd.option_context('mode.use_inf_as_na', True)

numeric_columns = df.select_dtypes(include=['float64', 'float32', 'int64', 'int8'])
fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(20, 20))
axes = axes.flatten()
for i, col in enumerate(numeric_columns.columns):
    sns.histplot(data=df, x=col, kde=True, ax=axes[i])
    axes[i].set_title(col)

plt.tight_layout()
plt.show()
```



Observations

- Second_max, Speed_mean, Car_age, net_gyro_max and net_acc_count_harsh see to have a right-skewed distribution to fix this, we will have to explore box-cox or log transformations.
- Remaining disibutions seem fine, they just need scaling.

Box-Cox vs Log Transformations

To fix the distirbution of our numeric coumns and make them more normal, we will be using a transformation.

Box-Cox Transformation

- **Definition**: The Box-Cox transformation is a family of power transformations that are indexed by a parameter, lambda (λ). It is defined as:

$$y^{(\lambda)} = \begin{cases} \frac{y^{\lambda}-1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(y) & \text{if } \lambda = 0 \end{cases}$$

- **Usage**: Box-Cox transformation is used when data exhibit heteroscedasticity (varying variance) and skewness.
- **Parameter Estimation**: The optimal value of λ can be estimated using maximum likelihood estimation or other optimization techniques.
- **Restrictions**: Box-Cox transformation requires that the data be strictly positive (greater than zero).

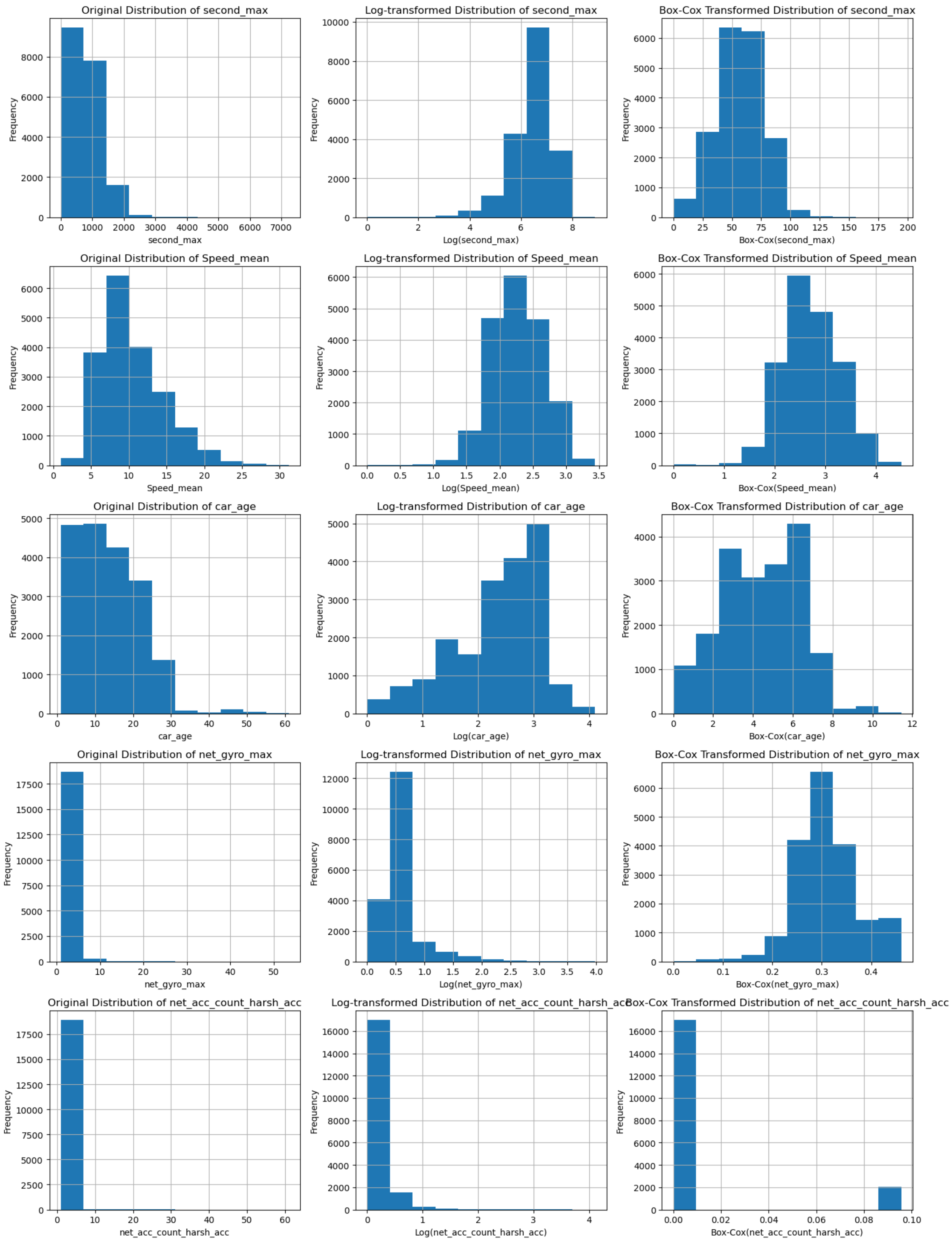
Log Transformation

- **Definition**: The logarithm transformation involves taking the logarithm (usually natural logarithm) of each data point. It is defined as:

$$y_{\log} = \log(y)$$

- **Usage**: Log transformation is used when data exhibit exponential growth or decay and when the variance is proportional to the mean.
- **Parameter Estimation**: There are no parameters to estimate; log transformation is a simple mathematical operation.
- **Restrictions**: Log transformation can only be applied to strictly positive data.

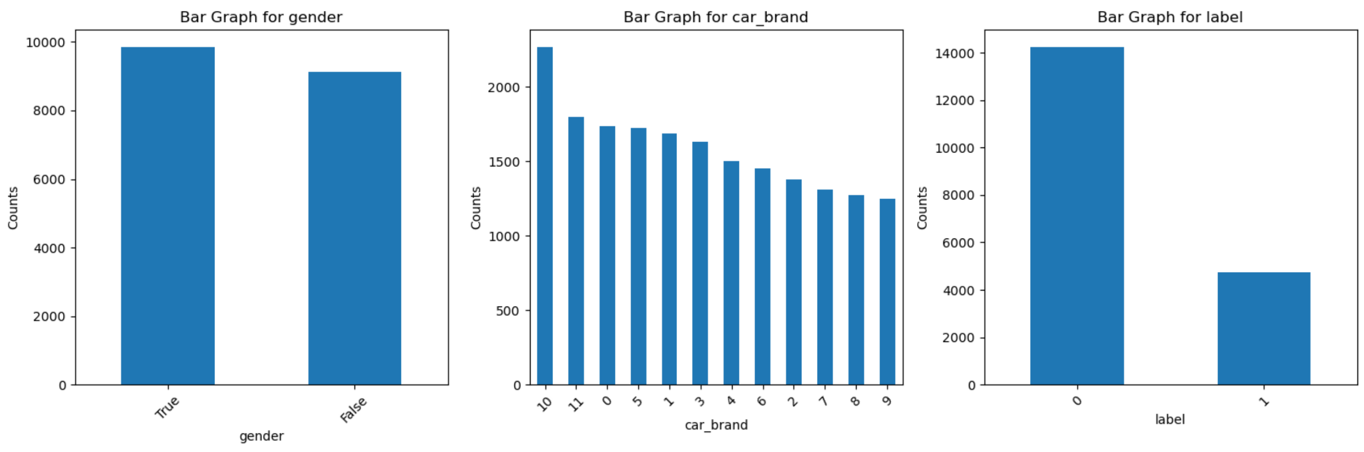
In [90]:
```
cols = ['second_max', 'Speed_mean', 'car_age', 'net_gyro_max', 'net_acc_count_harsh_a

# Create subplots
fig, axes = plt.subplots(nrows=len(cols), ncols=3, figsize=(15, 20))

# Plot original, log-transformed, and Box-Cox transformed distributions
for i, col in enumerate(cols):
    # Original distribution
    shifted_data= df[col] - np.min(df[col]) + 1
    shifted_data.hist(ax=axes[i, 0], bins=10)
    axes[i, 0].set_title(f'Original Distribution of {col}')
    axes[i, 0].set_xlabel(col)
    axes[i, 0].set_ylabel('Frequency')

    # Log-transformed distribution
    log_transformed = np.log(shifted_data)
    log_transformed.hist(ax=axes[i, 1], bins=10)
    axes[i, 1].set_title(f'Log-transformed Distribution of {col}')
    axes[i, 1].set_xlabel('Log(' + col + ')')
    axes[i, 1].set_ylabel('Frequency')

    # Box-Cox transformed distribution (if applicable)
    if (shifted_data > 0).all():  # Box-Cox transformation requires strictly positive
        boxcox_transformed, _ = boxcox(shifted_data)
        pd.Series(boxcox_transformed).hist(ax=axes[i, 2], bins=10)
        axes[i, 2].set_title(f'Box-Cox Transformed Distribution of {col}')
        axes[i, 2].set_xlabel('Box-Cox(' + col + ')')
        axes[i, 2].set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



As can be observed from the graphs, box-cox is more successful in acheiving a normal distribution that log transformation. We will be using that.

We will also take a look at the categorical columns.

In [91]:
```
cols = ['gender', 'car_brand', 'label']

fig, axes = plt.subplots(nrows=1, ncols=len(cols), figsize=(15, 5))

for i, col in enumerate(cols):
    counts = df[col].value_counts()
    counts.plot(kind='bar', ax=axes[i])
    axes[i].set_title(f'Bar Graph for {col}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Counts')
    axes[i].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()
```



Finding Highly Correlated Columns

Correlation analysis helps to identify these relationships, providing insights into how variables are related to each other. When variables are highly correlated in a classification problem, several issues can arise:

1. **Multicollinearity**: High correlation between predictor variables can destabilize the estimation of coefficients in linear models, making them unreliable.

2. **Overfitting**: Highly correlated features may lead to overfitting resulting in poor performance on test data.

3. **Inflated Importance of Correlated Features**: In models that rely on feature importance measures, highly correlated features may appear more important than they actually are, skewing the interpretation of feature importance.

To mitigate these issues, we will try to remove highly correlated features. We will be using a few metrics to do so.

Correlation Coefficient

The correlation coefficient is a statistical measure that quantifies the strength and direction of a relationship between two variables. It ranges from -1 to 1, where:

- 1 indicates a perfect positive correlation,
- -1 indicates a perfect negative correlation, and
- 0 indicates no correlation.

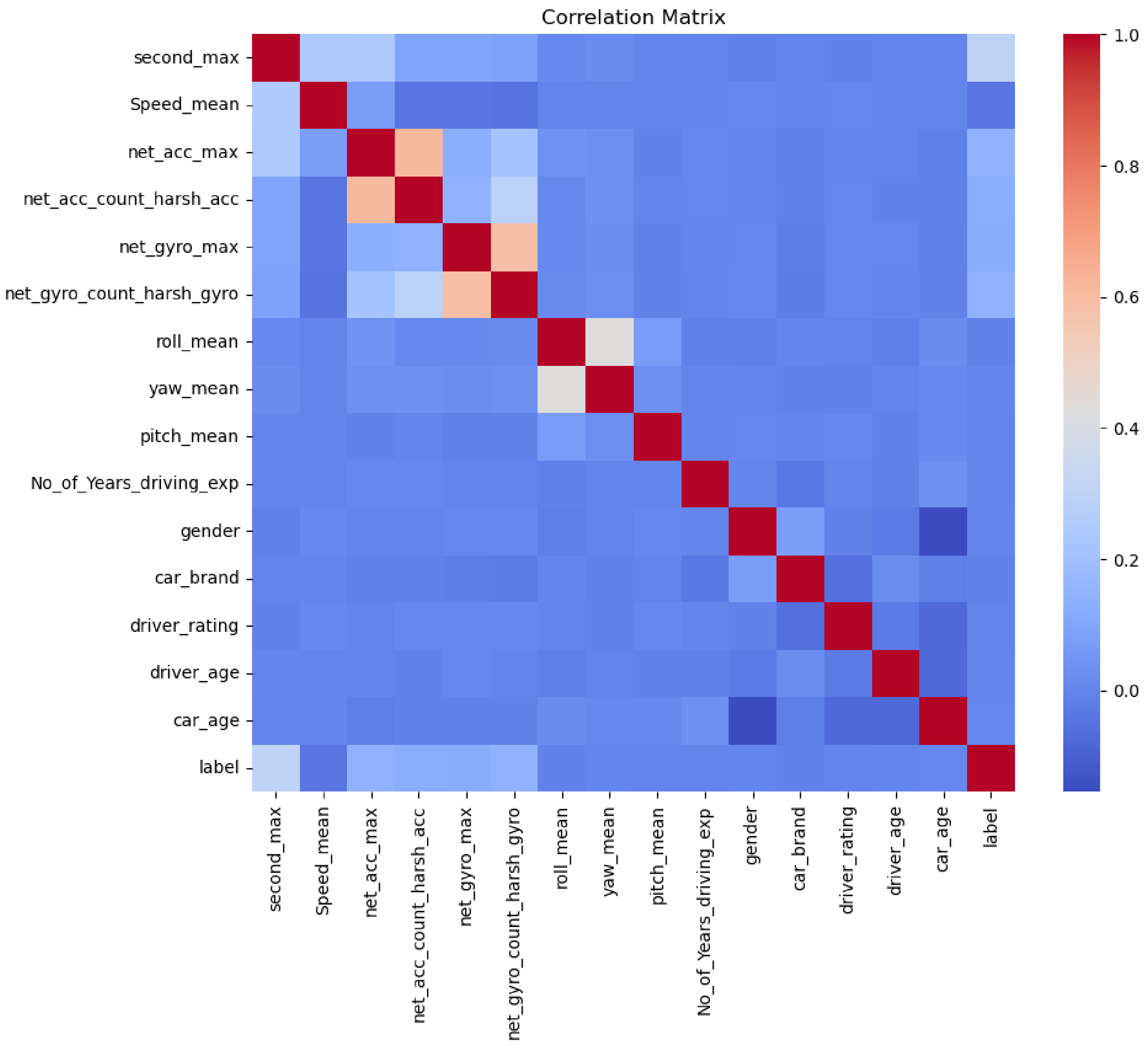
Methods for Calculating Correlation

1. **Pearson Correlation Coefficient**: Measures the linear relationship between two continuous variables.
2. **Spearman Correlation Coefficient**: Measures the strength and direction of association between two ranked variables.

We will be using Pandas `.corr()` function to calculate correlation.

In [102…
```
one_hot_encoder = OneHotEncoder(sparse=False, drop="first")
df["gender"] = df["gender"].replace({True: 1, False: 0})
one_hot_car = one_hot_encoder.fit_transform(df["car_brand"].to_numpy().reshape(-1, 1)
encoded_columns = one_hot_encoder.get_feature_names_out(["car_brand"])
encoded_df = pd.DataFrame(one_hot_car, columns=encoded_columns)
new_df = pd.concat([df.reset_index(drop=True), encoded_df.reset_index(drop=True)], ax
```

In [104…
```
correlation_matrix = df.corr()



plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.show()

high_correlation_pairs = correlation_matrix.unstack()
high_correlation_pairs = high_correlation_pairs[
    (high_correlation_pairs.between(0.19, 1.0, inclusive="left")) |
    (high_correlation_pairs.between(-1.0, -0.19, inclusive="left"))
]
high_correlation_pairs = high_correlation_pairs[
    high_correlation_pairs.index.get_level_values(0) != high_correlation_pairs.index.
]

highest_10 = high_correlation_pairs.nlargest(10)

highest_10
```



```
Out[104…  net_acc_max                net_acc_count_harsh_acc      0.612009
          net_acc_count_harsh_acc    net_acc_max                  0.612009
          net_gyro_max               net_gyro_count_harsh_gyro    0.587225
          net_gyro_count_harsh_gyro  net_gyro_max                 0.587225
          roll_mean                  yaw_mean                     0.428897
          yaw_mean                   roll_mean                    0.428897
          second_max                 label                        0.304729
          label                      second_max                   0.304729
          net_acc_count_harsh_acc    net_gyro_count_harsh_gyro    0.288366
          net_gyro_count_harsh_gyro  net_acc_count_harsh_acc      0.288366
          dtype: float64
```

Since the features net_acc_count_harsh_acc and net_gyro_count_harsh_gyro are highly correlated. These are also columns we created during feature engineering. Since these add uncessary noise to the dataset, we will remove them. For Yaw and Roll we will be combining them using the Pythagoras theorem to capture the information in a more concise way.

```
In [105…  filtered_df = new_df.drop(columns=['net_acc_count_harsh_acc','net_gyro_count_harsh_gy
```

Dimensionality Reduction with PCA

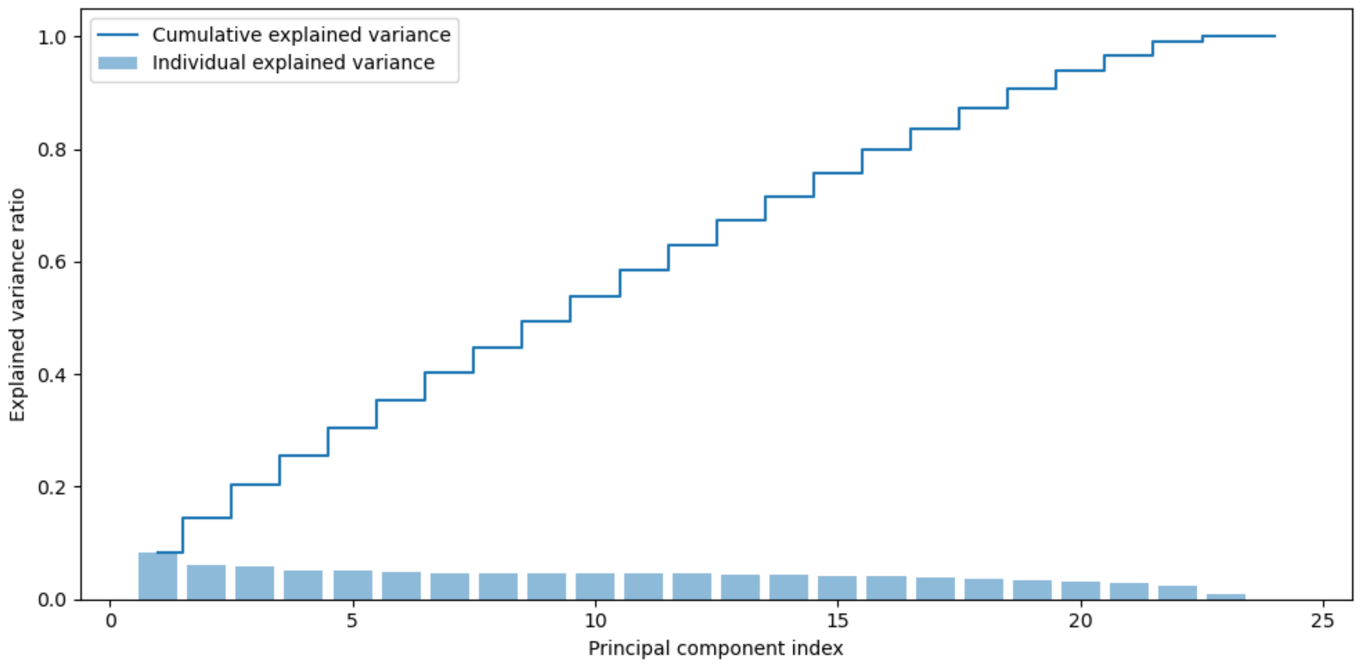
In addition to addressing multicollinearity, we will employ Principal Component Analysis (PCA) to reduce the dimensionality of our dataset. PCA is a powerful technique that transforms high-dimensional data into a lower-dimensional space while retaining most of the variance. By reducing the number of features, PCA simplifies the model and helps prevent overfitting, leading to improved generalization performance.

In [106…
```
features = filtered_df.drop('label', axis=1)
target = filtered_df['label']

scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

features = pd.DataFrame(features_scaled, columns=features.columns)
features_mean_std = features.aggregate(['mean', 'std']).T

features_mean_std.describe()
```

Out[106…

	mean	std
count	2.400000e+01	2.400000e+01
mean	2.592098e-17	1.000026e+00
std	1.685189e-16	5.968471e-14
min	-2.517215e-16	1.000026e+00
25%	-2.683657e-17	1.000026e+00
50%	-3.225999e-18	1.000026e+00
75%	2.029107e-17	1.000026e+00
max	6.822287e-16	1.000026e+00

In [108…
```
pca = PCA()
pca.fit(features_scaled)

explained_variance = pca.explained_variance_ratio_

# Calculate the cumulative explained variance
cumulative_explained_variance = pca.explained_variance_ratio_.cumsum()

# Create a scree plot
plt.figure(figsize=(10, 5))
plt.bar(range(1, len(explained_variance) + 1), explained_variance, alpha=0.5, align='
plt.step(range(1, len(cumulative_explained_variance) + 1), cumulative_explained_varia
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.tight_layout()
plt.show()

cumulative_explained_variance
```



```
Out[108…  array([0.08357821, 0.14502554, 0.20348976, 0.25509315, 0.30550179,
                 0.35443549, 0.40168869, 0.44834466, 0.49392652, 0.53934384,
                 0.58459976, 0.62961523, 0.67380965, 0.71655534, 0.75783039,
                 0.79867019, 0.83682329, 0.87340463, 0.90763013, 0.93863826,
                 0.96703366, 0.99067306, 1.        , 1.        ])
```

PCA May Not Be Very Useful Here

PCA as seen on the graph doesn't seem very successfuly in reducing complexity due to the following factors:

1. High Retained Variance

In the provided array, the variance explained by the principal components increases gradually, with the first few components capturing a significant portion of the total variance.

2. No Clear Elbow Point

In an ideal scenario, the explained variance ratio plot would exhibit a clear elbow point, indicating the number of principal components to retain. However, here there isn't any point where the rate of increase in explained variance significantly slows down.

3. Minimal Reduction in Dimensionality

The array suggests that even with a relatively small number of principal components, a high percentage of the variance is retained. This implies that reducing the dimensionality of the data using PCA may not lead to a significant reduction in information loss.

Therefore, we will not be using PCA for our project.

Modelling

Based on the Exploratory Data Analysis (EDA) conducted, we have identified a set of transformations and techniques to be applied in our modelling approach:

Pipeline:

1. **One-Hot Encoding for Car Brand:**

  - Since car brand is a categorical variable with no order, one-hot encoding will be applied to represent each category as a binary feature.
2. **Ordinal Encoding for Gender:**
  - Gender is another categorical variable,ordinal encoding will be applied to convert these categories into numerical values (0 and 1) while preserving the ordinal relationship.
3. **Combining Yaw and Roll with Pythagorean Theorem:**
  - Yaw and mean represent two numerical features in the dataset. By combining them using the Pythagorean theorem, we can create a new feature that captures their relationship more effectively, potentially improving the model's predictive power.
4. **Box-Cox Transformation:**
  - Box-Cox transformation is a technique used to stabilize the variance and make the data more Gaussian-like. This transformation will be applied to certain numerical features in the dataset to address issues such as skewness and heteroscedasticity.
5. **Standard Scaler:**
  - Standardization, or scaling, is a crucial preprocessing step to ensure that all features have a mean of 0 and a standard deviation of 1.
6. **SMOTE (Synthetic Minority Over-sampling Technique):**
  - SMOTE is a technique used to address class imbalance by oversampling the minority class. This helps improve the model's performance, especially when dealing with imbalanced datasets where one class is significantly underrepresented compared to others.

All these steps, barring the combination of Yaw and Mean will be done in the SkLearn pipeline.

We will be using a 80-10-10 Train-Test-Val-Split

In [4]:
```
df = pd.read_hdf('./h5/very_cleaned_dataset.h5')
df["gender"] = df["gender"].replace({True:1,False:0})
y = df['label']
X = df.drop(columns=['label','roll_mean', 'yaw_mean','net_acc_count_harsh_acc','net_g

X['roll_yaw_mean'] = (df['roll_mean']**2 + df['yaw_mean'] **2)**0.5
numeric_cols = list(set(X.columns.to_list()).difference({'car_brand', 'gender'}))
```

In [86]:
```
# Assuming df is your DataFrame containing the relevant columns
columns_to_check = ['second_max', 'Speed_mean', 'car_age', 'net_gyro_max']

for col_name in columns_to_check:
    col_values = df[col_name]
    col_min = col_values.min()
    col_max = col_values.max()
    print(f"Range of column '{col_name}': [{col_min}, {col_max}]")
```

```
Range of column 'second_max': [114.0, 7354.0]
Range of column 'Speed_mean': [-1.0, 29.213970778515396]
Range of column 'car_age': [11, 71]
Range of column 'net_gyro_max': [0.008876276590069967, 52.49278526731891]
```

In [87]:
```
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler, Powe

col_transformer = ColumnTransformer([
                        ('OHE', OneHotEncoder(), ['car_brand']),
                        ('power_transform', PowerTransformer(method='yeo-johnson'), ['s
                        ('scaler', StandardScaler(), numeric_cols)
                        ])
X = col_transformer.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.2, random_state = 42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_test, y_test, test_size = 0.5, random_state = 42
)
smote = SMOTE(random_state=42)
X_train, y_train = smote.fit_resample(X_train, y_train)
```

In [10]:
```
njobs = 12

models = [
    ("AdaBoost", AdaBoostClassifier(n_estimators=50, learning_rate=1.0)),
    ("DecisionTreeClassifier", DecisionTreeClassifier(max_depth=None, min_samples_spl
    ("ExtraTreesClassifier", ExtraTreesClassifier(n_estimators=100, max_depth=None, m
    ("GradientBoostingClassifier", GradientBoostingClassifier(n_estimators=100, learn
    ("HistGradientBoostingClassifier", HistGradientBoostingClassifier(max_iter=100)),
    ("KNeighborsClassifier", KNeighborsClassifier(n_neighbors=5, n_jobs=njobs)),
    ("LogisticRegression", LogisticRegression(max_iter=1000, n_jobs=njobs)),
    ("NaiveBayes", GaussianNB()),
    ("Perceptron", Perceptron(max_iter=1000, n_jobs=njobs)),
    ("RandomForest", RandomForestClassifier(n_estimators=100, max_depth=None, min_sam
    ("Ridge", RidgeClassifier()),
    ("SVC", SVC())
]
```

In [8]:
```
def test_all_models(X_train, X_val, y_train, y_val):

    model_results = pd.DataFrame(columns=['Model name', 'Train Accuracy', 'Train Reca
                                          'Validation Accuracy', 'Validation Recall
                                          'Fitting time (s)' , 'Predicting time (s)

    for name, model in models:
        # fit the model and also time how long it takes to do so
        with mlflow.start_run():
            start_time = time()
            model.fit(X_train, y_train)
            dt_fit = round(time() - start_time, 4)

            # Predict on both training and validation sets
            y_train_pred = model.predict(X_train)

            # and time how long it takes to predict on validation data
            start_time = time()
            y_val_pred = model.predict(X_val)
            dt_pred = round(time() - start_time, 4)

            # Calculate metrics for the training set
            train_accuracy = accuracy_score(y_train, y_train_pred)
            train_recall = recall_score(y_train, y_train_pred, average='macro')
            train_f2 = fbeta_score(y_train, y_train_pred, beta=2, average='macro')

            # Calculate metrics for the validation set
            val_accuracy = accuracy_score(y_val, y_val_pred)
            val_recall = recall_score(y_val, y_val_pred)
            val_f2 = fbeta_score(y_val, y_val_pred, beta=2)

            mlflow.log_metric("Train Accuracy", train_accuracy)
            mlflow.log_metric("Train Recall", train_recall)
            mlflow.log_metric("Train F2 Score", train_f2)
            mlflow.log_metric("Validation Accuracy", val_accuracy)
            mlflow.log_metric("Validation Recall", val_recall)
            mlflow.log_metric("Validation F2 Score", val_f2)
            mlflow.log_metric("Fitting time ", dt_fit)
            mlflow.log_metric("Predicting time", dt_pred)
            mlflow.sklearn.log_model(model, name)
            mlflow.set_tag('mlflow.runName', name)

            result = pd.DataFrame([{'Model name' : name,
                        'Train Accuracy': train_accuracy,
                        'Train Recall': train_recall,
                        'Train F2 Score': train_f2,
                        'Validation Accuracy': val_accuracy,
                        'Validation Recall': val_recall,
                        'Validation F2 Score': val_f2,
                        'Fitting time (s)' : dt_fit,
                        'Predicting time (s)' : dt_pred}])

            model_results = pd.concat([model_results, result], ignore_index=True)

    return model_results
```

In [89]:
```
mlflow.set_experiment("Models Comparison")
test_all_models(X_train, X_val, y_train, y_val)
```

Out[89]:

	Model name	Train Accuracy	Train Recall	Train F2 Score	Validation Accuracy	Validation Recall	Validation F2 Score
0	AdaBoost	0.696246	0.696246	0.696129	0.666316	0.573840	0.523077
1	DecisionTreeClassifier	1.000000	1.000000	1.000000	0.624737	0.396624	0.374353
2	ExtraTreesClassifier	1.000000	1.000000	1.000000	0.730526	0.276371	0.298270
3	GradientBoostingClassifier	0.747978	0.747978	0.747553	0.703684	0.516878	0.494949
4	HistGradientBoostingClassifier	0.845596	0.845596	0.844833	0.732105	0.411392	0.420078
5	KNeighborsClassifier	0.843354	0.843354	0.839388	0.594211	0.561181	0.488073
6	LogisticRegression	0.647679	0.647679	0.647535	0.660000	0.632911	0.562219
7	NaiveBayes	0.610364	0.610364	0.595188	0.737895	0.413502	0.423875
8	Perceptron	0.609265	0.609265	0.608568	0.630526	0.599156	0.527489
9	RandomForest	1.000000	1.000000	1.000000	0.731579	0.390295	0.401825
10	Ridge	0.648163	0.648163	0.648123	0.652105	0.643460	0.566283
11	SVC	0.719541	0.719541	0.719405	0.651053	0.618143	0.548484

Model Evaluation Results

- **Training Performance**:
  - Several models achieve near-perfect accuracy, recall, and F2 score on the training data, such as Decision Tree, Extra Trees, and Random Forest. This suggests potential overfitting.
  - Models like Logistic Regression and Perceptron show relatively lower but still reasonable performance on the training data.
- **Test Performance**:
  - The performance on the test data varies across models, with some exhibiting a drop in performance compared to the training set, indicating overfitting.
  - Models like AdaBoost, Gradient Boosting, and LightGBM demonstrate decent performance on the test set, with accuracy and F2 score above 0.7.
  - However, other models like Decision Tree, Naive Bayes, and KNeighborsClassifier show significant drops in performance on the test data compared to the training data.

- **Fitting and Predicting Time**:
    - SVC (Support Vector Classifier) and Gradient Boosting Classifier exhibit the highest fitting time, indicating longer training times.
    - Logistic Regression, Perceptron, and Ridge Regression have relatively shorter fitting times.
    - Predicting time varies across models, with some models like SVC showing significantly higher prediction times.

Conclusion:

Considering the trade-off between performance and computational efficiency, **Ridge Regression** emerges as a suitable choice:

- It achieves a respectable accuracy, recall, and F2 score on the test data, indicating good generalization.
- The fitting and predicting times are relatively low compared to other models like SVC and Gradient Boosting.
- Ridge Regression also avoids the issue of overfitting seen in some models like Decision Tree and Random Forest.

Therefore, based on these observations, employing Ridge Regression seems prudent for this classification task.

Feature Importance

Next, check for feature importance on some selected models.

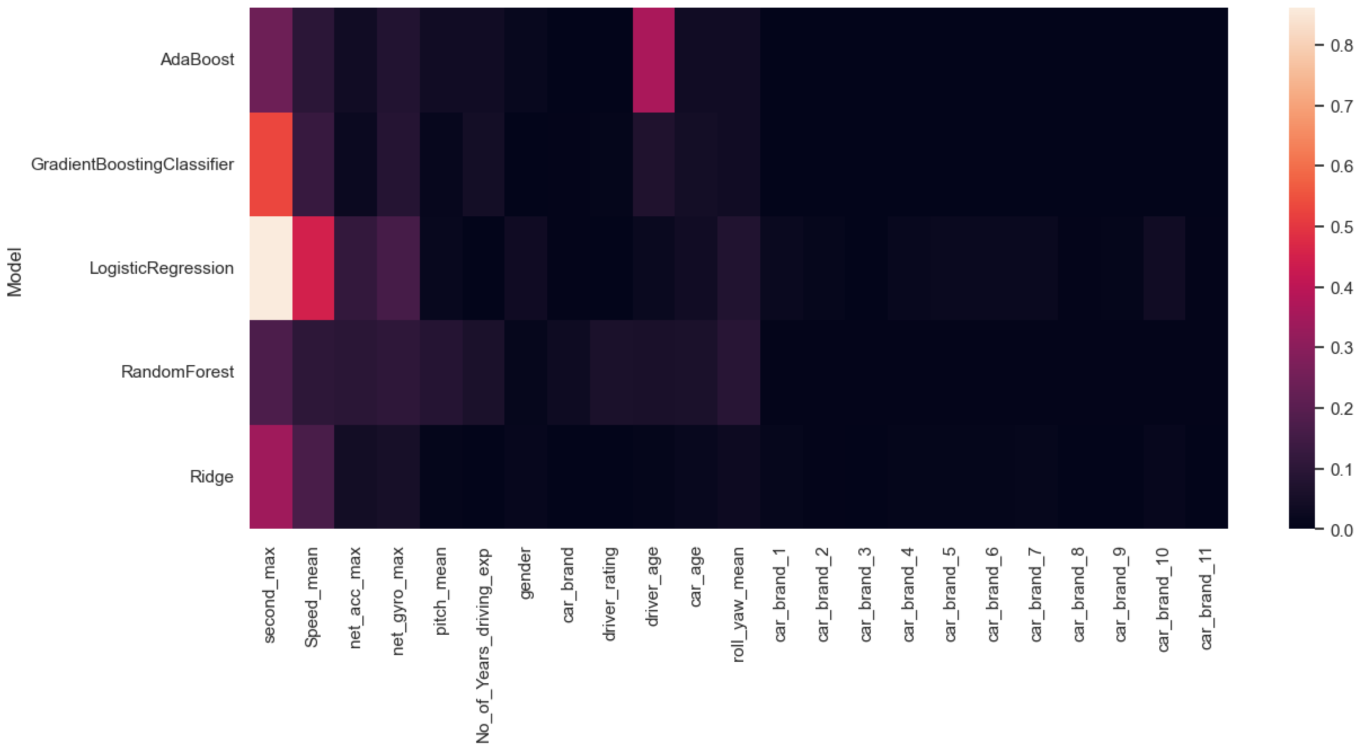
The way to access the coefficient is not standardised, so not feasible to put in for loop

Also, we need to manually encode, scale, and transform the data because by using the column transformer, we essentially got rid of the feature names, so it defeats the purpose of doing feature importance if we do not know which feature is important, and even if we try to remake the dataframe, the number of columns changed after the column transform, so there will be shape mismatch, and thus we will have to break down the transform into bits by bits.

In [99]:
```
ToOHEncode = ['car_brand']
toOrdinalEncode = ['gender']

one_hot_encoder = OneHotEncoder(sparse=False, drop="first")
df["gender"] = df["gender"].replace({True: 1, False: 0})
one_hot_car = one_hot_encoder.fit_transform(df["car_brand"].to_numpy().reshape(-1, 1)
encoded_columns = one_hot_encoder.get_feature_names_out(["car_brand"])
encoded_df = pd.DataFrame(one_hot_car, columns=encoded_columns)
new_df = pd.concat([df.reset_index(drop=True), encoded_df.reset_index(drop=True)], ax

# combining the yaw and roll should've been done earlier, so no need do it again
y = new_df['label']
new_df['roll_yaw_mean'] = (new_df['roll_mean']**2 + new_df['yaw_mean'] **2)**0.5
X = new_df.drop(columns=['label','roll_mean', 'yaw_mean','net_acc_count_harsh_acc','n
```

In [100…
```
# scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# we need a transformed X but in dataframe, so we can do feature importance and get w
X = pd.DataFrame(X_scaled, index=new_df.index, columns=X.columns)
```

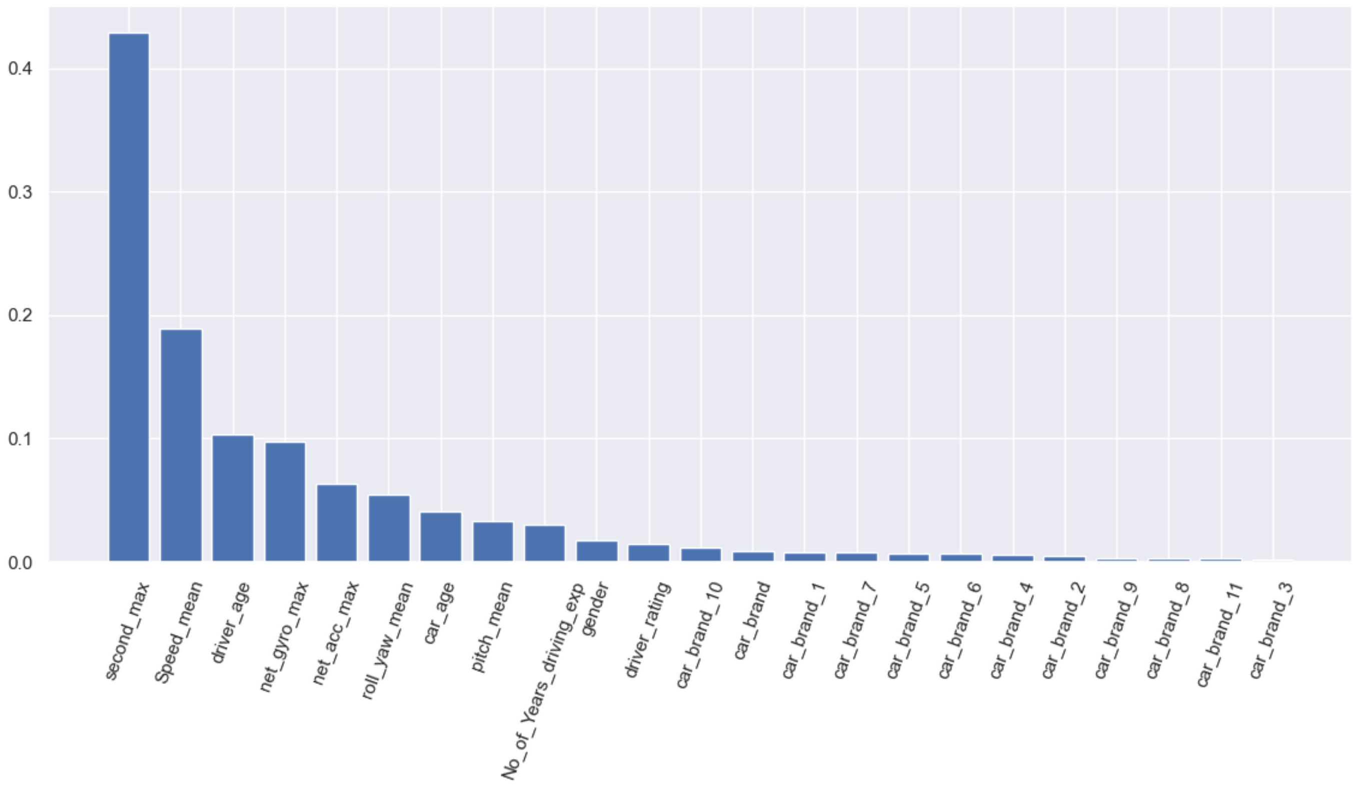
```
# perform oversampling
smote = SMOTE(random_state=42)
X, y = smote.fit_resample(X, y)

result = pd.DataFrame(columns=['Model'] + X.columns.to_list())

# there are two types of models, which does things differently
models_for_fi = [
    ("AdaBoost", 'non-linear', AdaBoostClassifier(n_estimators=50, learning_rate=1.0)
    ("GradientBoostingClassifier", 'non-linear', GradientBoostingClassifier(n_estimat
    ("LogisticRegression", 'linear', LogisticRegression(max_iter=1000, n_jobs=njobs))
    ("RandomForest", 'non-linear', RandomForestClassifier(n_estimators=100, max_depth
    ("Ridge", 'linear', RidgeClassifier())
]

for model_name, model_type, model in models_for_fi:

    model.fit(X, y)

    # get importance
    if model_type == 'linear':
        importances = model.coef_[0]
    else:
        importances = model.feature_importances_

    # get feature names
    features = model.feature_names_in_

    feature_imps = {'Model' : model_name}
    # store them in a dict
    for feature, importance in zip(features, importances):
        feature_imps[feature] = abs(importance) if model_type == 'linear' else import
    result = pd.concat([result, pd.DataFrame([feature_imps])], ignore_index=True)

# drop the model column into the index
result = result.set_index(result['Model']).drop(columns=['Model'])

sns.set(rc={'figure.figsize':(14,6)})
sns.heatmap(result)
plt.show()
```



In [101…

```
# get the columns and the average results
cols = result.columns
```

```
result_mean = result.mean(axis=0, numeric_only=True)

# sort the result
sorted_result = result_mean.reset_index().drop(columns=['index']).sort_values(0, asce
cols_sorted = [cols[i] for i in sorted_result.index.to_list()]

plt.bar(cols_sorted, sorted_result[0])
plt.xticks(rotation=70)
plt.show()
```



- Driver's car brands show negligible impact on predicting trip danger.
- Trip duration, harsh turn frequency, average speed, and harsh acceleration instances are significant predictors.
- Consequently, gender and car brand variables will be excluded from analysis.
- Focus shifts to numerical features, requiring standard scaling and combining yaw and roll as preprocessing steps.

We will apply these changes and observe if they improve the model.

In [93]:
```
df.columns
```

Out[93]:
```
Index(['second_max', 'Speed_mean', 'net_acc_max', 'net_acc_count_harsh_acc',
       'net_gyro_max', 'net_gyro_count_harsh_gyro', 'roll_mean', 'yaw_mean',
       'pitch_mean', 'No_of_Years_driving_exp', 'gender', 'car_brand',
       'driver_rating', 'driver_age', 'car_age', 'label', 'roll_yaw_mean'],
      dtype='object')
```

In [11]:
```
df3 = pd.read_hdf('./h5/very_cleaned_dataset.h5')
df["gender"] = df["gender"].replace({True:1,False:0})
# now just drop gender and car brands
df3.drop(columns=[ 'car_brand'], inplace=True)

X = df3.drop(columns=['label','roll_mean', 'yaw_mean','net_acc_count_harsh_acc','net_
y = df3['label']

X['roll_yaw_mean'] = (df['roll_mean']**2 + df['yaw_mean'] **2)**0.5
numeric_cols = list(set(X.columns.to_list()).difference({'car_brand', 'gender'}))

col_transformer = ColumnTransformer([
                        ('power_transform', PowerTransformer(method='yeo-johnson'), ['s
                        ('scaler', StandardScaler(), numeric_cols)
                        ])

X = col_transformer.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.2, random_state = 42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_test, y_test, test_size = 0.5, random_state = 42
)

smote = SMOTE(random_state=42)
X_train, y_train = smote.fit_resample(X_train, y_train)

mlflow.set_experiment("Filtered Features Run")
test_all_models(X_train, X_val, y_train, y_val)
```

Out[11]:

	Model name	Train Accuracy	Train Recall	Train F2 Score	Validation Accuracy	Validation Recall	Validation F2 Score
0	AdaBoost	0.693477	0.693477	0.693477	0.654211	0.569620	0.515464
1	DecisionTreeClassifier	1.000000	1.000000	1.000000	0.636316	0.424051	0.399602
2	ExtraTreesClassifier	1.000000	1.000000	1.000000	0.742632	0.358650	0.377610
3	GradientBoostingClassifier	0.765295	0.765295	0.764819	0.716842	0.489451	0.478548
4	HistGradientBoostingClassifier	0.847706	0.847706	0.845893	0.759474	0.337553	0.363802
5	KNeighborsClassifier	0.847486	0.847486	0.843924	0.586316	0.527426	0.461595
6	LogisticRegression	0.646537	0.646537	0.646354	0.648421	0.632911	0.557621
7	NaiveBayes	0.606320	0.606320	0.590143	0.741053	0.417722	0.428571
8	Perceptron	0.608122	0.608122	0.607975	0.638947	0.630802	0.552476
9	RandomForest	1.000000	1.000000	1.000000	0.727368	0.417722	0.423801
10	Ridge	0.646844	0.646844	0.646774	0.641053	0.635021	0.556171
11	SVC	0.684819	0.684819	0.684789	0.642632	0.645570	0.563951

- After removing car brand variables, overall scores exhibit a slight improvement.
- Priority shifts towards models with high recall or F2 score to minimize false negatives.
- Logistic Regression, Ridge, SVC, and Perceptron will be selected and fine-tuned due to their promising scores.

In [16]:
```
f2_scorer = make_scorer(fbeta_score, beta=2)

model_results = pd.DataFrame(columns=['Model name', 'Train Accuracy', 'Train Recall',
                                    'Val Accuracy', 'Val Recall', 'Val F2 Score']

def search_params(params, Model, model_name, model_results):
    with mlflow.start_run():
        model = Model(random_state=43)
        model_grid = GridSearchCV(model, params, cv=5, scoring=f2_scorer, n_jobs=njob
        model_grid.fit(X_train, y_train)

        best_lg = model_grid.best_estimator_
        y_train_pred = best_lg.predict(X_train)
        y_val_pred = best_lg.predict(X_val)
        signature = infer_signature(X_val, y_val_pred)

        print("Best hyperparameters for lg:", model_grid.best_params_)
        print("Train Accuracy:", accuracy_score(y_train, y_train_pred))
        print("Train Recall:", recall_score(y_train, y_train_pred))
        print("Train F2:", fbeta_score(y_train, y_train_pred, beta=2))

        print("Val Accuracy:", accuracy_score(y_val, y_val_pred))
        print("Val Recall:", recall_score(y_val, y_val_pred))
        print("Val F2:", fbeta_score(y_val, y_val_pred, beta=2))

        # Log parameters and metrics with MLflow
        mlflow.log_params(model_grid.best_params_)
        mlflow.log_metric("Train Accuracy", accuracy_score(y_train, y_train_pred))
        mlflow.log_metric("Train Recall", recall_score(y_train, y_train_pred))
        mlflow.log_metric("Train F2 Score", fbeta_score(y_train, y_train_pred, beta=2
        mlflow.log_metric("Val Accuracy", accuracy_score(y_val, y_val_pred))
        mlflow.log_metric("Val Recall", recall_score(y_val, y_val_pred))
        mlflow.log_metric("Val F2 Score", fbeta_score(y_val, y_val_pred, beta=2))

        mlflow.sklearn.log_model(
        sk_model=model_grid.best_estimator_,
        artifact_path="sklearn-model",
        signature=signature,
        registered_model_name=f"{model_name} - Tuning",
    )
        mlflow.set_tag('mlflow.runName', model_name)

        result = pd.DataFrame([{'Model name' : model_name,
                'Train Accuracy': accuracy_score(y_train, y_train_pred),
                'Train Recall': recall_score(y_train, y_train_pred),
                'Train F2 Score': fbeta_score(y_train, y_train_pred, beta=2),
                'Val Accuracy': accuracy_score(y_val, y_val_pred),
                'Val Recall': recall_score(y_val, y_val_pred),
                'Val F2 Score': fbeta_score(y_val, y_val_pred, beta=2)}])

        return pd.concat([model_results, result], ignore_index=True)
```

Hyperparameter Tuning

We will be finding the best combination of hyper parameters using GridSearch.

In [17]:
```
mlflow.set_experiment("Hyper Parameter Tuning")

logistic_regression_params = {
    'max_iter': [1000, 1500],
    'C': [0.1, 1, 10],
    'penalty': ['l2', 'l1', 'elasticnet']
}

model_results = search_params(logistic_regression_params, LogisticRegression, 'Logist

ridge_params = {
    'alpha' : [0, 0.01, 0.1, 0.2, 0.5, 0.75, 0.9, 1, 5, 10]
}

model_results = search_params(ridge_params, RidgeClassifier, 'Ridge', model_results)

perceptron_params = {
    'penalty' : ['l2', 'l1', 'elasticnet'],
    'alpha' : [0.00001, 0.00005, 0.0001, 0.0005, 0.001]
}

model_results = search_params(perceptron_params, Perceptron, 'Perceptron', model_resu
model_results
```

```
Best hyperparameters for lg: {'C': 0.1, 'max_iter': 1000, 'penalty': 'l2'}
Train Accuracy: 0.6467123769338959
Train Recall: 0.6227144866385372
Train F2: 0.628749955621827
Val Accuracy: 0.6468421052631579
Val Recall: 0.6329113924050633
Val F2: 0.5569996286669142
```

```
Successfully registered model 'Logistic Reg - Tuning'.
2024/02/08 15:57:33 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300
seconds for model version to finish creation. Model name: Logistic Reg - Tuning, versi
on 1
Created version '1' of model 'Logistic Reg - Tuning'.
```

```
Best hyperparameters for lg: {'alpha': 0.5}
Train Accuracy: 0.6468881856540084
Train Recall: 0.6315928270042194
Train F2: 0.6354807896412652
Val Accuracy: 0.6410526315789473
Val Recall: 0.6350210970464135
Val F2: 0.5561714708056171
```

```
Successfully registered model 'Ridge - Tuning'.
2024/02/08 15:57:34 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300
seconds for model version to finish creation. Model name: Ridge - Tuning, version 1
Created version '1' of model 'Ridge - Tuning'.
```

```
Best hyperparameters for lg: {'alpha': 0.001, 'penalty': 'l1'}
Train Accuracy: 0.5704553445850914
Train Recall: 0.9621132208157525
Train F2: 0.8318007022237084
Val Accuracy: 0.3773684210526316
Val Recall: 0.9514767932489452
Val F2: 0.6429997148560022
```

```
Successfully registered model 'Perceptron - Tuning'.
2024/02/08 15:57:36 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300
seconds for model version to finish creation. Model name: Perceptron - Tuning, version
1
Created version '1' of model 'Perceptron - Tuning'.
```

Out[17]:

	Model name	Train Accuracy	Train Recall	Train F2 Score	Val Accuracy	Val Recall	Val F2 Score
0	Logistic Reg	0.646712	0.622714	0.628750	0.646842	0.632911	0.557000
1	Ridge	0.646888	0.631593	0.635481	0.641053	0.635021	0.556171
2	Perceptron	0.570455	0.962113	0.831801	0.377368	0.951477	0.643000

Model	Tested Parameters	Best Parameters
Logistic Reg	`max_iter` : [1000, 1500], `C` : [0.1, 1, 10], `penalty` : ['l2', 'l1', 'elasticnet']	`C` : 0.1, `max_iter` : 1000, `penalty` : 'l2'
Ridge	`alpha` : [0, 0.01, 0.1, 0.2, 0.5, 0.75, 0.9, 1, 5, 10]	`alpha` : 0.001
Perceptron	`penalty` : ['l2', 'l1', 'elasticnet'], `alpha` : [0.00001, 0.00005, 0.0001, 0.0005, 0.001]	`penalty` : 'elasticnet', `alpha` : 0.00005

Final model

It looks like the logistic regressor, support vector classifier, and the ridge performed similarly, we pick the logistic regressor because SVC took a longer time to predict, while the ridge performed slightly worser in terms of recall and F2 score. This will be the parameters used for the logistic regressor: {'C': 0.1, 'max_iter': 1000, 'penalty': 'l2'}

In [15]:
```
mlflow.set_experiment("Final Model")
with mlflow.start_run():
    C = 0.1
    max_iter = 1000
    penalty = 'l2'
    lg = LogisticRegression(C=C, max_iter=max_iter, penalty=penalty)
    lg.fit(X_train, y_train)

    y_train_pred = lg.predict(X_train)
    train_accuracy = accuracy_score(y_train, y_train_pred)
    train_recall = recall_score(y_train, y_train_pred)
    train_f2 = fbeta_score(y_train, y_train_pred, beta=2)

    y_test_pred = lg.predict(X_test)
    signature = infer_signature(X_test, y_test_pred)
    test_accuracy = accuracy_score(y_test, y_test_pred)
    test_recall = recall_score(y_test, y_test_pred)
    test_f2 = fbeta_score(y_test, y_test_pred, beta=2)

    y_val_pred = lg.predict(X_val)
    val_accuracy = accuracy_score(y_val, y_val_pred)
    val_recall = recall_score(y_val, y_val_pred)
    val_f2 = fbeta_score(y_val, y_val_pred, beta=2)

    print(f"Train Accuracy: {train_accuracy:.4f}")
    print(f"Train Recall: {train_recall:.4f}")
    print(f"Train F2 Score: {train_f2:.4f}\n")

    print(f"Test Accuracy: {test_accuracy:.4f}")
    print(f"Test Recall: {test_recall:.4f}")
    print(f"Test F2 Score: {test_f2:.4f}\n")

    print(f"Val Accuracy: {val_accuracy:.4f}")
    print(f"Val Recall: {val_recall:.4f}")
    print(f"Val F2 Score: {val_f2:.4f}")

    mlflow.log_param("C", C)
    mlflow.log_param("max_iter", max_iter)
    mlflow.log_param("penalty", penalty)
    mlflow.log_metric("Train Accuracy", train_accuracy)
    mlflow.log_metric("Train Recall", train_recall)
    mlflow.log_metric("Train F2 Score", train_f2)
    mlflow.log_metric("Test Accuracy", test_accuracy)
    mlflow.log_metric("Test Recall", test_recall)
    mlflow.log_metric("Test F2 Score", test_f2)
    mlflow.log_metric("Val Accuracy", val_accuracy)
    mlflow.log_metric("Val Recall", val_recall)
    mlflow.log_metric("Val F2 Score", val_f2)
    mlflow.sklearn.log_model(
        sk_model=lg,
        artifact_path="sklearn-model",
        signature=signature,
        registered_model_name="Final Logistic Regression",
    )
    mlflow.set_tag('mlflow.runName', "Logistic Regression Model")
```

```
Train Accuracy: 0.6467
Train Recall: 0.6227
Train F2 Score: 0.6287

Test Accuracy: 0.6521
Test Recall: 0.6339
Test F2 Score: 0.5519

Val Accuracy: 0.6468
Val Recall: 0.6329
Val F2 Score: 0.5570
```

```
Successfully registered model 'Final Logistic Regression'.
2024/02/08 15:56:04 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300
seconds for model version to finish creation. Model name: Final Logistic Regression, v
ersion 1
Created version '1' of model 'Final Logistic Regression'.
```

Final Model Performance

Metric	Value
Train Accuracy	0.6467
Train Recall	0.6227
Train F2 Score	0.6287
Test Accuracy	0.6521
Test Recall	0.6339
Test F2 Score	0.5519
Validation Accuracy	0.6468
Validation Recall	0.6329
Validation F2 Score	0.5570

In [213…
```
df3.drop(columns=['label','roll_mean', 'yaw_mean','net_acc_count_harsh_acc','net_gyro
```

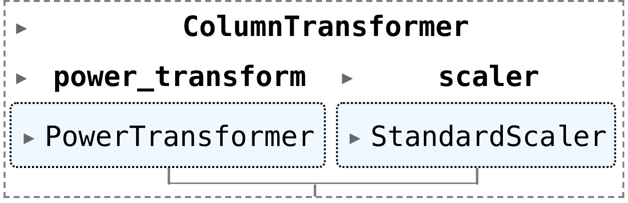
Out[213…
```
Index(['second_max', 'Speed_mean', 'net_acc_max', 'net_gyro_max', 'pitch_mean',
       'No_of_Years_driving_exp', 'gender', 'driver_rating', 'driver_age',
       'car_age', 'roll_yaw_mean'],
      dtype='object')
```

In [98]:
```
with open('./model.pkl', 'wb') as f:
    pickle.dump(lg, f)

with open('./transform.pkl', 'wb') as f:
    pickle.dump(col_transformer, f)
```

In [61]:
```
final = Pipeline([
    ('transformer', col_transformer),  # Apply column transformation
    ('model', lg)  # Apply logistic regression model
])

with open('./pipeline.pkl', 'wb') as f:
    pickle.dump(final, f)
```

In [97]:
```
col_transformer
```

Out[97]:



- **Power Transform**
  - Transformer: PowerTransformer()
  - Features:
    - second_max
    - Speed_mean
    - car_age
    - net_gyro_max
- **Standard Scaling**
  - Transformer: StandardScaler()
  - Features:
    - driver_rating
    - Speed_mean
    - net_gyro_max
    - pitch_mean
    - net_acc_max
    - No_of_Years_driving_exp
    - driver_age
    - roll_yaw_mean
    - second_max
    - car_age