DATA ANALYTICS NSSC'24

Cosmic Collision-
Analysing Asteroid Risks with Data

Team : Planetary Pookies
IIT Kharagpur

Nabayan Saha
Shuvraneel Mitra
Srinjoy Das
Agnij Biswas

Overview:

Asteroids are small, rocky objects that orbit the Sun, primarily found in the region between Mars and Jupiter known as the asteroid belt. They are remnants from the early solar system, formed over 4.6 billion years ago, and are considered planetesimals, or building blocks of planets that never coalesced into a larger body due to gravitational disturbances, particularly from Jupiter. Asteroid impacts have been a constant force shaping Earth's history. From minor events tha leave barely a trace to catastrophic collisions that have caused mass extinctions, these cosmic encounters have played a significant role in our planet's evolution. Given the potential for devastating consequences, scientists worldwide are dedicated to detecting and tracking the asteroid impact threat. To mitigate this risk, it is crucial to identify and analyse asteroids that could pose a hazard.

Objective of the Analysis:

The primary objective of this analysis is to use data analytics and machine learning techniques to determine the likelihood of an asteroid being hazardous to Earth based on various features provided in the dataset. This includes examining characteristics such as the asteroid's size, orbital parameters, velocity, and proximity to Earth's orbit. By analysing these features, the goal is to develop predictive models that classify asteroids into hazardousand and non hazardous categories. This classification is crucial for prioritising which asteroids require further monitoring, potential deflection efforts, or other mitigation strategies.
We will also identify asteroids exhibiting unusual or anomalous behaviour that may warrant closer attention.

Data Description:

The given dataset contains a total of 24 features out of which some are redundant as some features represent same physical quantities in different units. All 24 features are as following:
'Name', 'Epoch Date Close Approach', 'Relative Velocity km per sec', 'Relative Velocity km per hr', 'Miles per hour', 'Miss Dist.(Astronomical)', 'Miss Dist.(lunar)', 'Miss Dist. (kilometers)', 'Miss Dist.(miles)', 'Jupiter Tisserand Invariant', 'Epoch Osculation', 'Semi Major Axis', 'Asc Node Longitude', 'Perihelion Arg', 'Aphelion Dist', 'Perihelion Time', 'Mean Anomaly', 'Mean Motion', 'approach_year', 'approach_month', 'approach_day', 'Orbital Period', 'Orbit Uncertainity', 'Hazardous'

1.Explanatory Data Analysis:

1.1 Data Inspection

As described earlier the dataset contains a total of 24 features out of which only 6 are Categorical and the rest 18 are Numerical in nature.

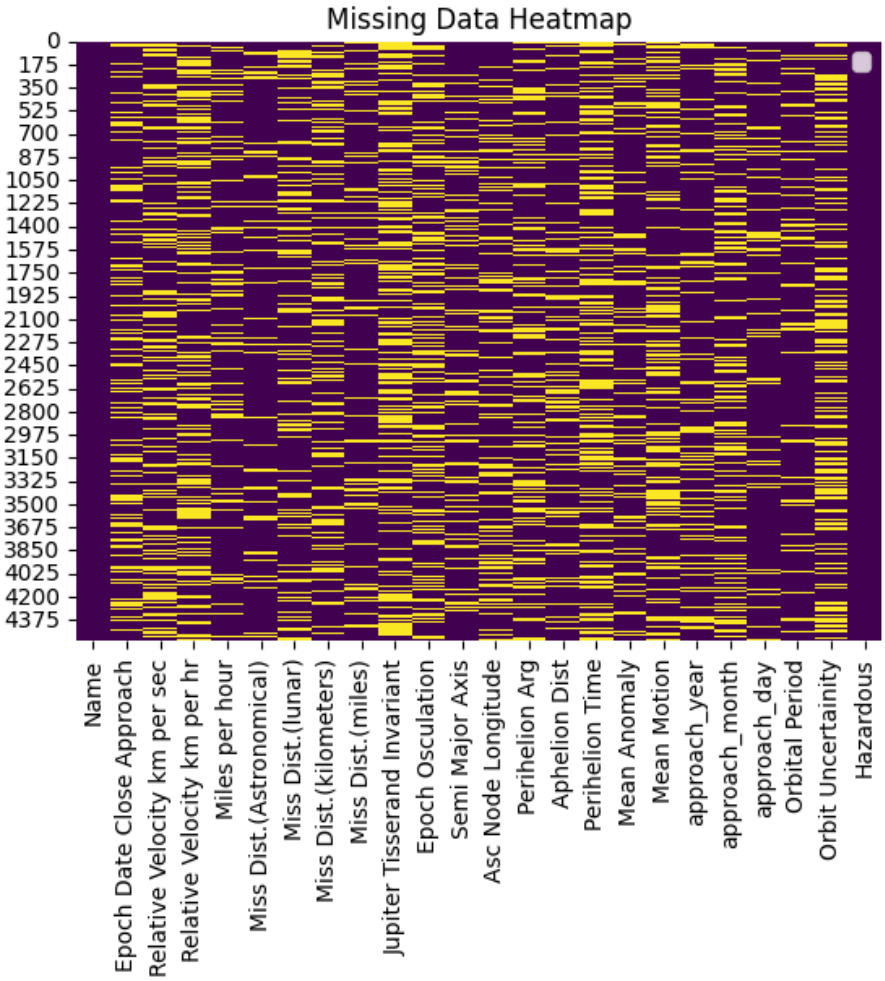
Feature Name	Type of Feature	Feature Name	Type of Feature	Feature Name	Type of Feature
Epoch Date Close Approach	Numerical	Miss Dist. (miles)	Numerical	Mean Anomaly	Numerical
Name	Numerical	Jupiter Tisserand Invariant	Numerical	Mean Motion	Numerical
Relative Velocity km per sec	Categorical	Epoch Osculation	Numerical	approach_year	Numerical
Relative Velocity km per hr	Numerical	Semi Major Axis	Numerical	approach_month	Categorical
Miles per hour	Numerical	Asc Node Longitude	Numerical	approach_day	Categorical
Miss Dist. (Astronomical)	Numerical	Perihelion Arg	Numerical	Orbital Period	Categorical
Miss Dist. (lunar)	Numerical	Aphelion Dist	Numerical	Orbit Uncertainity	Categorical
Miss Dist. (kilometers)	Numerical	Perihelion Time	Numerical	Hazrdous	Categorical

These 18 numerical features have different range but some of them show similar statistical behavior which is as following

Feature	Range	Mean	Median	Standard Deviation	1st Quartile	3rd Quartile
Relative Velocity km per hr	159473.673047	50516.969113	46968.245275	26530.144294	30437.415189	65210.346095
Miles per hour	99090.738678	31312.455735	28959.416222	16386.183907	18843.393552	40331.941346
Miss Dist. (Astronomical)	0.499706	0.258221	0.265281	0.146070	0.135807	0.387033
Miss Dist. (lunar)	194.171981	100.709883	104.2614517	56.93873947	52.87751389	150.434433
Miss Dist. (kilometers)	74754990	38424405	39879014	22074424	19503181	57699616
Miss Dist. (miles)	46450597	23911785	24764642	13575955	12466343	35817823
Jupiter Tisserand Invariant	6.658	5.126265	5.1025	1.197144	4.17925	6.043
Epoch Osculation	7064	2457720	2458001	924.8399	2458001	2458001
Semi Major Axis	1.952633	1.358242	1.223551	0.465313	0.990008	1.62635
Asc Node Longitude	359.9039	172.1858	173.8952	103.0559	83.28843	253.6353
Perihelion Arg	359.9862	185.1489	192.4201	103.4172	96.10051	273.068
Aphelion Dist	3.858393	1.898115	1.590005	0.834499	1.260397	2.331365

Perihelion Time	8605.42	2457741	2457976	915.5433	2457826	2458109
Mean Anomaly	359.9148	182.7352	189.0511	107.757	87.06667	278.0387
Mean Motion	1.707437	0.76109	0.737684	0.337745	0.483841	1.00287

Multiple Features of the given data contains NaN values which are as following 'Epoch Date Close Approach', 'Relative Velocity km per sec', 'Relative Velocity km per hr', 'Miles per hour', 'Miss Dist.(Astronomical)', 'Miss Dist.(lunar)', 'Miss Dist.(kilometers)', 'Miss Dist.(miles)', 'Jupiter Tisserand Invariant', 'Epoch Osculation', 'Semi Major Axis', 'Asc Node Longitude', 'Perihelion Arg', 'Aphelion Dist', 'Perihelion Time', 'Mean Anomaly', 'Mean Motion', 'approach_year', 'approach_month', 'approach_day', 'Orbital Period', 'Orbit Uncertainty'.



We can clearly see that only ‘Name’ and ‘Hazardous’ do not contain any missing values.

Numerical columns: 'Epoch Date Close Approach', 'Relative Velocity km per hr', 'Miles per hour', 'Miss Dist. (Astronomical)', 'Miss Dist. (lunar)', 'Miss Dist. (kilometers)', 'Miss Dist. (miles)', 'Jupiter Tisserand Invariant', 'Epoch Osculation', 'Semi Major Axis', 'Asc Node Longitude', 'Perihelion Arg', 'Aphelion Dist', 'Perihelion Time', 'Mean Anomaly', 'Mean Motion', 'approach_year', 'approach_month'
Categorical columns: 'Name', 'Relative Velocity km per sec', 'approach_day', 'Orbital Period', 'Orbit Uncertainty', 'Hazardous'

Imputation:

The imputation process handles missing values in the dataset through several tailored methods:

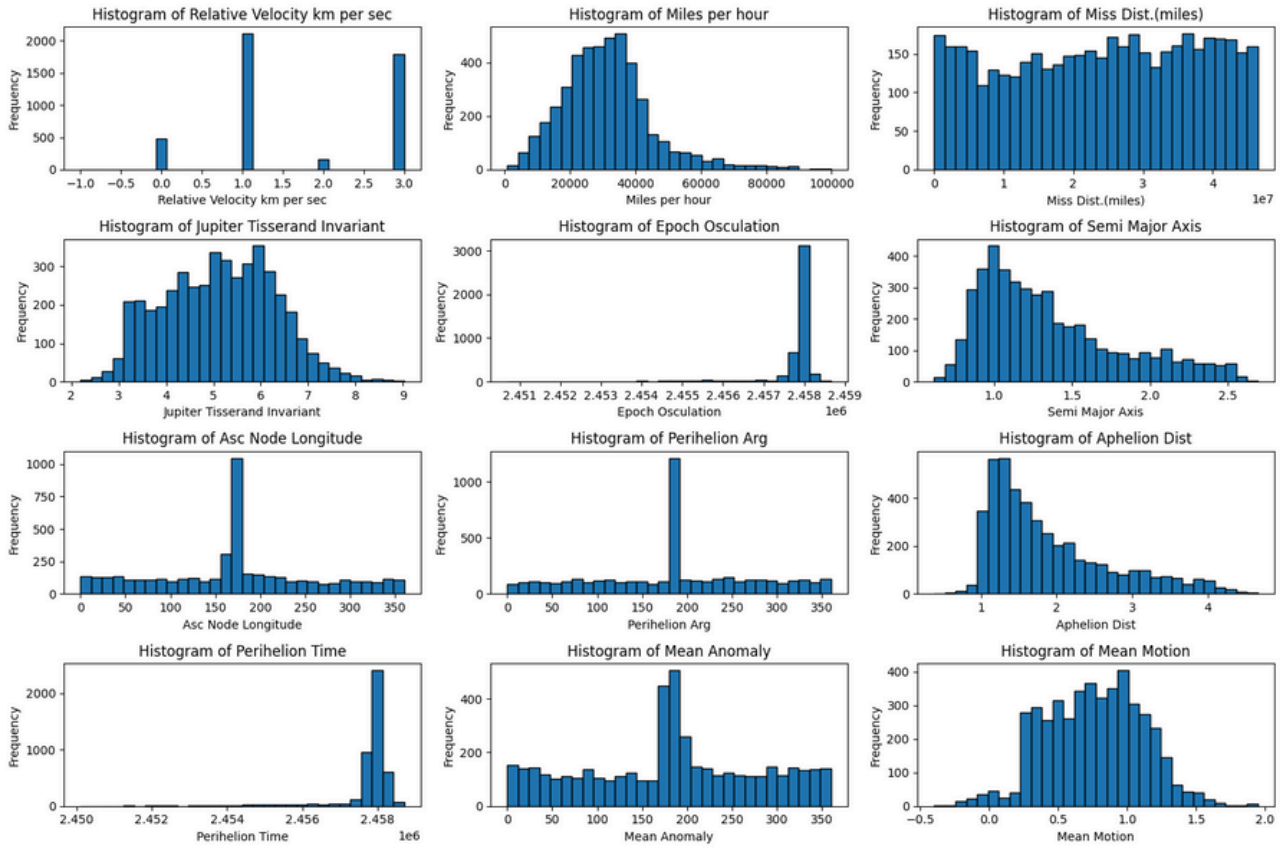
1. **Speed Imputation (impute_speed function):**
    - Missing values in the "Miles per hour" column are filled using "Relative Velocity km per sec."
    - Speeds are categorized as "Very Slow," "Slow," "Fast," or "Very Fast" based on defined thresholds.
2. **Orbital Period Imputation (impute_orbital function):**
    - "Mean Motion" is categorized as "High," "Medium," or "Low" based on specified values.
3. **Filling Missing Days (fill_day_nan function):**
    - Missing days in the "approach_day" column are filled using forward and backward filling methods and clipped to a valid range (1-31).

```
def impute_data(data):
    mph_calc = data['Relative Velocity km per hr'] * 0.621371
    mph_act = data['Miles per hour']
    data['Miles per hour'] = np.where(mph_act.isna(), mph_calc, (mph_calc + mph_act) / 2)
    data.drop(['Relative Velocity km per hr'], axis=1, inplace=True)

    data['Relative Velocity km per sec'] = data.index.map(impute_speed)

    miss_miles1 = data['Miss Dist.(kilometers)'] * 0.621371
    miss_miles2 = data['Miss Dist.(Astronomical)'] * 92955807.267433
    miss_miles3 = data['Miss Dist.(lunar)'] / 4.186628026462e-6
    data['Miss Dist.(miles)'] = np.nanmean([miss_miles1, miss_miles2, miss_miles3, data['Miss Dist.(miles)']], axis=0)
    data.drop(['Miss Dist.(kilometers)', 'Miss Dist.(Astronomical)', 'Miss Dist.(lunar)'], axis=1, inplace=True)

    data['Orbital Period'] = data.index.map(impute_orbital)

    le = LabelEncoder()
    for col in ['Relative Velocity km per sec', 'Orbital Period', 'Orbit Uncertainity']:
        data[col] = le.fit_transform(data[col])
        data[col] = data[col].replace(data[col].max(), np.nan)

    data['approach_year'] = data['approach_year'].fillna(method='ffill').fillna(method='bfill')
    data['approach_month'] = data['approach_month'].fillna(method='ffill').fillna(method='bfill')
    data = fill_day_nan(data, 'approach_day')
    data['Epoch Date Close Approach'] = fill_nan_with_mean(data['Epoch Date Close Approach'])

    imputer = IterativeImputer(random_state=42)
    imputed_data = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)

    for col in ['Relative Velocity km per sec', 'Orbital Period', 'Orbit Uncertainity']:
        imputed_data[col] = imputed_data[col].astype(int)
    imputed_data[['approach_day', 'approach_year', 'approach_month']] = data[['approach_day', 'approach_year', 'approach_month']].astype(int)

    return imputed_data

df_imputed = impute_data(df.copy(deep=True))
```

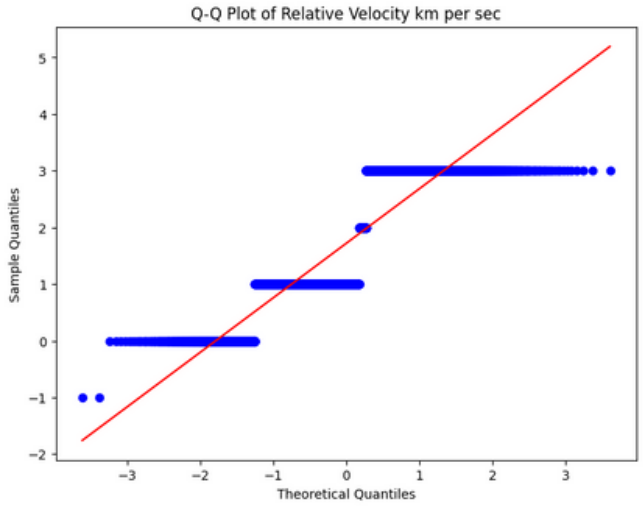
4. **Numerical Feature Imputation:**
    - For "Miles per hour," if values are missing, they are calculated from "Relative Velocity km per hr." Actual and calculated values are averaged where both exist, and "Relative Velocity km per hr" is then dropped.
5. **Miss Distance Imputation:**
    - The "Miss Dist. (miles)" column is filled by averaging converted values from "Miss Dist. (kilometers)," "Miss Dist. (Astronomical)," and "Miss Dist. (lunar)."
6. **Encoding and Dropping Columns:**
    - Categorical columns like "Relative Velocity km per sec" and "Orbital Period" are label-encoded. Maximum encoded values are replaced with NaN.
7. **Forward and Backward Fill:**
    - Missing values in "approach_year" and "approach_month" are filled similarly, while "approach_day" is also clipped between 1 and 31.
8. **Epoch Date Close Approach:**
    - Missing values in this column are interpolated using a linear interpolation method.
9. **Iterative Imputer:**
    - Finally, the Iterative Imputer fills any remaining missing values by modeling each feature with missing values based on other features. Categorical columns are cast back to integers after imputation.

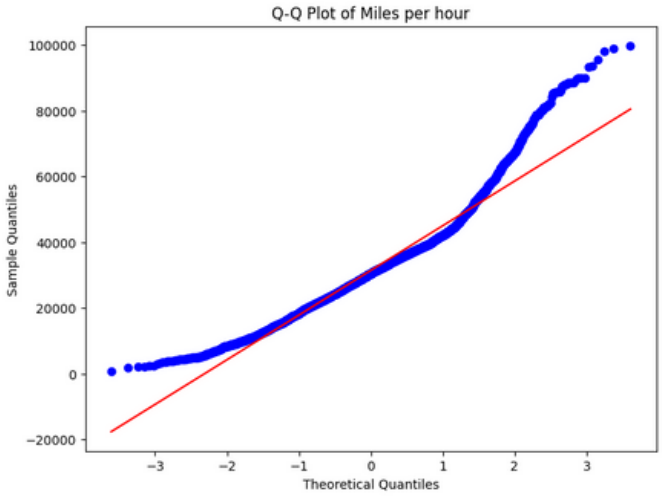
1.2 Statistical Inference

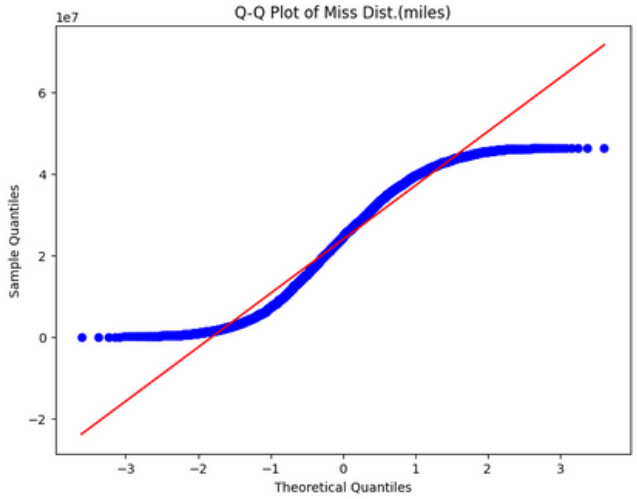
distribution of numerical features to assess the skewness of the data

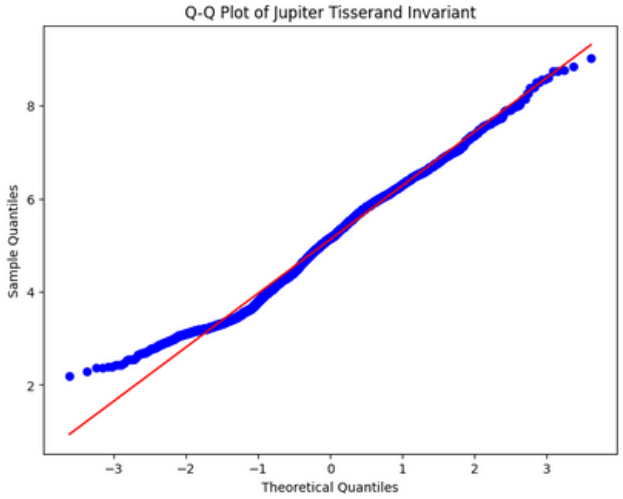


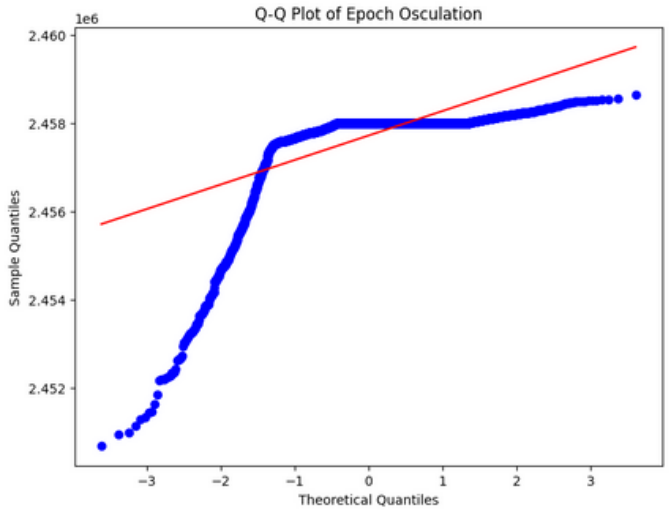
The `Relative velocity km per hour` and `Miles per hour` columns seem the most close to normally distributed, but the other columns show either a skewed distribution or an almost uniform one. This the dataset **needs normalisation**, which can be further confirmed by the use of a **Q-Q plot**.

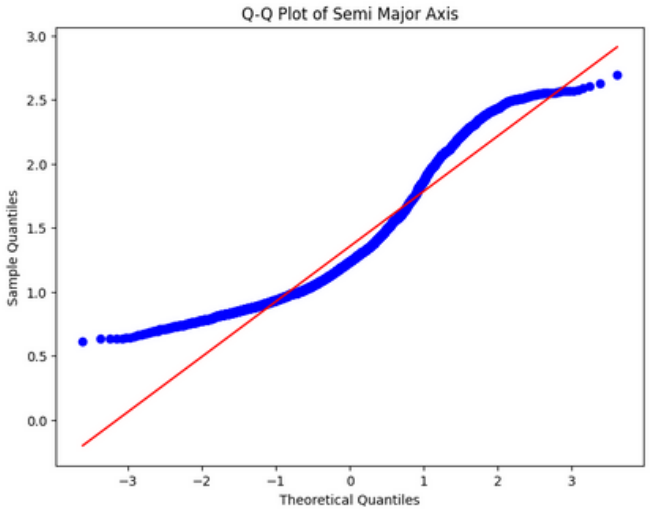


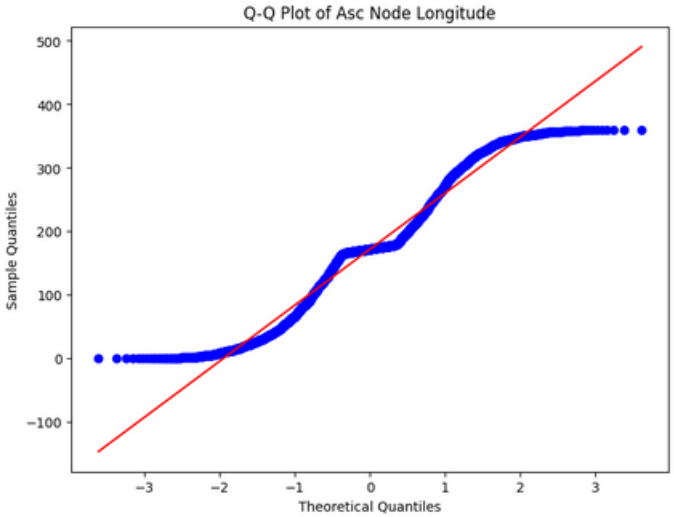


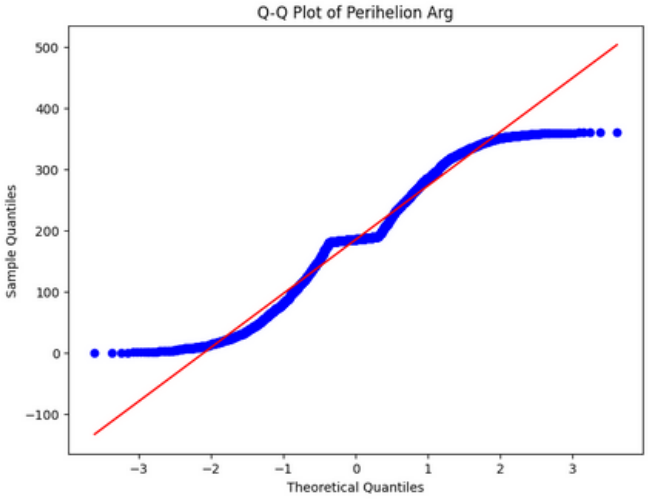


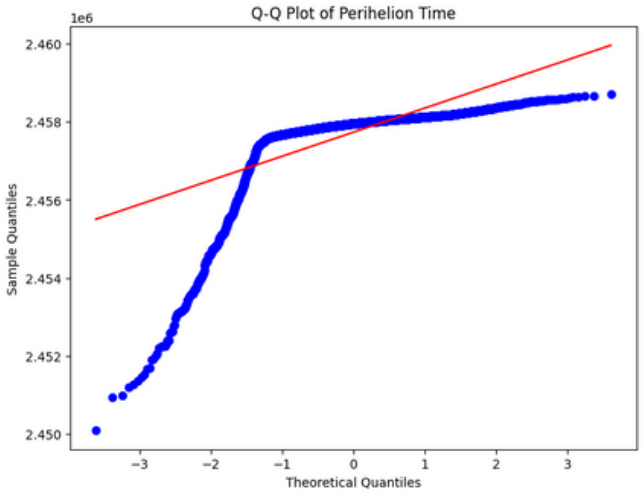


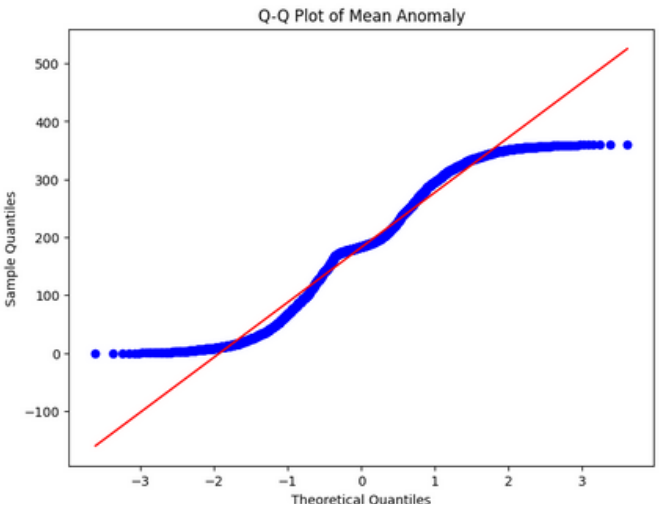


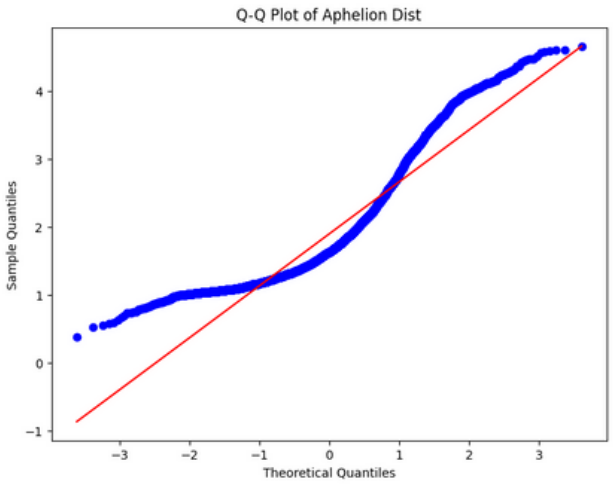


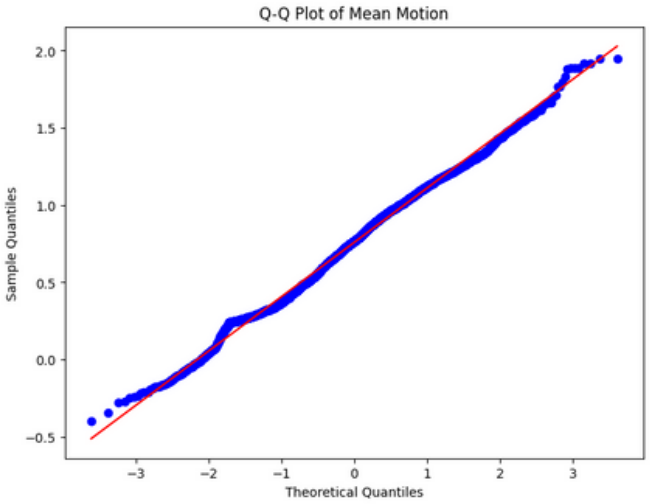

Q-Q Plot of Asc Node Longitude
Sample Quantiles
Theoretical Quantiles

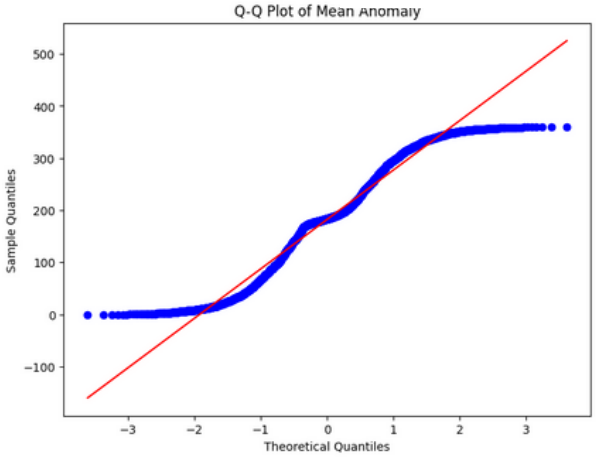

Q-Q Plot of Perihelion Arg
Sample Quantiles
Theoretical Quantiles


Q-Q Plot of Perihelion Time
1e6
Sample Quantiles
Theoretical Quantiles


Q-Q Plot of Mean Anomaly
Sample Quantiles
Theoretical Quantiles


Q-Q Plot of Aphelion Dist
Sample Quantiles
Theoretical Quantiles


Q-Q Plot of Mean Motion
Sample Quantiles
Theoretical Quantiles


Q-Q Plot of Mean Anomaly
Sample Quantiles
Theoretical Quantiles

The Z-score is a statistical measurement that describes how many standard deviations a data point is from the mean. A Z-score of 0 indicates that the data point is exactly at the mean. Z-scores larger than 3 (or smaller than -3) are typically considered outliers.

$$Z = \frac{x - \mu}{\sigma}$$

Z = standard score

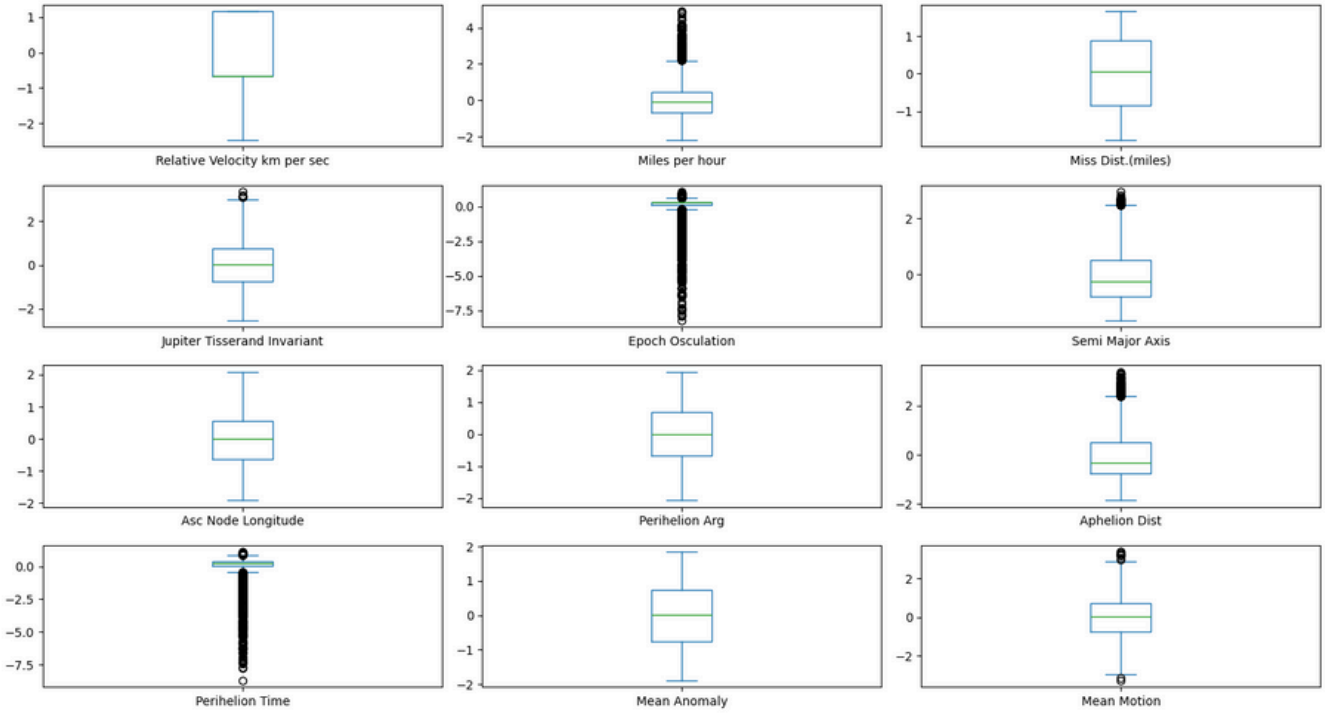
x = observed value

μ = mean of the sample

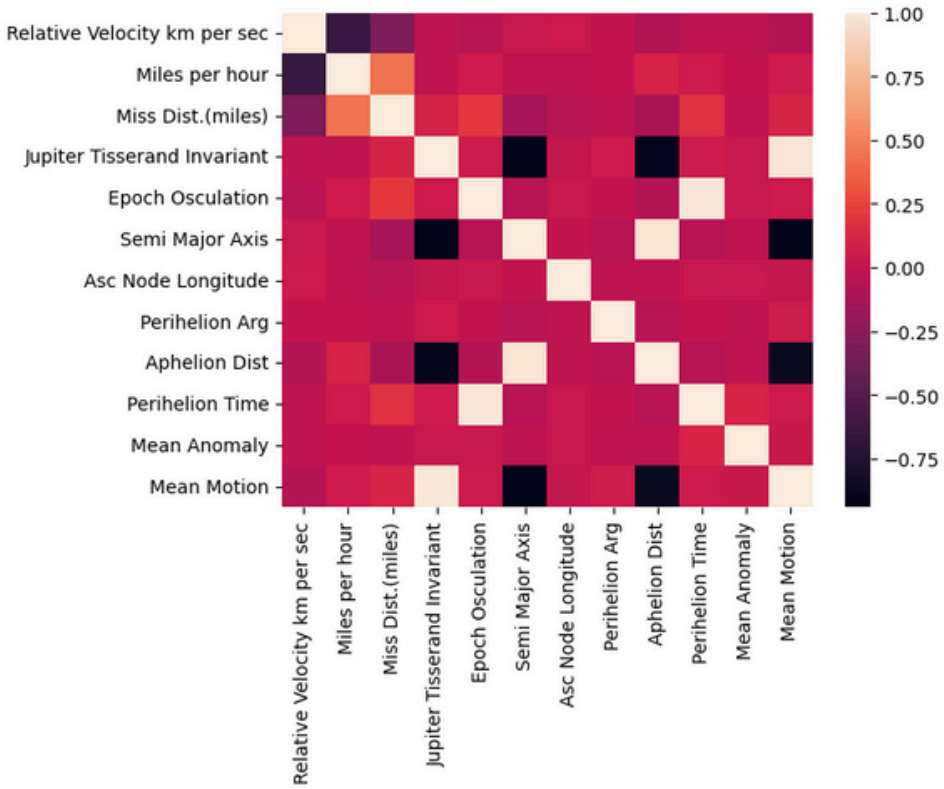
σ = standard deviation of the sample

```python
def detect_outliers_z_score(df, threshold=3):
    numerical_columns = df.select_dtypes(include=['number']).columns
    outliers = {}

    for column in numerical_columns:
        if column not in ["Name", "Epoch Date Close Approach", "approach_year",
                          "approach_month", "approach_day", "Orbital Period", "Orbit Uncertainity", "Hazardous"]:
            col_zscore = (df[column] - df[column].mean()) / df[column].std()
            outliers[column] = df[np.abs(col_zscore) > threshold]

    return outliers

for i in (outliers:=detect_outliers_z_score(df_scaled)):
    print(f"These are the Z-score based outliers for the column { i }")
    display(outliers[i])
```

A box plot (or box-and-whisker plot) visually represents the distribution of a dataset. It highlights the minimum, maximum, median (Q2), and the interquartile range (IQR) through a box with "whiskers" extending to the smallest and largest non-outlier points.
The box is drawn from the first quartile (Q1) to the third quartile (Q3), covering the interquartile range (IQR). The whiskers extend from the box to the smallest and largest data points within 1.5 times the IQR from Q1 and Q3. Outliers are plotted as individual points beyond the whiskers. Outlier condition: Points beyond the whiskers (outside(Q1 - 1.5 IQR, Q3 + 1.5 IQR)) are visually marked as outliers.



A correlation matrix plot visually shows how variables are related, with color coding or values representing the strength and direction of correlations. It's useful for identifying relationships, selecting features, and detecting multicollinearity. This helps in improving model performance by addressing redundant or highly correlated features



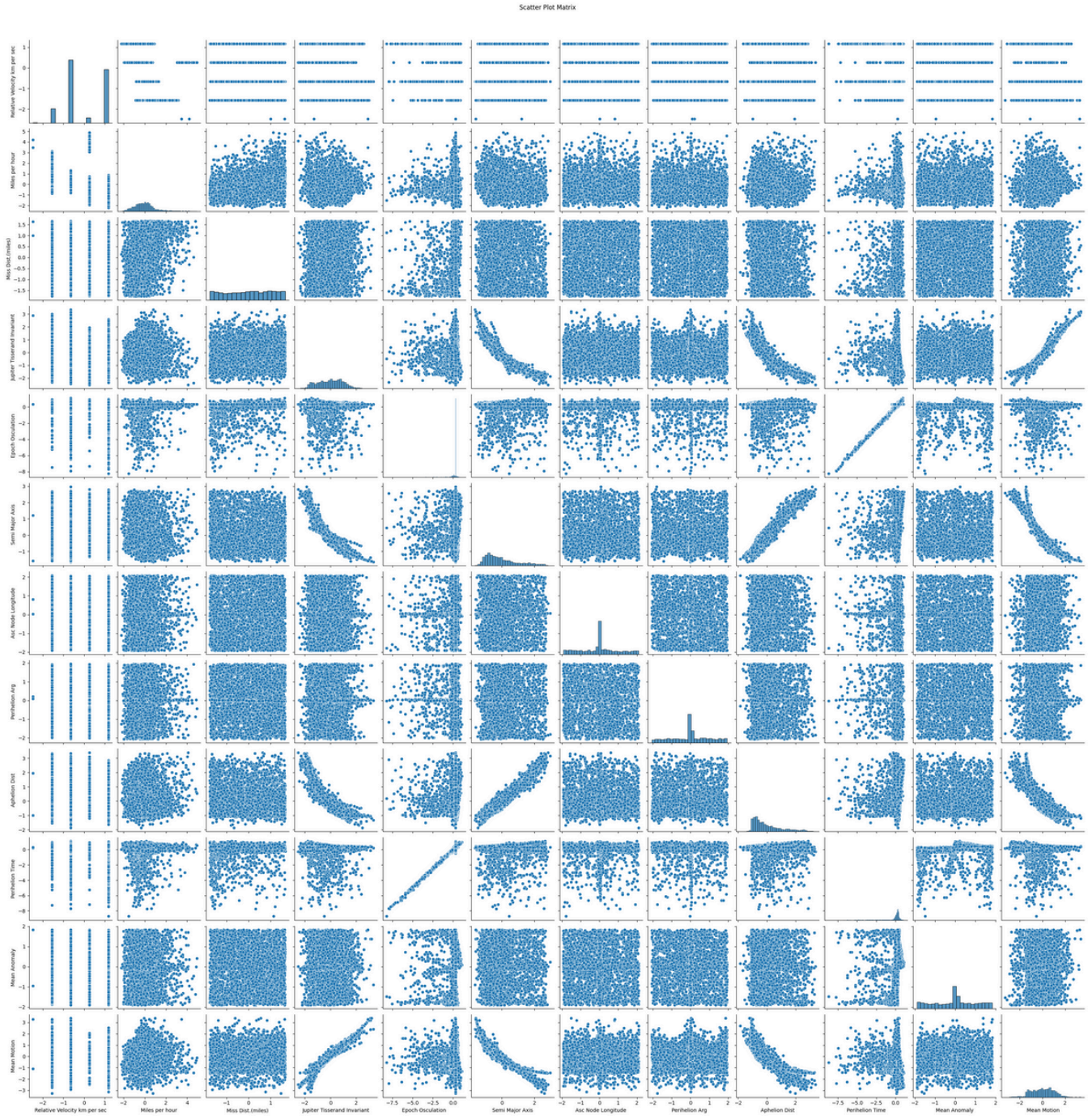
As we can see from the correlation matrix that few features are highly correlated with each other which indicates the redundancy of some features.

1.3 Visualisation:

We can see that the following pairs of columns have strong linear relationships:

1. Jupiter Tisserand Invariant and Mean motion
2. Jupiter Tisserand Invariant and Aphelion Distance
3. Jupiter Tisserand Invariant and Semi Major Axis
4. Epoch Osculation and Perihelion Time
5. Semi Major Axis and Mean motion
6. Semi Major Axis and Aphelion Dist.
7. Aphelion Dist. and Mean motion

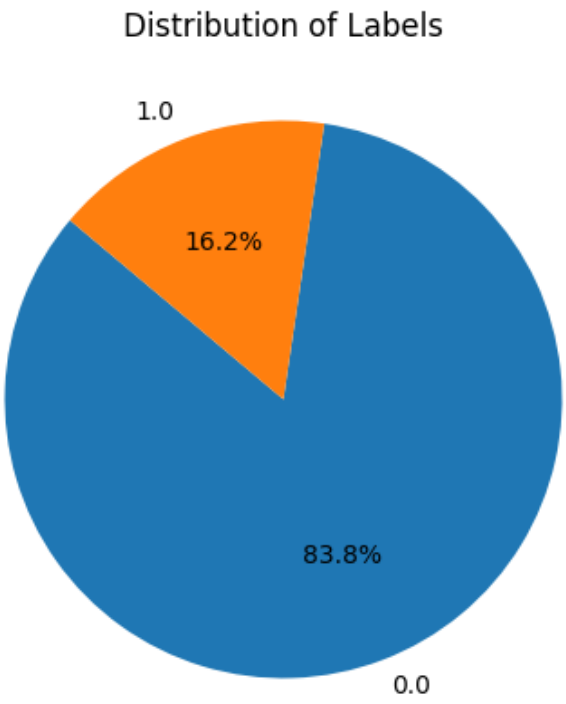
Given this set of associations, there is a strong possibility that the feature "Jupiter Tisserand Invariant" is **simply a confounder** in pairs Semi Major Axis and Mean motion, Semi Major Axis and Aphelion Dist., Aphelion Dist. and Mean motion.



The diagonal plots in a pairplot usually show the distribution of individual variables (features) in the dataset. These are often represented as histograms or kernel density estimation (KDE) plots, depending on the configuration. The diagonal plots provide insight into the univariate distribution of each variable. This helps understand the spread, skewness, and modality of each feature.

The off-diagonal plots display scatter plots (or sometimes regression plots) between every pair of variables in the dataset. These plots show the bivariate relationships between pairs of variables, helping you understand potential correlations or interactions between features.

1.4 Tackling Class Imbalance



Thus we can see that there is a high degree of class imbalance in the given data. We can tackle it by using SMOTE. SMOTE (Synthetic Minority Over-sampling Technique) is used to address the issue of class imbalance in datasets, particularly in classification problems. SMOTE works by generating synthetic samples of the minority class rather than simply duplicating existing ones. It does this by selecting data points from the minority class and interpolating between them to create new, similar instances. This helps balance the class distribution and improves the model's ability to generalize to the minority class.

IMPORTANT: In order to ensure that no information leaks over into the training set from the test set, we first split the data into trian and test sets with test_size = 0.2 and a random state for reproducibility, and only apply the SMOTE technique to the train set.

```
X = df_scaled.drop(columns=["Hazardous"])
y = df_scaled["Hazardous"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
smote_tomek = SMOTETomek(random_state=42,sampling_strategy=1)
X_resampled, y_resampled = smote_tomek.fit_resample(X_train, y_train)
df_resampled = pd.concat([X_resampled, y_resampled], axis=1)
print("Class distribution before resampling:")
print(y.value_counts())
print("\nClass distribution after resampling:")
print(pd.Series(y_resampled).value_counts())
```

Class imbalance occurs when one class in a dataset significantly outnumbers another, leading to biased machine learning models that favor the majority class. This imbalance can result in misleading accuracy metrics, as models may predict the majority class correctly while failing to identify instances of the minority class, which is often crucial in applications like fraud detection or medical diagnosis. Consequently, sensitivity (recall) for the minority class decreases, which can have serious real-world implications. To address these challenges, it's essential to use alternative evaluation metrics, such as precision, recall, and F1-score, and apply techniques like SMOTE or undersampling to create a more balanced dataset, ensuring the model performs well across all classes and effectively generalizes to unseen data.

2.Numerical Interpretation and Mathematical Analysis

2.1 Feature Engineering:

Given data contains approach day, approach month and approach year as separate fields which can be combined into one approach date field, which will be a datetime feature also 'Epoch Date Close Approach' will also give the same datetime value as that represents the same date (again represented in ms).

```
def date(year, month, day):
    return datetime.datetime(year, month, day)

df_resampled.drop(['Epoch Date Close Approach'], axis=1, inplace=True)
df_resampled['approach_date'] = df_resampled.apply(lambda x: date(int(x['approach_year']), int(x['approach_m
df_resampled['Time Until Approach'] = df_scaled['Epoch Date Close Approach'] - int(datetime.datetime.now().t
df_resampled['Time Until Approach'] = df_resampled['Time Until Approach'].apply(lambda x: x/(1000*60*60*24*3
```

Further we calculated the ratio of Miss distance and Semi major axis (indicating the closest approach it'll make)

```
df_resampled['Miss_Distance_ratio'] = 1.609*df_resampled['Miss Dist.(miles)'] / df_resampled['Semi Major Axi
```

Further Time until Closest Approach is Calculated which indicates how much urgency is there if the asteroid is hazardous in nature.

```
df_resampled['Time Until Approach'] = df['Epoch Date Close Approach'] - int(datetime.datetime.now().timestam
```

After this Average Orbital Velocity, eccentricity of the orbit and Orbital period is calculated.

$$A = a(1 + e) \qquad P^2 = [4p^2/G(m_1 + m_2)]a^3. \qquad v^2 = G(m_1 + m_2)\,(2/r - 1/a).$$

```
G = 6.67430e-11  # Gravitational constant (m^3 kg^-1 s^-2)
M_sun = 1.989e30  # Mass of the Sun (kg)
```

```
df_resampled['eccentricity'] = (df_resampled['Aphelion Dist']/df_resampled['Semi Major Axis']) - 1
df_resampled['eccentricity'] = df_resampled['eccentricity'].apply(lambda x: 0 if x<0 else x)

ap = 5.2
df_resampled['Inclination'] =np.sqrt(ap/(df_resampled['Semi Major Axis']*(1-df_resampled['eccentricity']**2)
r = df_resampled['Semi Major Axis']*(1-df_resampled['eccentricity']**2)/(1+df_resampled['eccentricity']*np.c
df_resampled['Avg Orbital Velocity'] = np.sqrt(G*M_sun*(2/r - 1/df_resampled['Semi Major Axis']))*(1/1.496e11

df_resampled['Orbital Period'] = np.sqrt((4*(np.pi**2)/(G*M_sun))*(1.496e11*df_resampled['Semi Major Axis'])
```

Now we calculated the heliocentric distance, escape velocity, and specific orbital energy for the specified asteroids.

```
df_resampled['heliocentric_distance'] = df_resampled['Semi Major Axis'] * (1 - df_resampled['eccentricity']*

df_resampled['escape_velocity'] = np.sqrt(2 * G * M_sun / (1.496e11*df_resampled['heliocentric_distance']))

df_resampled['Specific orbital energy'] = - M_sun / 2 * df_resampled['Semi Major Axis']
```

From the given Data then Specific Angular momentum is calculated as following

$$h = \sqrt{GMa(1-e^2)}$$

```
df_resampled['specific_angular_momentum'] = np.sqrt(G * M_sun * df_resampled['Semi Major Axis'] * (1 - df_re
```

Now the velocity at aphelion and Perihelion is calculated according to the following formulas

$$v_{peri} = (2pa/P)[(1 + e)/(1 - e)]^{1/2} \qquad v_{ap} = (2pa/P)[(1 - e)/(1 + e)]^{1/2}.$$

```
df_resampled['Aphelion Velocity'] = (2*np.pi*df_resampled['Semi Major Axis']*1.496e11/df_resampled['Orbital
df_resampled['Perhelion Velocity'] = (2*np.pi*df_resampled['Semi Major Axis']*1.496e11/df_resampled['Orbital
```

The Closest Distance of approach for different Catagories is Calculated as following

```
average_miss_distance = df.groupby('Hazardous')['Miss Dist.(miles)'].mean().reset_index()

closest_approach_distance = df['Miss Dist.(miles)'].min()

print("Average Miss Distance by Category:")
print(average_miss_distance)
print("\nClosest Approach Distance (in miles):", closest_approach_distance)
```

```
Average Miss Distance by Category:
   Hazardous  Miss Dist.(miles)
0      False       2.363638e+07
1       True       2.535247e+07

Closest Approach Distance (in miles): 16534.6171875
```

Finally we computed the values for synodic period and Mean motion.

```
earth_orbital_period = 365.25*24*60*60  # days
df_resampled['mean_motion'] = 2 * np.pi / df_resampled['Orbital Period']
df_resampled['synodic_period'] = np.abs(1 / df_resampled['Orbital Period'] - 1 / earth_orbital_period)
df_resampled['synodic_period'] = 1 / df_resampled['synodic_period']
```

2.2 Additional Features:

ω is defining the angular velocity of the object when it is closest to the earth.

```
df_resampled['Omega'] = df_resampled['Miles per hour']*df_resampled['Miss Dist.(miles)']
df_test['Omega'] = df_test['Miles per hour']*df_test['Miss Dist.(miles)']
```

The AutoFeatClassifier is a tool from the AutoFeat Python library that automates feature engineering and selection for classification tasks. It can create, combine, and select the best features from your dataset to improve model performance without requiring you to manually craft new features.

```
X = df_resampled.drop(columns=['Name', 'approach_year', 'approach_month', 'approach_day','approach_date','Epoch Osculation','Relative Velocity km per se
       'Orbital Period', 'Orbit Uncertainity','Hazardous'])
y = df_resampled['Hazardous']
autofeature = AutoFeatClassifier()
X_transformed = autofeature.fit_transform(X, y)
X_transformed_df = pd.DataFrame(X_transformed, columns=X_transformed.columns)
df_resampled = pd.concat([df_resampled.reset_index(drop=True), X_transformed_df.reset_index(drop=True)], axis=1)
X_test = df_test.drop(columns=['Name', 'approach_year', 'approach_month', 'approach_day','approach_date','Epoch Osculation','Relative Velocity km per se
       'Orbital Period', 'Orbit Uncertainity','Hazardous'])  # Again, replace with your target column name
X_test_transformed = autofeature.transform(X_test)
X_test_transformed_df = pd.DataFrame(X_test_transformed, columns=X_test_transformed.columns)
df_test = pd.concat([df_test.reset_index(drop=True), X_test_transformed_df.reset_index(drop=True)], axis=1)
print("Training dataset with new features:\n", df_resampled.head())
print("Test dataset with new features:\n", df_test.head())
```

AutoFeat automates the process of creating and selecting features, saving time and reducing human error. It explores complex feature combinations that might be overlooked in manual feature engineering. This leads to improved model performance, especially in cases with intricate relationships between features.

3. Handling Binned Values:

Binned features are categorical variables where values are grouped into discrete categories such as: [very slow, slow, fast, very fast, etc.]

```
X = df_scaled.drop(columns=["Hazardous"])
y = df_scaled["Hazardous"]

le = LabelEncoder()
for cols in ["Relative Velocity km per sec", "Orbital Period", "Orbit Uncertainity"]:
    X[cols] = le.fit_transform(X[cols])
y = le.fit_transform(y)
```

Also all categorical variables have an ordinal relationship in the categories.

4. Hazardous Classification:

To classify the data we are going to use a boosting model specifically XGBoost.
XGBoost (Extreme Gradient Boosting) is a highly efficient and scalable machine learning algorithm based on the principle of gradient boosting, which builds an ensemble of decision trees to improve predictive accuracy.

```
X_resampled = X_resampled.values
X_test = X_test.values
xgb_model = XGBClassifier(
    objective='binary:logistic',
    alpha=0.01,
    lambda_=0.1,
    eval_metric='logloss',
    use_label_encoder=False
)
xgb_model.fit(X_resampled, y_resampled)
y_pred_proba = xgb_model.predict_proba(X_test)[:, 1]
y_pred = (y_pred_proba > 0.5).astype(int)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print( f"Train Accuracy: {accuracy_score(y_resampled, xgb_model.predict(X_resampled))}")
print(f"Test Accuracy: {accuracy}")
print("Classification Report:")
print(report)
```

A RandomForestClassifier is used with Recursive Feature Elimination (RFE) to select the 15 most important features from X_resampled. RFE fits the model and identifies which features contribute most to predictions. The dataset X_resampled and X_test are then reduced to include only these selected features for further model training and testing.

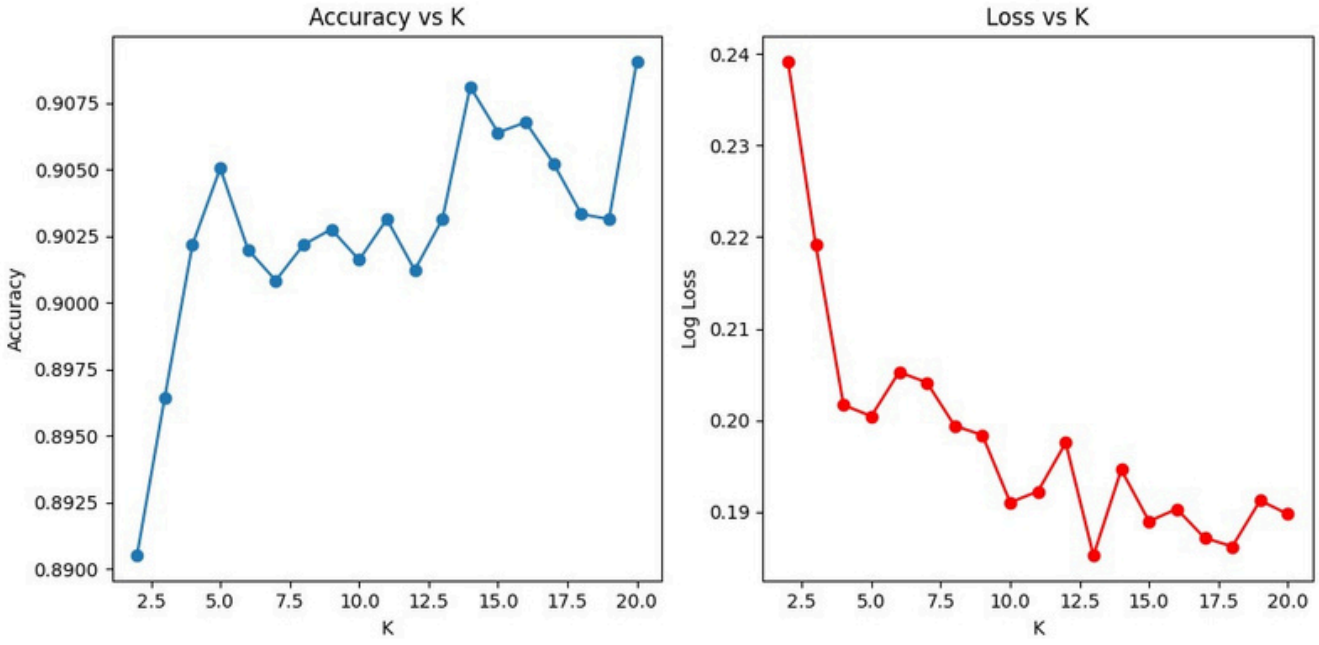
```
X_resampled = df_resampled.drop(['Name','approach_date','approach_year','approach_month', 'approach_day', 'Hazardous','Epoch Osculation'], axis=1)
y_resampled = df_resampled['Hazardous']
X_test = df_test.drop(['Name','approach_date','approach_year','approach_month', 'approach_day', 'Hazardous','Epoch Osculation'], axis=1)
y_test = df_test['Hazardous']
```

```
model = RandomForestClassifier(random_state=42)
rfe = RFE(model, n_features_to_select=15)
rfe = rfe.fit(X_resampled, y_resampled)
selected_features = X_resampled.columns[rfe.support_]

X_resampled = X_resampled[selected_features]
X_test = X_test[selected_features]
```

K-fold cross-validation is used to evaluate the performance of a machine learning model in a more reliable and generalized way by splitting the dataset into K equally-sized subsets (or "folds"). This method helps ensure that the model is not overfitting or underfitting to a particular subset of data and provides a better estimate of how well the model will perform on unseen data.

```
k_values = range(2, 11)
fold_accuracies = []
fold_losses = []
import sklearn.metrics as metrics
for k in k_values:
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    accuracies = []
    losses = []
    for train_idx, val_idx in kf.split(X_resampled):
        X_train_k, X_val_k = X_resampled[train_idx], X_resampled[val_idx]
        y_train_k, y_val_k = y_resampled[train_idx], y_resampled[val_idx]
        xgb_clf.fit(X_train_k, y_train_k, eval_set=[(X_val_k, y_val_k)], verbose=False)

        y_val_pred_prob = xgb_clf.predict_proba(X_val_k)[:, 1]
        y_val_pred = (y_val_pred_prob >= 0.25).astype(int)

        acc = accuracy_score(y_val_k, y_val_pred)
        loss = metrics.log_loss(y_val_k, y_val_pred_prob)

        accuracies.append(acc)
        losses.append(loss)

    fold_accuracies.append(np.mean(accuracies))
    fold_losses.append(np.mean(losses))
```

Where K refers to the number of folds in cross-validation, Accuracy vs K shows how model accuracy changes as the number of folds increases, reflecting how well the model generalizes across different subsets of data. Log loss vs K measures the model's confidence-based error across folds, with lower log loss indicating better-calibrated probabilities. Together, they help assess the stability and reliability of the model's performance as K increases.



We've tried two hyperparameter tuning techniques namely Bayesian Optimization and GridSearch.
GridSearch is a technique used to find the optimal hyperparameters for a machine learning model by systematically searching through a predefined set of hyperparameter combinations. It exhaustively evaluates every combination of hyperparameters using cross-validation to determine which set yields the best model performance.

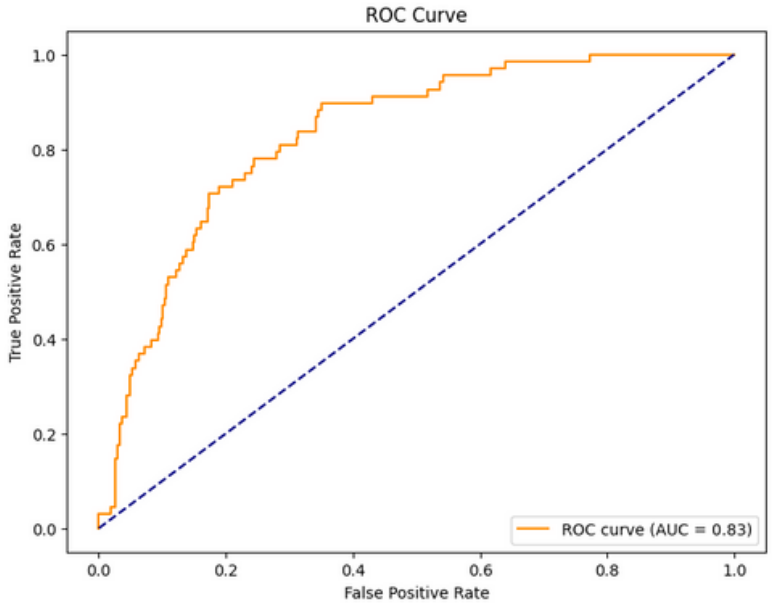
```
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'subsample': [0.6, 0.8, 1.0]
}
grid_search = GridSearchCV(estimator=xgb_clf, param_grid=param_grid,
                           scoring='f1', cv=5, verbose=1)
grid_search.fit(X_resampled, y_resampled)
best_params = grid_search.best_params_
print(f"Best Hyperparameters: {best_params}")
best_xgb_clf = grid_search.best_estimator_
best_xgb_clf.fit(X_resampled, y_resampled)
y_pred_proba = best_xgb_clf.predict_proba(X_test)[:, 1]
y_pred = (y_pred_proba > 0.25).astype(int)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)
print(f"Test Accuracy: {accuracy}")
print(f"Train Accuracy: {accuracy_score(y_resampled, best_model.predict(X_resampled))}")
print(f"Classification Report:\n{report}")
```

Bayesian Optimization is a probabilistic method used to efficiently search for optimal hyperparameters for models like XGBClassifier by modeling the performance of hyperparameters.An objective function evaluates the model with suggested hyperparameters, and cross-validation is employed to assess its performance.

```
def objective(trial):
    param = {
        'booster': 'gbtree',
        'objective': 'binary:logistic',
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3),
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'n_estimators': trial.suggest_int('n_estimators', 50, 400),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
        'gamma': trial.suggest_float('gamma', 0.0, 0.5),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
        'lambda': trial.suggest_float('lambda', 1e-8, 1.0),
        'alpha': trial.suggest_float('alpha', 1e-8, 1.0)
    }
    model = XGBClassifier(**param)
    scores = cross_val_score(model, X_resampled, y_resampled, cv=5, scoring='f1', n_jobs=-1)
    return scores.mean()
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)
print("Best trial parameters:", study.best_trial.params)
print("Best F1 score:", study.best_value)
best_params = study.best_trial.params
best_model = XGBClassifier(**best_params)
best_model.fit(X_resampled, y_resampled)
y_pred_proba = best_model.predict_proba(X_test)[:, 1]
y_pred = (y_pred_proba > 0.25).astype(int)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)
print(f"Test Accuracy: {accuracy}")
print(f"Train Accuracy: {accuracy_score(y_resampled, best_model.predict(X_resampled))}")
print(f"Classification Report:\n{report}")
```

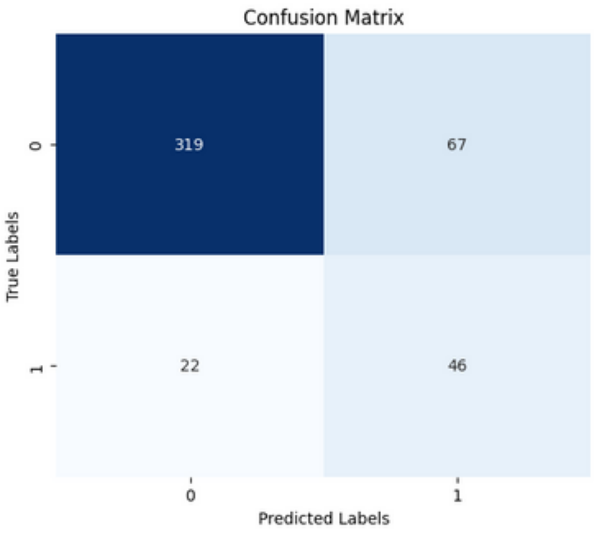
Bayesian optimization is more efficient than GridSearch because it focuses on exploring the most promising hyperparameter regions using past results, whereas GridSearch exhaustively tests all combinations, wasting time on less optimal configurations. This leads to faster convergence to the best hyperparameters.

The ROC curve (Receiver Operating Characteristic curve) is used to evaluate the performance of a binary classification model by showing the trade-off between the true positive rate (TPR) and the false positive rate (FPR) at different threshold settings. It helps visualize how well a model distinguishes between the two classes across various decision thresholds.



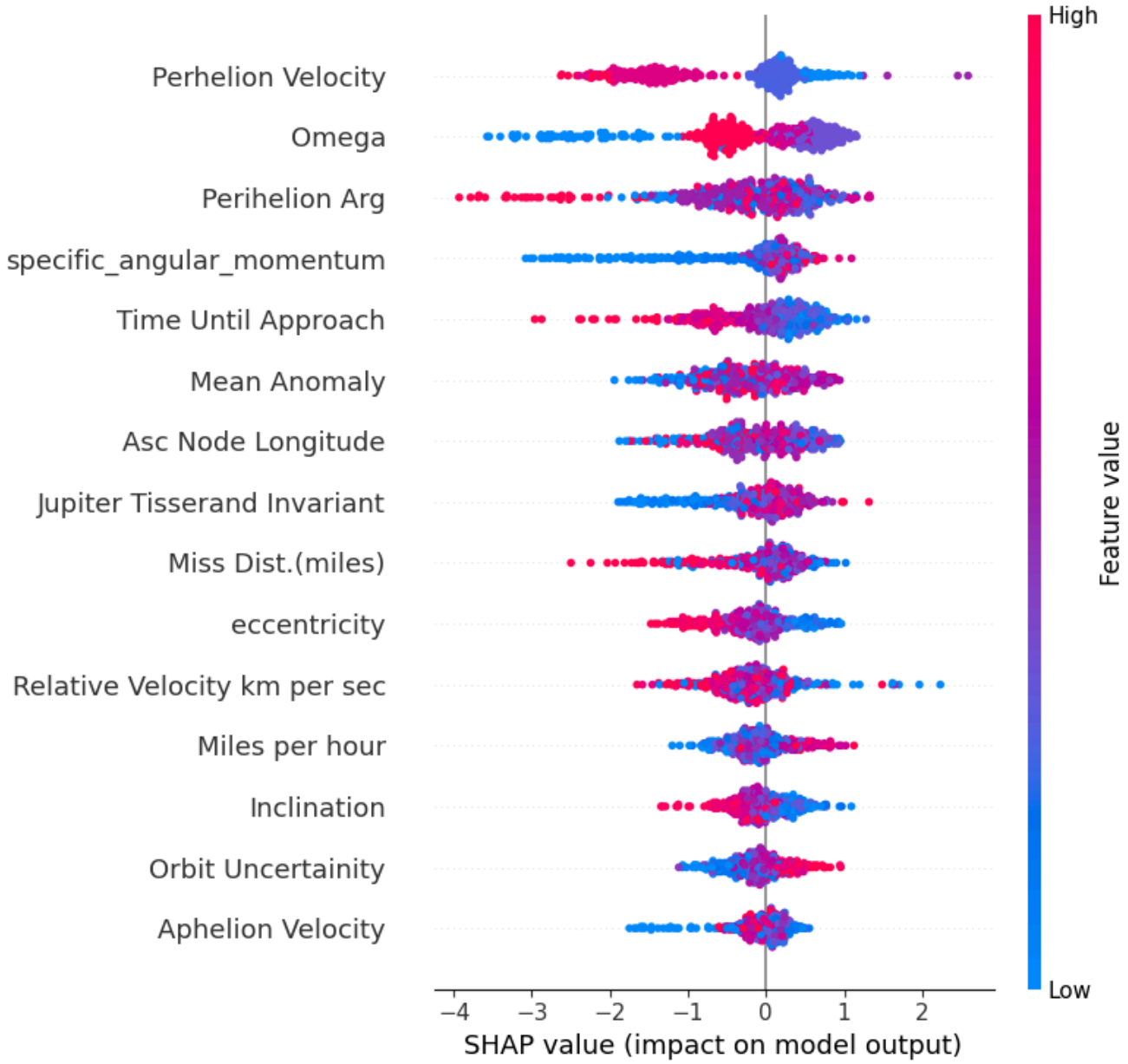
An **AUC** of **0.83** is very good for a class-imbalanced dataset as it indicates strong model discrimination between classes, despite the imbalance. AUC is less sensitive to imbalanced data, making it a reliable measure of how well the model ranks positive instances higher than negative ones.

Confusion matrix is used to evaluate the performance of a classification model by providing a detailed breakdown of the model's predictions compared to the actual outcomes. It helps identify how well the model distinguishes between different classes and provides insights into specific types of prediction errors.



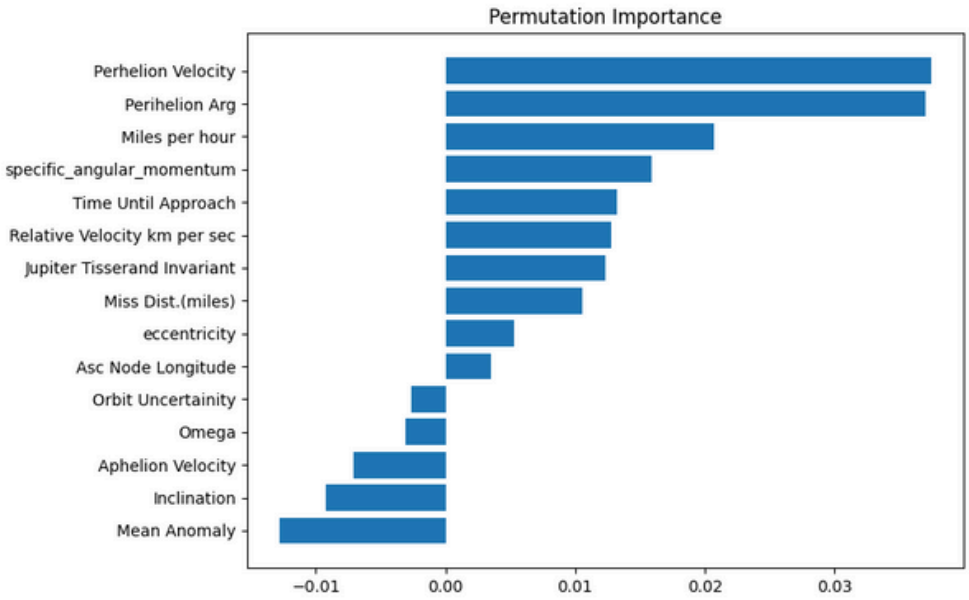
SHAP (SHapley Additive exPlanations) is used to explain the output of machine learning models by attributing the contribution of each feature to the final prediction. SHAP provides a consistent and unified measure of feature importance, which is especially useful for interpreting complex models like neural networks, gradient boosting, or ensemble methods.

```
explainer = shap.Explainer(xgb_clf)
X_test_df = pd.DataFrame(X_test,columns=selected_features)
shap_values = explainer(X_test_df)
shap.summary_plot(shap_values, X_test_df)
```



Permutation importance is used to assess the significance of individual features in a machine learning model by measuring the impact on model performance when the values of a feature are randomly shuffled. This method helps determine how much a model relies on each feature for making accurate predictions.

```
perm_importance = permutation_importance(best_xgb_clf, X_test, y_test)
sorted_idx = perm_importance.importances_mean.argsort()
plt.figure(figsize=(8, 6))
plt.barh(range(len(sorted_idx)), perm_importance.importances_mean[sorted_idx])
plt.yticks(range(len(sorted_idx)), X_test.columns[sorted_idx])
plt.title('Permutation Importance')
plt.show()
```



5. Anomaly Detection:

Anomaly detection identifies data points, events, or patterns that deviate significantly from the norm or expected behavior. It's used to detect rare or unusual occurrences, often signaling potential issues like fraud, defects, or security breaches. It can be applied to time-series data, images, or any structured/unstructured data. Techniques include statistical methods, machine learning, and deep learning models.

Isolation Forest is an ensemble learning method used primarily for anomaly detection (also known as outlier detection). It is particularly effective for high-dimensional datasets.

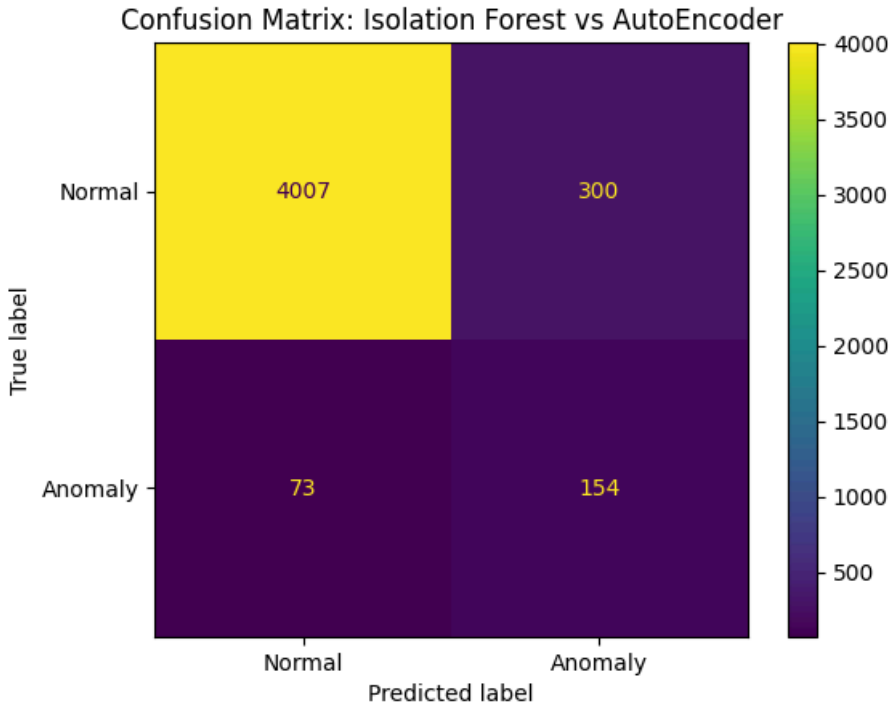
```
df_imputed.drop(columns=['Orbital Period','Orbit Uncertainity','Relative Velocity km per sec'],inplace=True)
iso_forest = IsolationForest(contamination=0.05, random_state=42)
df_imputed['iso_forest_anomaly'] = iso_forest.fit_predict(df_imputed.drop(columns=['Name', 'Hazardous']))
df_imputed['iso_forest_anomaly'] = df_imputed['iso_forest_anomaly'].apply(lambda x: 1 if x == -1 else 0)

iso_forest_anomalies = df_imputed['iso_forest_anomaly'].sum()
print(f"Number of anomalies detected by Isolation Forest: {iso_forest_anomalies}")
```

We implemented an **AutoEncoder** to compress and reconstruct normal data, training it to minimize reconstruction loss. Afterward, we passed the dataset through the model and flagged anomalies based on high reconstruction errors. Finally, we compared the anomalies detected by the AutoEncoder with those from Isolation Forest using a confusion matrix.

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_imputed.drop(columns=['Name', 'Hazardous']).values)

normal_data = df_imputed[df_imputed['Hazardous'] == 0].drop(columns=['Name', 'Hazardous']).values
normal_data_scaled = scaler.transform(normal_data)

input_layer = layers.Input(shape=(normal_data_scaled.shape[1],))
encoded = layers.Dense(32, activation='relu')(input_layer)
encoded = layers.Dense(16, activation='relu')(encoded)
decoded = layers.Dense(32, activation='relu')(encoded)
decoded = layers.Dense(normal_data_scaled.shape[1], activation='linear')(decoded)

autoencoder = keras.Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(normal_data_scaled, normal_data_scaled, epochs=10, batch_size=32, shuffle=True, validation_split=0.1)
reconstructed_scaled = autoencoder.predict(scaled_data)
mse = np.mean(np.square(scaled_data - reconstructed_scaled), axis=1)
threshold = np.percentile(mse, 90)
df_imputed['autoencoder_anomaly'] = (mse > threshold).astype(int)
autoencoder_anomalies = df_imputed['autoencoder_anomaly'].sum()
print(f"Number of anomalies detected by AutoEncoder: {autoencoder_anomalies}")
```

Finally, we constructed a Confusion Matrix for the two anomaly detection algorithms' output and we found out that **154** anomaly is detected by both the algorithm.

```
cm = confusion_matrix(df_imputed['iso_forest_anomaly'], df_imputed['autoencoder_anomaly'])
print(f"Confusion Matrix:\n{cm}")
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Normal', 'Anomaly'])
disp.plot()
plt.title('Confusion Matrix: Isolation Forest vs AutoEncoder')
plt.show()
both_flagged = np.logical_and(df_imputed['iso_forest_anomaly'] == 1, df_imputed['autoencoder_anomaly'] == 1).sum()
print(f"Number of examples flagged by both algorithms: {both_flagged}")
```



Results:

As we are dealing with a dataset with class imbalance the Accuracy score is not a great measure of classification rather, the usage of the AUC score, F1-score, Precision, and Recall gives a better idea about the model performance.

Metric	Value
Peak Accuracy	**0.91**
AUC	**0.83**
Recall (weighted Avg)	**0.81**
F1-score (weighted Avg)	**0.82**