PROBLEM STATEMENT

Astronomers are always keeping an eye on Near-Earth Objects (NEOs) — asteroids or comets that pass close to our planet and might pose a threat. Depending on their size, speed, and how near they come to Earth, some of these objects can be considered dangerous. Being able to predict which ones might be hazardous is an important step in protecting our planet.

OBJECTIVE

To develop a machine learning model that accurately predicts whether a Near-Earth Object (NEO) is hazardous based on its physical and orbital characteristics.

DATA UNDERSTANDING & ANALYSIS

```
import numpy as np  # Fundamental package for numerical operations on arrays
import pandas as pd  # Library for data manipulation and analysis using DataFrames

import matplotlib.pyplot as plt  # Plotting library for visualizing data
#%matplotlib inline  # Enables inline plotting in Jupyter notebooks

import seaborn as sns  # High-level interface for statistical data visualization

NEO_data = pd.read_csv('Group5.csv')

NEO_data.columns
```

```
Index(['neo_id', 'name', 'absolute_magnitude', 'estimated_diameter_min',
       'estimated_diameter_max', 'orbiting_body', 'relative_velocity',
       'miss_distance', 'is_hazardous'],
      dtype='object')
```

```
NEO_data.head()
```

```
    neo_id                name  absolute_magnitude  estimated_diameter_min  \
0  2162117  162117 (1998 SD15)               19.14                0.394962
1  2349507    349507 (2008 QY)               18.50                0.530341
2  2455415    455415 (2003 GA)               21.45                0.136319
3  3132126           (2002 PB)               20.63
```

```
0.198863
4  3557844         (2011 DW)              22.70
0.076658

   estimated_diameter_max orbiting_body  relative_velocity
miss_distance  \
0                0.883161         Earth       71745.401048
5.814362e+07
1                1.185878         Earth      109949.757148
5.580105e+07
2                0.304818         Earth       24865.506798
6.720689e+07
3                0.444672         Earth       78890.076805
3.039644e+07
4                0.171412         Earth       56036.519484
6.311863e+07

   is_hazardous
0         False
1          True
2         False
3         False
4         False
```

EACH ROW REPRESENTS A RECORD OF AN ORBITING BODY

neo_id: The id of the near earth object

name: The name of the near earth object (possibly includes discovery year as well)

absolute_magnitude: The brightness level of the near earth object

estimated_diameter_min & estimated_diameter_max: The Maximum and minimum estimated diameter of the near earth object, mainly in KM

orbiting_body: The celestial body that the NEO is orbiting, in this dataset its always Earth

relative_velocity: The velocity at which NEO is travelling, relative to earth

miss_distance: The distance by which the NEO will miss Earth

is_hazardous: The target feature, A boolean value which may indicate the NEO as hazardous if true or not hazardous if false

```
NEO_data.tail()

          neo_id        name  absolute_magnitude
estimated_diameter_min  \
338194  54403809  (2023 VS4)              28.580
0.005112
338195  54415298  (2023 XW5)              28.690
```

```
0.004859
338196  54454871  (2024 KJ7)              21.919
0.109839
338197  54456245   (2024 NE)              23.887
0.044377
338198  54460573  (2024 NH3)              22.951
0.068290

        estimated_diameter_max orbiting_body  relative_velocity  \
338194                0.011430         Earth       56646.985988
338195                0.010865         Earth       21130.768947
338196                0.245607         Earth       11832.041031
338197                0.099229         Earth       56198.382733
338198                0.152700         Earth       42060.357830

        miss_distance  is_hazardous
338194   6.406548e+07         False
338195   2.948883e+07         False
338196   5.346078e+07         False
338197   5.184742e+06         False
338198   7.126682e+06         False
```

WE CAN STATE THAT

is_hazardous is a binary/categorical feature having only 2 options

absolute_magnitude,estimated_diameter_min,estimated_diameter_max,relative_velocity & miss_distance are numeric features

neo_id is a unique identifying feature as it identifies each NEO uniquely

name is a text feature

```
NEO_data.shape
```

```
(338199, 9)
```

We have 338199 observations, each with 9 attributes out of which 8 are features and the last one is the target column/target feature.

```
NEO_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 338199 entries, 0 to 338198
Data columns (total 9 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   neo_id                  338199 non-null  int64
 1   name                    338199 non-null  object
 2   absolute_magnitude      338171 non-null  float64
```

```
 3   estimated_diameter_min  338171 non-null  float64
 4   estimated_diameter_max  338171 non-null  float64
 5   orbiting_body           338199 non-null  object
 6   relative_velocity       338199 non-null  float64
 7   miss_distance           338199 non-null  float64
 8   is_hazardous            338199 non-null  bool
dtypes: bool(1), float64(5), int64(1), object(2)
memory usage: 21.0+ MB
```

orbiting_body & name are of object data type

is_hazardous(target feature) is boolean

neo_id is integer

while the remaining features are all float

since data is required in numeric form

the boolean values (true/false) can be found as (1/0)

while for the object data type we can also label it using a number like 0

```
NEO_data.describe()
```

```
             neo_id  absolute_magnitude  estimated_diameter_min  \
count  3.381990e+05       338171.000000           338171.000000
mean   1.759939e+07           22.932525                0.157812
std    2.287225e+07            2.911216                0.313885
min    2.000433e+06            9.250000                0.000511
25%    3.373980e+06           20.740000                0.025384
50%    3.742127e+06           22.800000                0.073207
75%    5.405374e+07           25.100000                0.189041
max    5.446281e+07           33.580000               37.545248

       estimated_diameter_max  relative_velocity  miss_distance
count           338171.000000      338199.000000   3.381990e+05
mean                 0.352878       51060.662908   4.153535e+07
std                  0.701869       26399.238435   2.077399e+07
min                  0.001143         203.346433   6.745533e+03
25%                  0.056760       30712.031471   2.494540e+07
50%                  0.163697       47560.465474   4.332674e+07
75%                  0.422708       66673.820614   5.933961e+07
max                 83.953727      291781.106613   7.479865e+07
```

Row	Meaning
count	Number of non-null entries (all 303, so no missing values).
mean	Average value for each column.
std	Standard deviation (how spread out the data is from the mean).
min	Minimum value in the column.

Row	Meaning
25%	First quartile (25% of data is below this value).
50%	Median (middle value).
75%	Third quartile (75% of data is below this value).
max	Maximum value in the column.

```
print("Hazardous median:", NEO_data['is_hazardous'].median())
```

```
Hazardous median: 0.0
```

FALSE = NOT HAZARDOUS

TRUE = HAZARDOUS

EXAMPLE INSIGHTS

The absolute magnitude of objects ranges from 18.5 to 22.7, with an average around 20.5, suggesting varying brightness and size.

The minimum estimated diameters range from 0.07 to 0.53 , and maximum estimated diameters go up to 1.18 showing a wide size distribution.

Relative velocities range between 24,865 to 109,950 km/h, and miss distances vary greatly, from 30 million km to 67 million km.

From the median in the previous cell, its clear that more then 50% of the NEO's are not hazardous.

```
print(NEO_data['is_hazardous'].value_counts())
```

```
is_hazardous
False    295037
True      43162
Name: count, dtype: int64
```

BALANCE RATIO OF DATASET

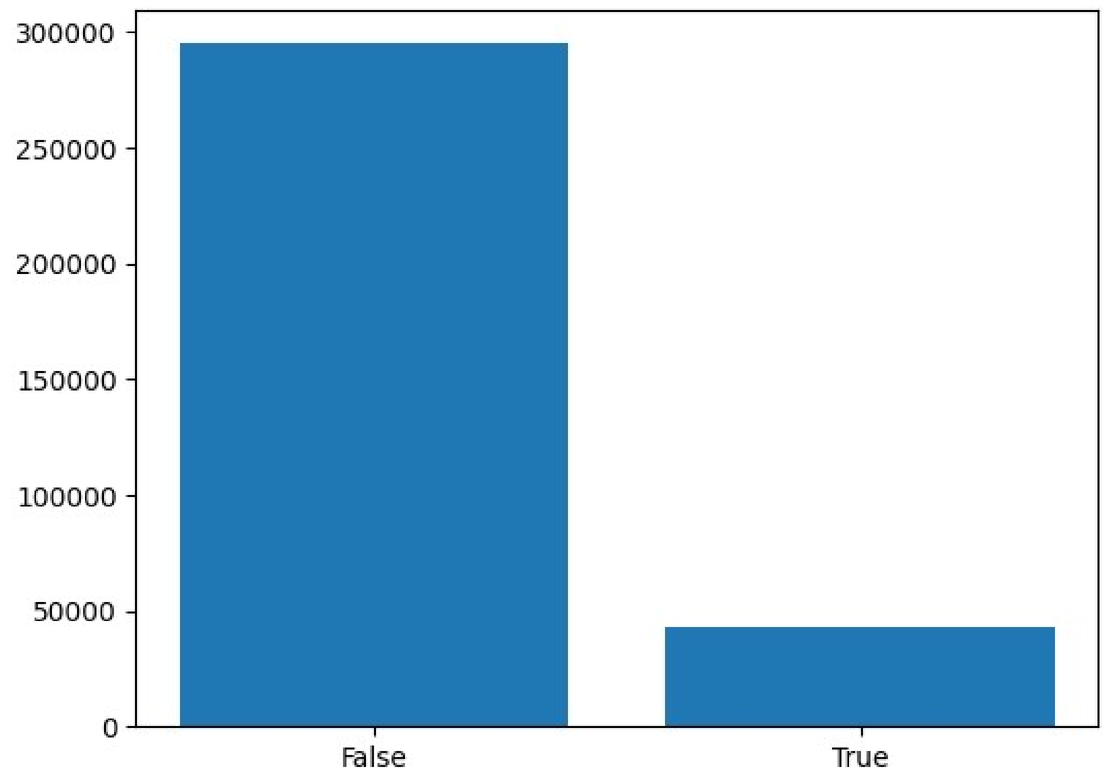
TRUE : FALSE

13% : 87%
This tells us that the dataset is severely imbalanced as its majorly dominated by the **FALSE** class
So we need to decide between oversampling & Undersampling

Deciding Between OverSampling & Undersampling

```
plt.bar(['False','True'], height=NEO_data.is_hazardous.value_counts())
```

```
<BarContainer object of 2 artists>
```



OverSampling the **TRUE** class is more ideal for this dataset as undersampling the other class could lead to the removal of some important data

```
NEO_data.pivot_table(values=
                        ['absolute_magnitude',
                         'estimated_diameter_min',
                         'estimated_diameter_max',
                         'relative_velocity',
                         'miss_distance'], index='is_hazardous',
aggfunc='mean')
```

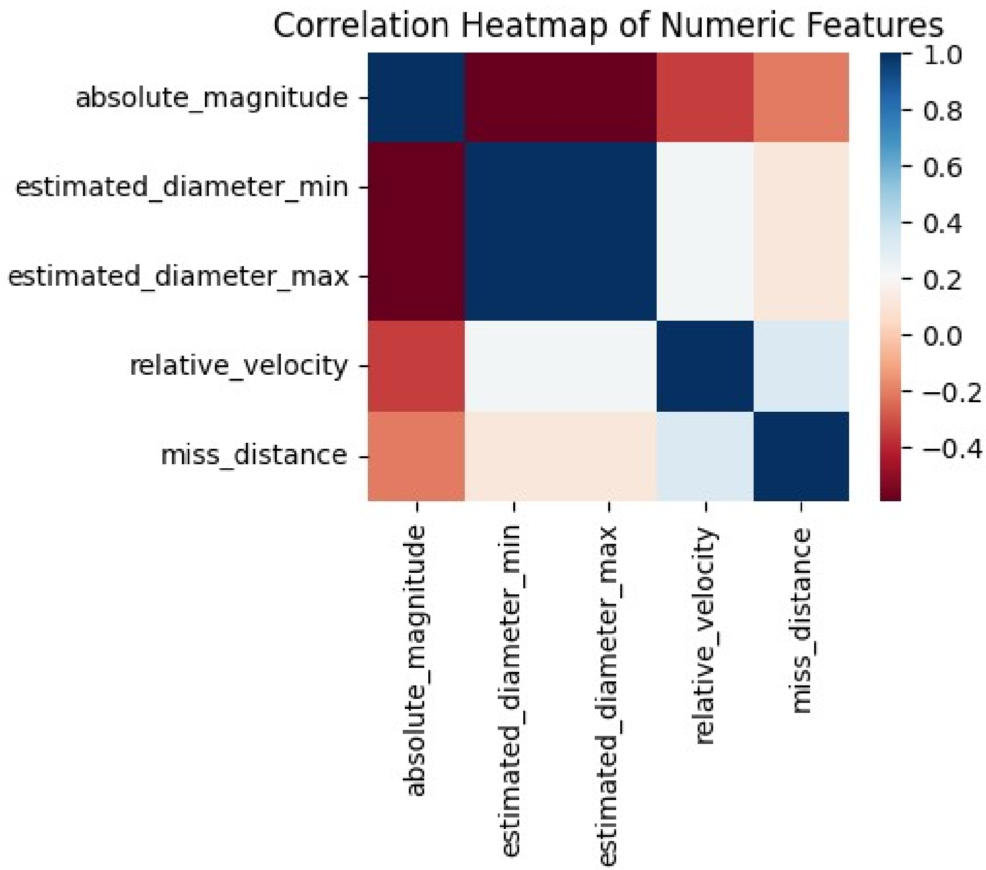
```
              absolute_magnitude  estimated_diameter_max  \
is_hazardous
False                  23.315579                0.308624
True                   20.314378                0.655353

              estimated_diameter_min  miss_distance  relative_velocity

is_hazardous

False                       0.138021   4.158731e+07       49172.265510

True                        0.293083   4.118015e+07       63968.941094
```

Feature By Feature Insights Using Mean Along With Target Feature

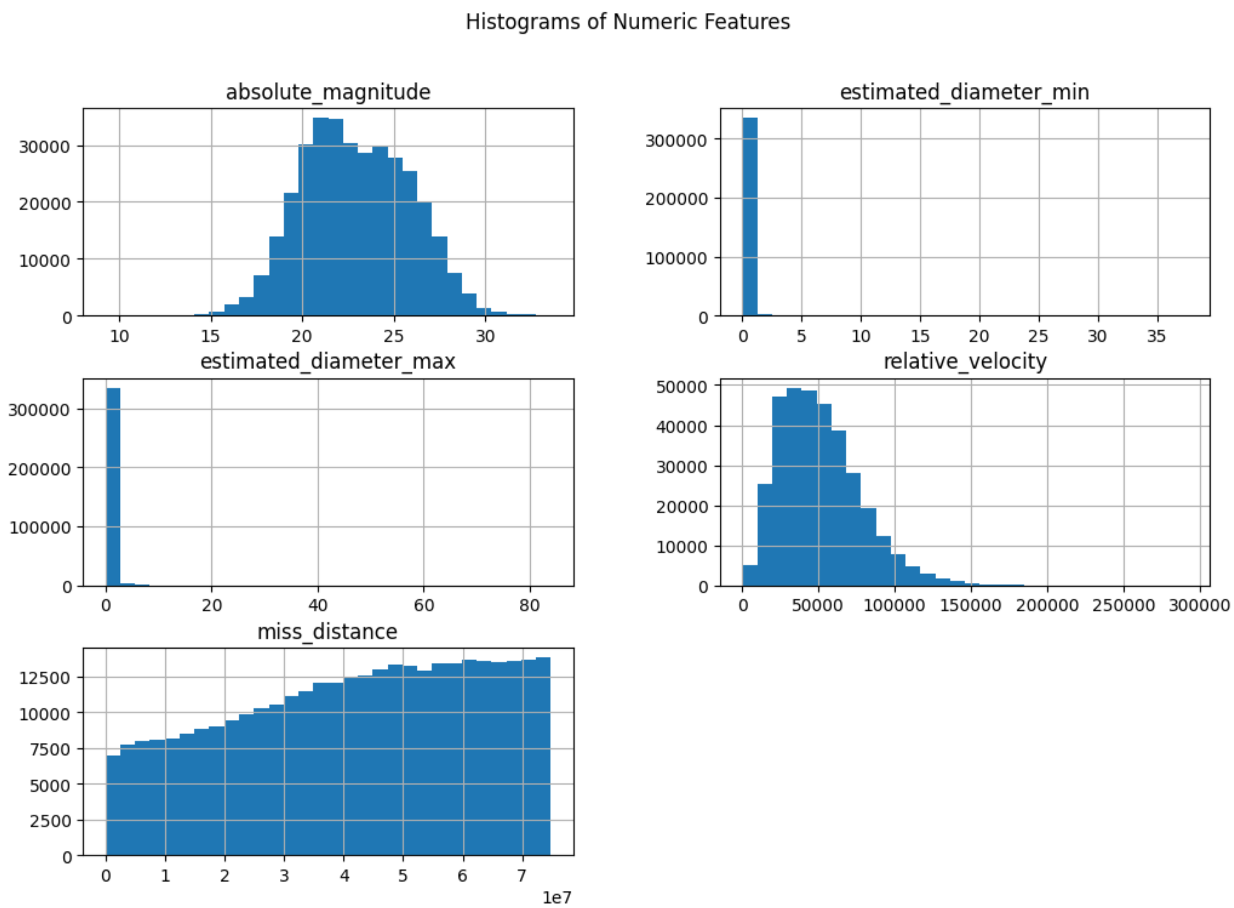
absolute_magnitude:
Mean(is_hazardous = False)= 23.3
Mean(is_hazardous = True)= 20.3
Insight: Hazardous NEOs tend to have a lower absolute magnitude,which means they are brighter and possibly larger in size.
estimated_diameter_max:
Mean(is_hazardous = False)= 0.3
Mean(is_hazardous = True)= 0.65
Insight: Hazardous NEOs are, larger in diameter, indicating that size contributes to Hazard Potential.
estimated_diameter_min:
Mean(is_hazardous = False)= 0.13
Mean(is_hazardous = True)= 0.29
Insight: Even the smallest estimated size of hazardous NEOs is roughly double that of non-hazardous ones.
miss_distance:
Mean(is_hazardous = False)= 4.15
Mean(is_hazardous = True)= 4.11
Insight: There is only a small margin of difference between the miss distance of hazardous & non hazardous objects.
relative_velocity:
Mean(is_hazardous = False)= 49172.2
Mean(is_hazardous = True)= 63968.9
Insight: Hazardous NEO's tend to travel much faster.

Features like relative_velocity & estimated_diameter(min & max) could be **important predictors** in model training.

```
#to visualize Correlation of each feature
NEO_numeric_data =
NEO_data.select_dtypes(include='number').drop(columns=['neo_id'])
plt.figure(figsize=(4, 3))
sns.heatmap(NEO_numeric_data.corr(), cmap='RdBu')
plt.title("Correlation Heatmap of Numeric Features")
plt.show()
```




```
numeric_data =
NEO_data.drop(columns=['neo_id']).select_dtypes(include='number')
numeric_data.hist(figsize=(12, 8), bins=30)
plt.suptitle('Histograms of Numeric Features')
plt.show()
```



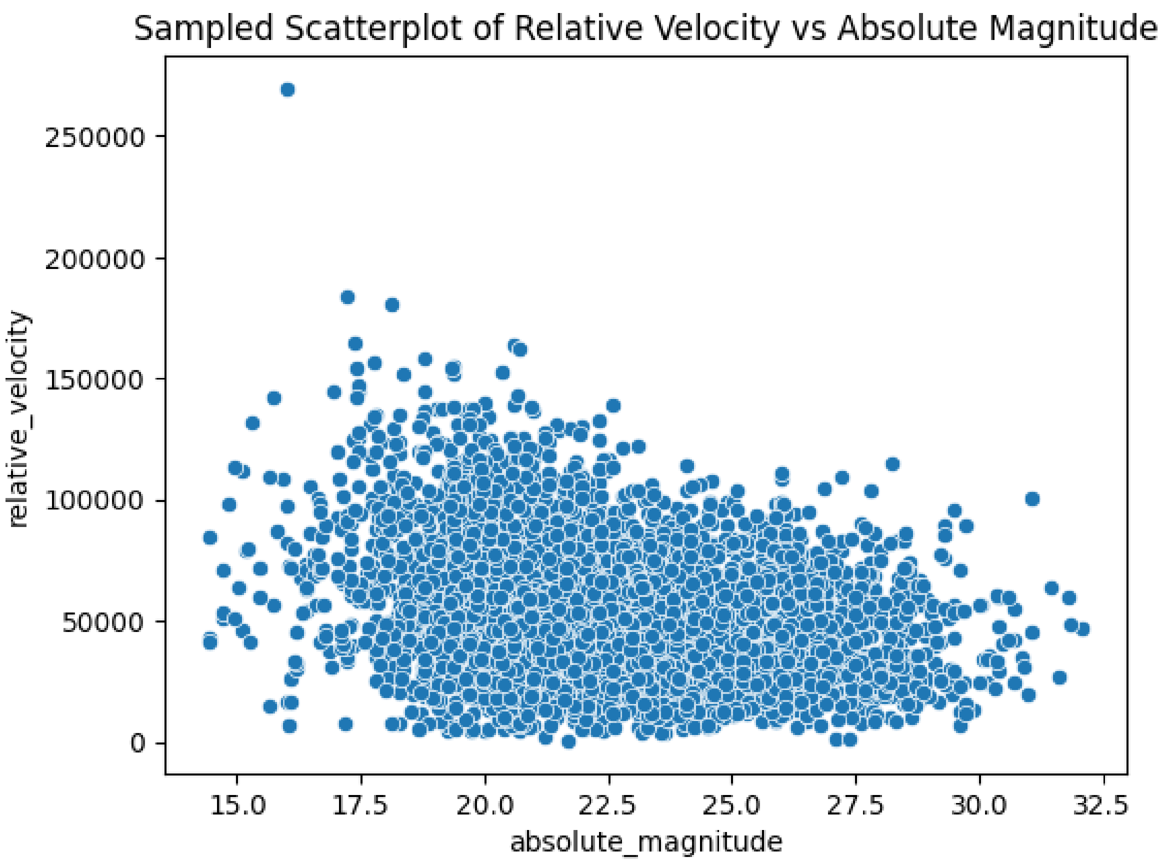

```
#Checking and handling missing values
NEO_data.isnull().sum()
```

```
neo_id                     0
name                       0
absolute_magnitude        28
estimated_diameter_min    28
estimated_diameter_max    28
orbiting_body              0
relative_velocity          0
miss_distance              0
is_hazardous               0
dtype: int64
```

the missing values in absolute_magnitude,estimated_diameter_min,estimated_diameter_max can be handled by calculating the mean or median of the respective feature since these features are numeric.

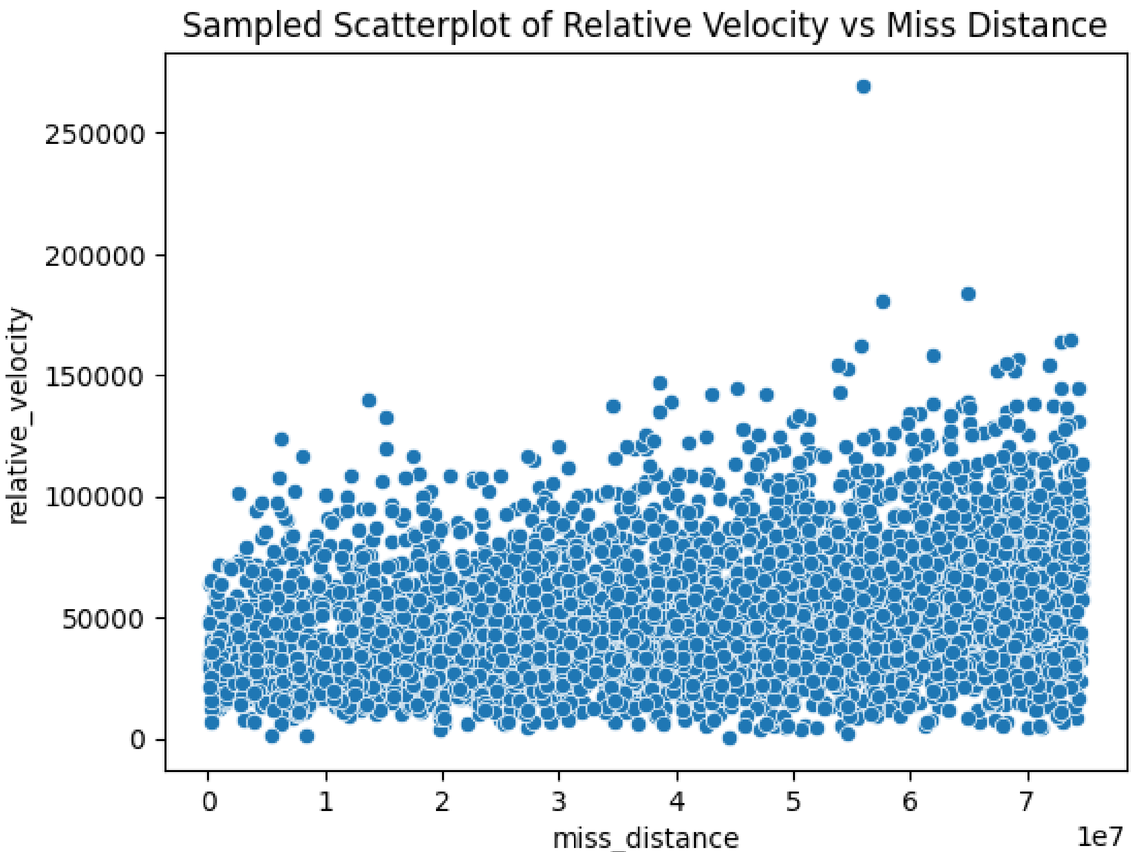
```
sample_data = NEO_data.sample(n=5000, random_state=42)
sns.scatterplot(x='absolute_magnitude', y='relative_velocity',
data=sample_data)
plt.title("Sampled Scatterplot of Relative Velocity vs Absolute
```

```
Magnitude")
plt.show()
```



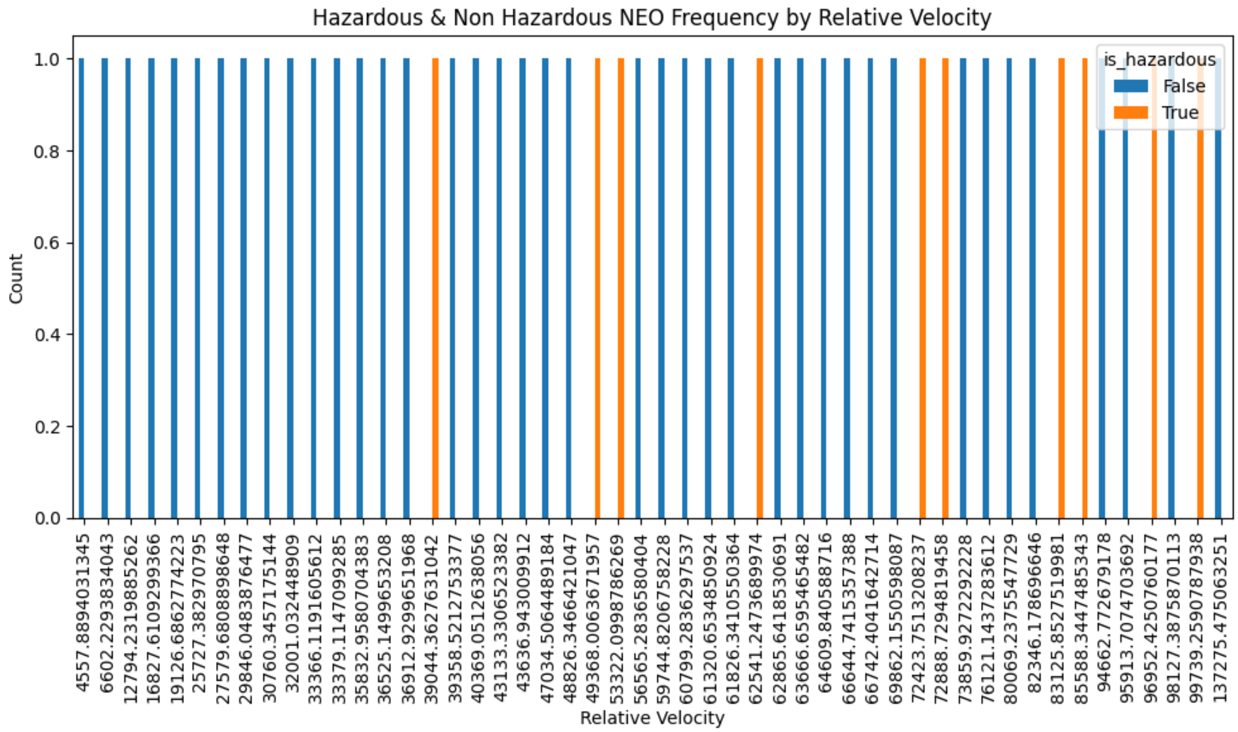
The above is a scatterplot between relative_velocity and absolute_magnitude (extracted portion of data from dataset).
This shows that there is a downward trend; a decline in relative velocity as the absolute magnitude(brightness level) of an object increases.

```
sample_data = NEO_data.sample(n=5000, random_state=42)
sns.scatterplot(x='miss_distance', y='relative_velocity',
data=sample_data)
plt.title("Sampled Scatterplot of Relative Velocity vs Miss Distance")
plt.show()
```



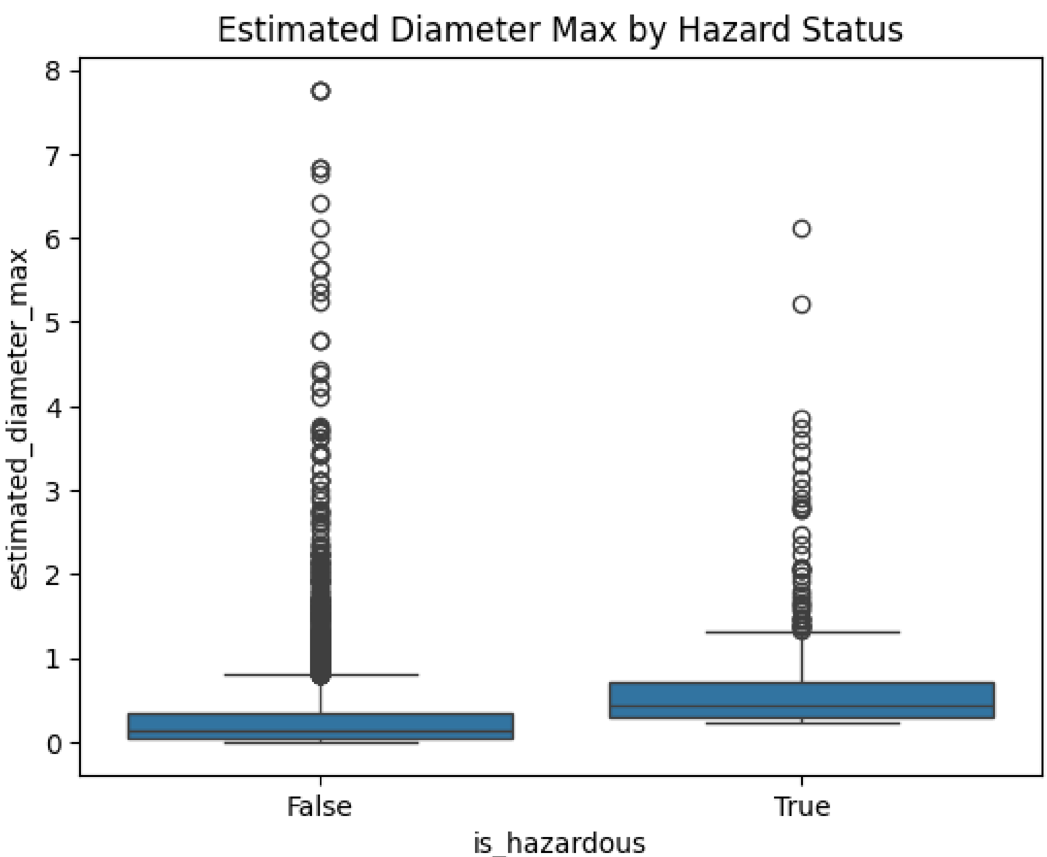
The above is a scatterplot between relative_velocity and miss_distance (extracted portion of data from dataset).
There is a very subtle/minor upward trend, the objects having relatively higher velocity may have a slightly larger miss distance

```
sample_data = NEO_data.sample(n=50, random_state=42)
pd.crosstab(sample_data.relative_velocity,
sample_data.is_hazardous).plot(kind="bar", figsize=(10,6))
plt.title('Hazardous & Non Hazardous NEO Frequency by Relative
Velocity')
plt.xlabel('Relative Velocity')
plt.xticks(rotation=90)

plt.ylabel('Count')
plt.tight_layout()
plt.show()
```



The above chart shows the distribution of hazardous & non hazardous NEO's based on their relative velocity.

```
sample_data = NEO_data.sample(n=5000, random_state=42)
sns.boxplot(x='is_hazardous', y='estimated_diameter_max',
data=sample_data)
plt.title("Estimated Diameter Max by Hazard Status")
plt.show()
```



The true box is positioned higher on the y axis in comparison to the False box. This indicates that on a general perspective hazardous NEO's are larger in size in comparison to Non hazardous NEO'S.
However, the outliers (circles) of the false box suggest that size of non hazardous NEO's varies. So while hazardous NEO'S are larger in general, there are cases where the non hazardous NEO maybe larger.

DATA PREPROCESSING - I

```
import pandas as pd
data = pd.read_csv('Group5.csv')
data.head()
```

```
    neo_id                  name  absolute_magnitude
estimated_diameter_min  \
0  2162117  162117 (1998 SD15)               19.14
0.394962
1  2349507    349507 (2008 QY)               18.50
0.530341
2  2455415    455415 (2003 GA)               21.45
0.136319
3  3132126           (2002 PB)               20.63
0.198863
```

```
4  3557844           (2011 DW)               22.70
0.076658

   estimated_diameter_max orbiting_body  relative_velocity
miss_distance  \
0                0.883161         Earth       71745.401048
5.814362e+07
1                1.185878         Earth      109949.757148
5.580105e+07
2                0.304818         Earth       24865.506798
6.720689e+07
3                0.444672         Earth       78890.076805
3.039644e+07
4                0.171412         Earth       56036.519484
6.311863e+07

   is_hazardous
0         False
1          True
2         False
3         False
4         False
```

```
missing_data = data.isna().sum()
missing_columns = missing_data[missing_data > 0]


if not missing_columns.empty:
    print("After checking, the following columns have missing values:\
n")
    print(missing_columns)
else:
    print("No columns with missing values were found.")
```

```
After checking, the following columns have missing values:

absolute_magnitude        28
estimated_diameter_min    28
estimated_diameter_max    28
dtype: int64
```

It shows we have missing values in our dataset which we have to handle

```
%pip install missingno
```

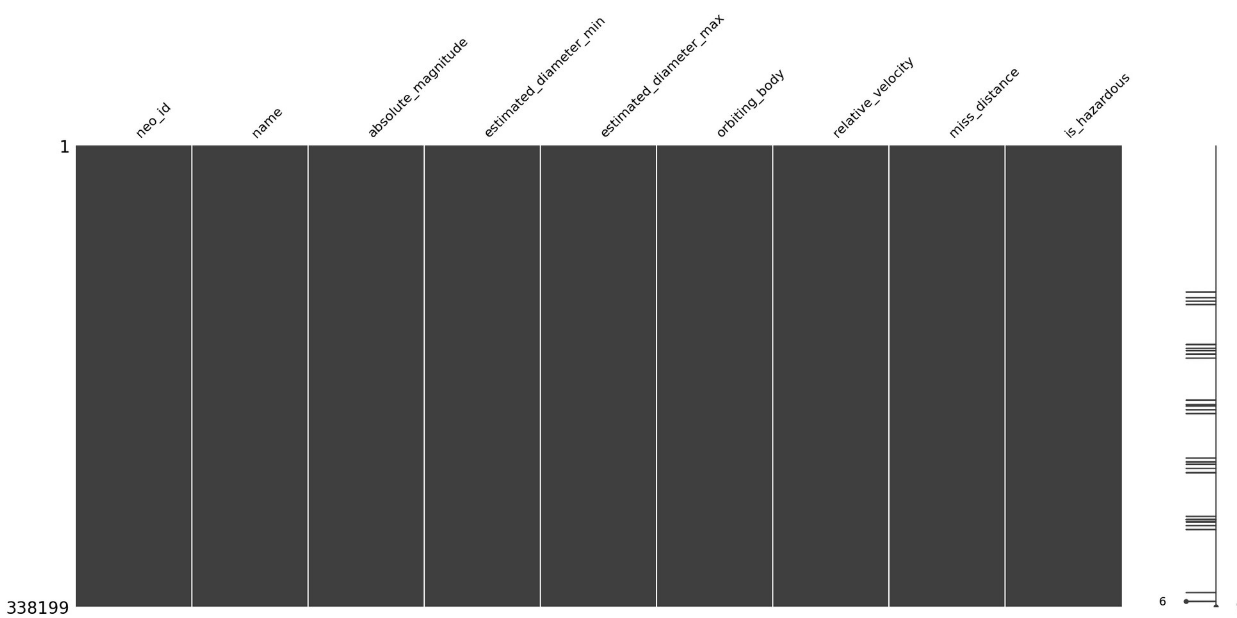
```
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: missingno in c:\users\pc\appdata\local\
packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\
localcache\local-packages\python313\site-packages (0.5.2)
```

```
Requirement already satisfied: numpy in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from missingno) (2.2.6)
Requirement already satisfied: matplotlib in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from missingno) (3.10.3)
Requirement already satisfied: scipy in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from missingno) (1.15.3)
Requirement already satisfied: seaborn in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from missingno) (0.13.2)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (1.3.2)
Requirement already satisfied: cycler>=0.10 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (4.58.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (1.4.8)
Requirement already satisfied: packaging>=20.0 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (25.0)
Requirement already satisfied: pillow>=8 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from matplotlib->missingno) (2.9.0.post0)
```

```
Requirement already satisfied: six>=1.5 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from python-dateutil>=2.7->matplotlib->missingno) (1.17.0)
Requirement already satisfied: pandas>=1.2 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from seaborn->missingno) (2.2.3)
Requirement already satisfied: pytz>=2020.1 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from pandas>=1.2->seaborn->missingno) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\pc\appdata\local\packages\pythonsoftwarefoundation.python.3.13_qbz5n2kfra8p0\localcache\local-packages\python313\site-packages (from pandas>=1.2->seaborn->missingno) (2025.2)
Note: you may need to restart the kernel to use updated packages.
```

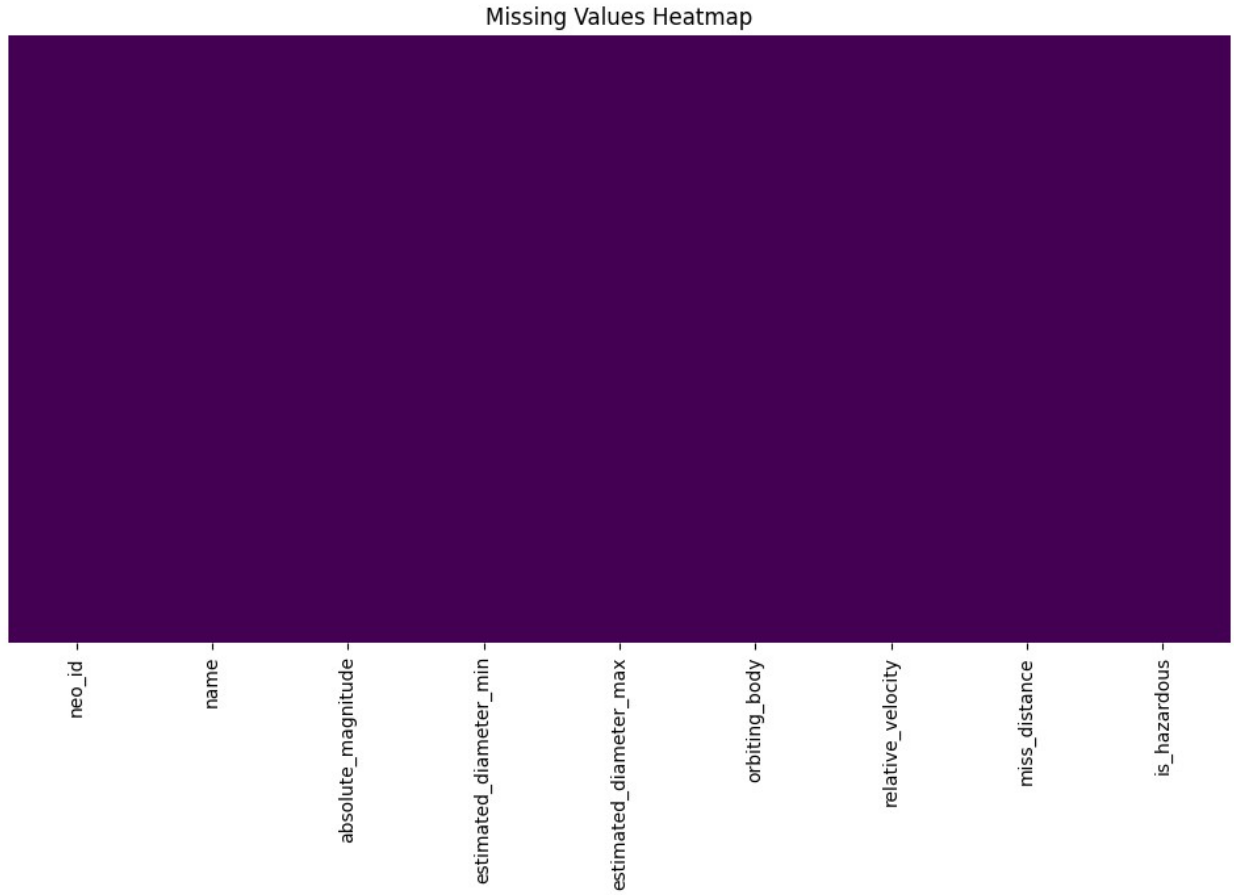
```python
import missingno as msno
msno.matrix(data)
```

```
<Axes: >
```




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

missing_data = data.isna()

plt.figure(figsize=(12, 6))
sns.heatmap(missing_data, cbar=False, cmap='viridis',
yticklabels=False)
```

```
plt.title("Missing Values Heatmap")
plt.show()
```



Why heatmaps and missing value matrix plots don’t work well on large datasets:

Libraries like Seaborn's heatmap or Missingno's matrix() or heatmap() try to plot row-wise visualizations. For 300,000+ rows:

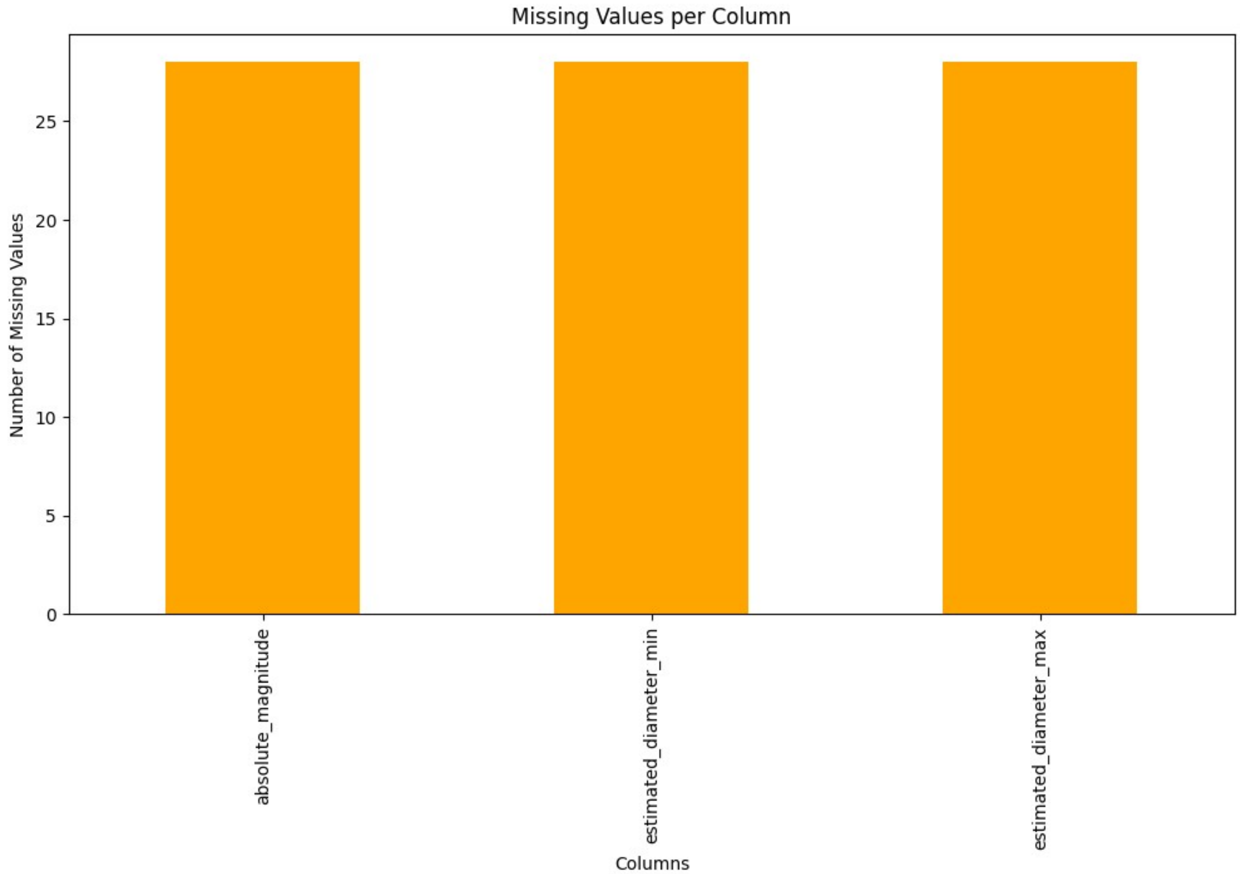
Too many pixels to render clearly.

Most missing patterns get compressed into a line, so you see a "flat" or "empty" graph.

Some libraries truncate or sample rows silently or clip colors, so missing blocks get lost visually.

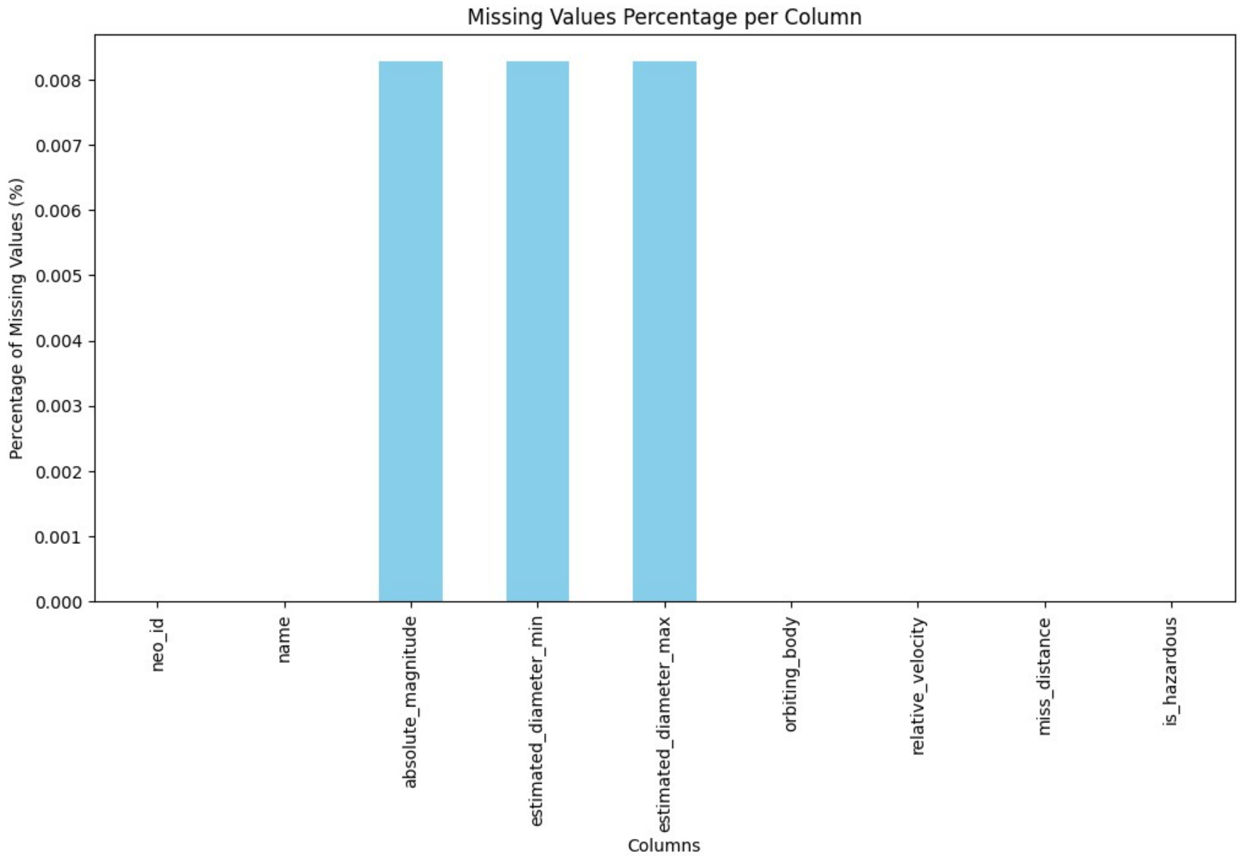
```
missing_counts = data.isna().sum()
plt.figure(figsize=(12, 6))
missing_counts[missing_counts > 0].plot(kind='bar', color='orange')
plt.title("Missing Values per Column")
```

```python
plt.ylabel("Number of Missing Values")
plt.xlabel("Columns")
plt.show()
```



Using a bar plot does confirm that we do really have missing values which we have to handle

```python
import matplotlib.pyplot as plt

# Plot percentage of missing values per column
missing_percentage = data.isnull().mean() * 100

plt.figure(figsize=(12, 6))
missing_percentage.plot(kind='bar', color='skyblue')
plt.title("Missing Values Percentage per Column")
plt.ylabel("Percentage of Missing Values (%)")
plt.xlabel("Columns")
plt.show()
```



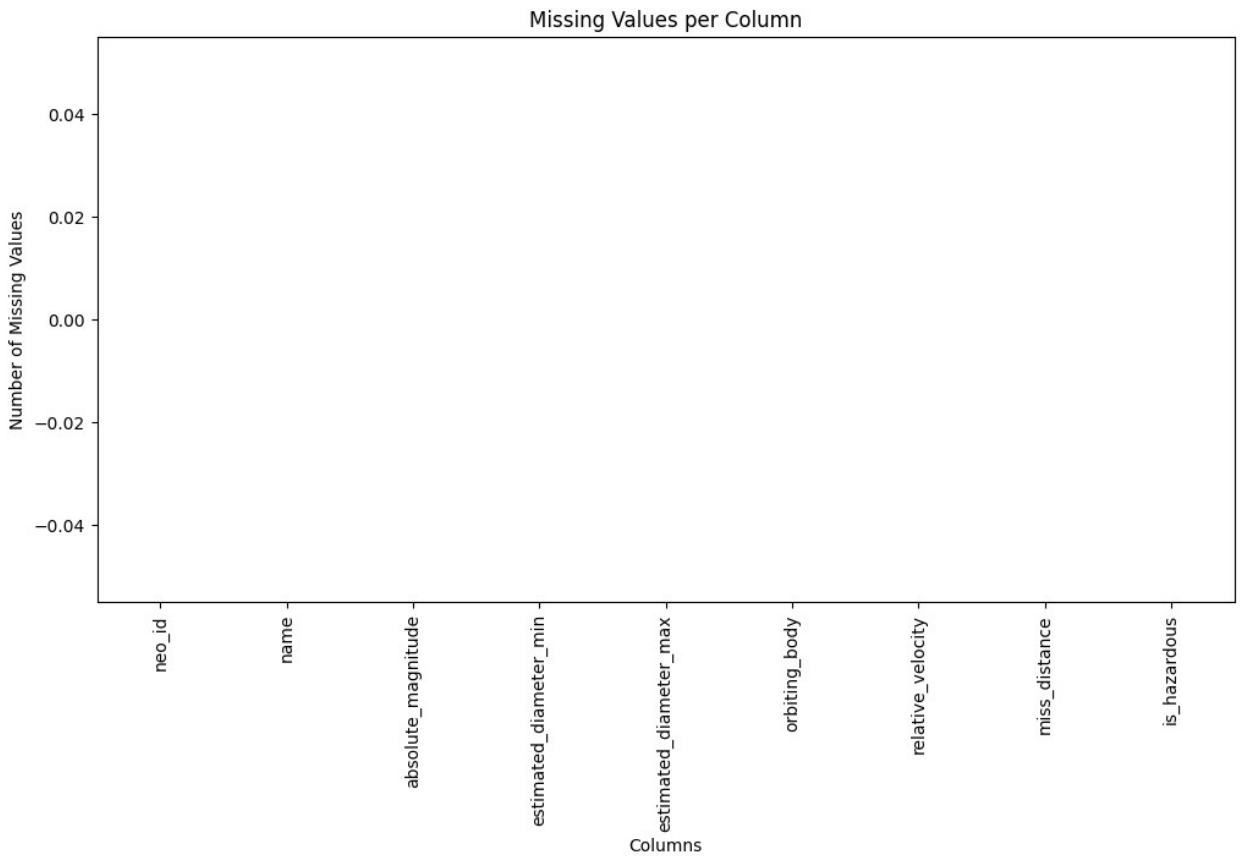
This plot tells us that we have near to 8.5% of missing values rest we have 91.5% of synthetic data.

```
data['absolute_magnitude'] =
data['absolute_magnitude'].fillna(data['absolute_magnitude'].median())
data['estimated_diameter_min'] =
data['estimated_diameter_min'].fillna(data['estimated_diameter_min'].m
edian())
data['estimated_diameter_max'] =
data['estimated_diameter_max'].fillna(data['estimated_diameter_max'].m
edian())
data.isna().sum()
```

```
neo_id                    0
name                      0
absolute_magnitude        0
estimated_diameter_min    0
estimated_diameter_max    0
orbiting_body             0
relative_velocity         0
miss_distance             0
is_hazardous              0
dtype: int64
```

We can also use the median to fill missing values. The median is less affected by outliers than the mean, making it a more robust choice in some cases. So we filled median values into the missing columns

```
missing_counts = data.isna().sum()
plt.figure(figsize=(12, 6))
missing_counts[missing_counts >= 0].plot(kind='bar', color='orange')
plt.title("Missing Values per Column")
plt.ylabel("Number of Missing Values")
plt.xlabel("Columns")
plt.show()
```



This plot shows that we have successfully terminated and handled all of the missing data.

```
from sklearn.preprocessing import LabelEncoder

label_encoders = {}
for col in ['is_hazardous']:
    le = LabelEncoder()
    data[col] = le.fit_transform(data[col].astype(str))
    label_encoders[col] = le
data.head()
```

```
    neo_id                name  absolute_magnitude
estimated_diameter_min  \
0  2162117  162117 (1998 SD15)               19.14
0.394962
1  2349507    349507 (2008 QY)               18.50
0.530341
2  2455415    455415 (2003 GA)               21.45
0.136319
3  3132126           (2002 PB)               20.63
0.198863
4  3557844           (2011 DW)               22.70
0.076658

   estimated_diameter_max orbiting_body  relative_velocity
miss_distance  \
0                0.883161         Earth       71745.401048
5.814362e+07
1                1.185878         Earth      109949.757148
5.580105e+07
2                0.304818         Earth       24865.506798
6.720689e+07
3                0.444672         Earth       78890.076805
3.039644e+07
4                0.171412         Earth       56036.519484
6.311863e+07

   is_hazardous
0             0
1             1
2             0
3             0
4             0
```

We have successfully encoded the **"is_hazardous"** column so that we can further preprocess it.

```
data['name'].value_counts().head(25000)
data.drop('name',axis=1,inplace=True)
data.head()

    neo_id  absolute_magnitude  estimated_diameter_min  \
0  2162117               19.14                0.394962
1  2349507               18.50                0.530341
2  2455415               21.45                0.136319
3  3132126               20.63                0.198863
4  3557844               22.70                0.076658

   estimated_diameter_max orbiting_body  relative_velocity
miss_distance  \
0                0.883161         Earth       71745.401048
```

```
5.814362e+07
1                1.185878         Earth      109949.757148
5.580105e+07
2                0.304818         Earth       24865.506798
6.720689e+07
3                0.444672         Earth       78890.076805
3.039644e+07
4                0.171412         Earth       56036.519484
6.311863e+07

   is_hazardous
0             0
1             1
2             0
3             0
4             0
```

The columns removed are just an identifier — like a label or ID. It doesn't contain useful numerical or categorical information that relates to the object's physical properties or its threat level.

Including it would:

Add noise (it's just a string of characters).

Confuse the model (it might try to find patterns in arbitrary strings).

Risk overfitting if names are unique or semi-unique.

```
data['neo_id'].value_counts().head(25000)
data.drop('neo_id',axis=1,inplace=True)
data.head()

   absolute_magnitude  estimated_diameter_min  estimated_diameter_max
\
0               19.14                0.394962                0.883161

1               18.50                0.530341                1.185878

2               21.45                0.136319                0.304818

3               20.63                0.198863                0.444672

4               22.70                0.076658                0.171412


  orbiting_body  relative_velocity  miss_distance  is_hazardous
0         Earth       71745.401048   5.814362e+07             0
1         Earth      109949.757148   5.580105e+07             1
2         Earth       24865.506798   6.720689e+07             0
```

```
3         Earth      78890.076805   3.039644e+07             0
4         Earth      56036.519484   6.311863e+07             0
```

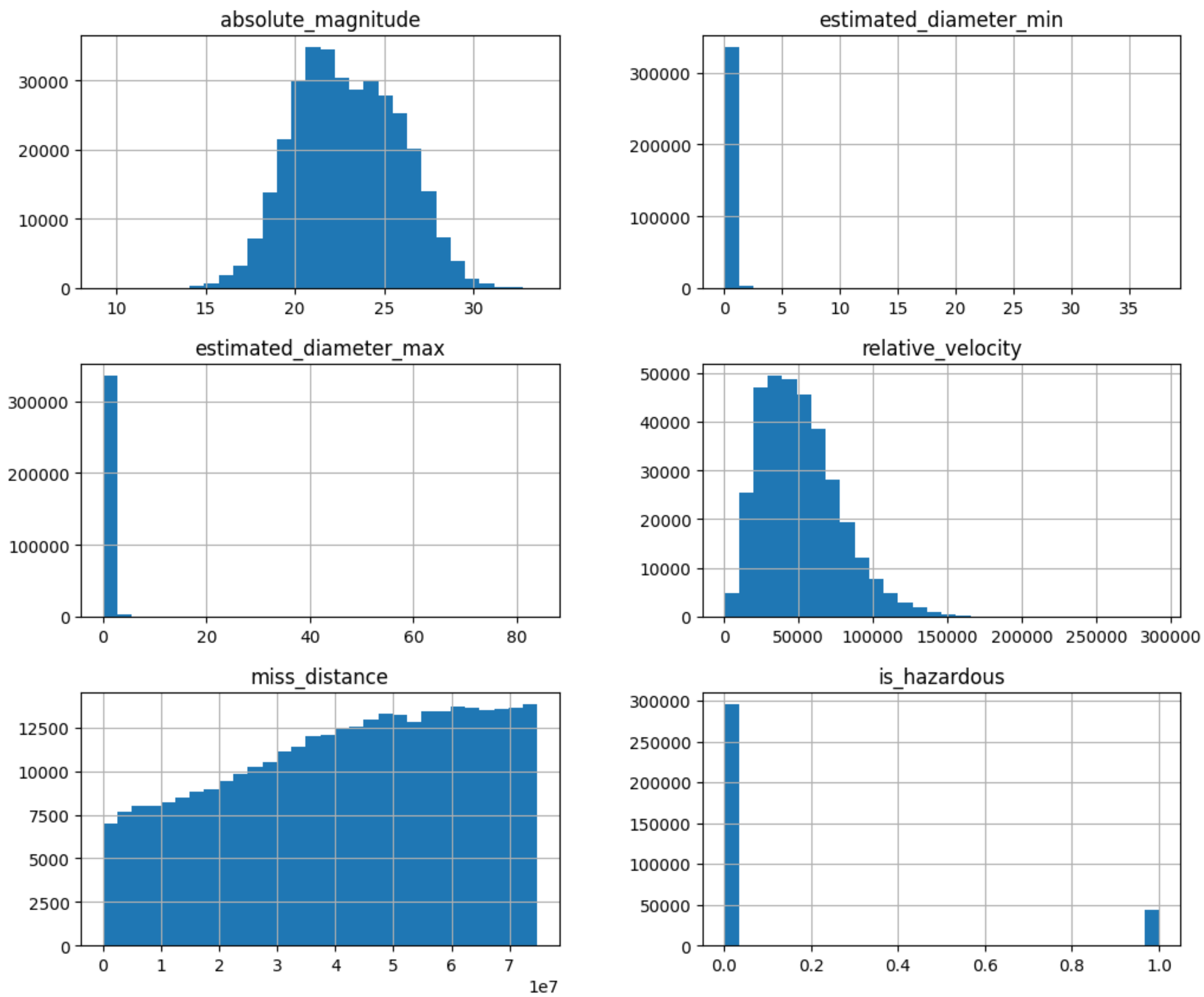
```
data['orbiting_body'].value_counts().head(25000)
data.drop('orbiting_body',axis=1,inplace=True)
data.head()
```

```
   absolute_magnitude  estimated_diameter_min  estimated_diameter_max  \
0               19.14                0.394962                0.883161

1               18.50                0.530341                1.185878

2               21.45                0.136319                0.304818

3               20.63                0.198863                0.444672

4               22.70                0.076658                0.171412


   relative_velocity  miss_distance  is_hazardous
0       71745.401048   5.814362e+07             0
1      109949.757148   5.580105e+07             1
2       24865.506798   6.720689e+07             0
3       78890.076805   3.039644e+07             0
4       56036.519484   6.311863e+07             0
```

Checking for Outliers

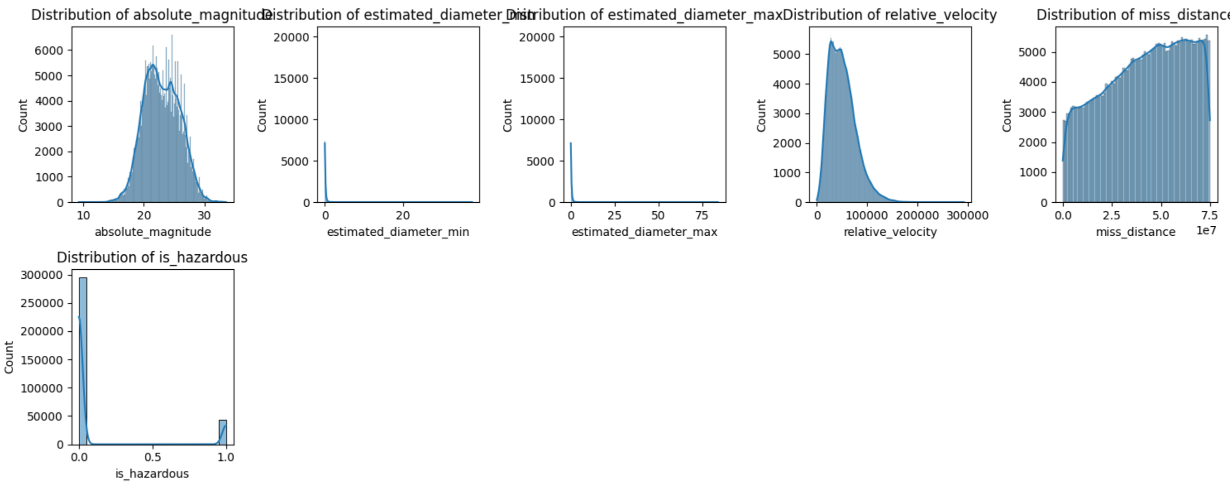
```
import matplotlib.pyplot as plt
import numpy as np

data.select_dtypes(include=[np.number]).hist(bins=30, figsize=(12,
10))
plt.suptitle("Histograms of Numeric Columns")
plt.show()
```

Histograms of Numeric Columns




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

# Number of numeric columns
num_cols = len(data.select_dtypes(include=[np.number]).columns)

# Calculate rows and columns for the grid
ncols = 5  # you can change this
nrows = int(np.ceil(num_cols / ncols))  # Ceiling to get enough rows

# Create subplots dynamically based on the number of columns
plt.figure(figsize=(ncols * 3, nrows * 3))

# Loop through the numeric columns and create a subplot for each
for i, col in
```

```
enumerate(data.select_dtypes(include=[np.number]).columns):
    plt.subplot(nrows, ncols, i + 1)  # Adjust to grid
    sns.histplot(data[col], kde=True)  # Plot histogram with KDE
    plt.title(f"Distribution of {col}")

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



Z-SCORE

```
from scipy.stats import zscore


z_scores = zscore(data.select_dtypes(include=[np.number]))


outliers = np.abs(z_scores) > 3


outliers_count = np.sum(outliers, axis=0)


outliers_count_df = pd.DataFrame(outliers_count,
index=data.select_dtypes(include=[np.number]).columns,
columns=["Outliers Count"])
print(outliers_count_df)
```

```
                        Outliers Count
absolute_magnitude                 388
estimated_diameter_min            4230
estimated_diameter_max            4230
relative_velocity                 3079
miss_distance                        0
is_hazardous                         0
```

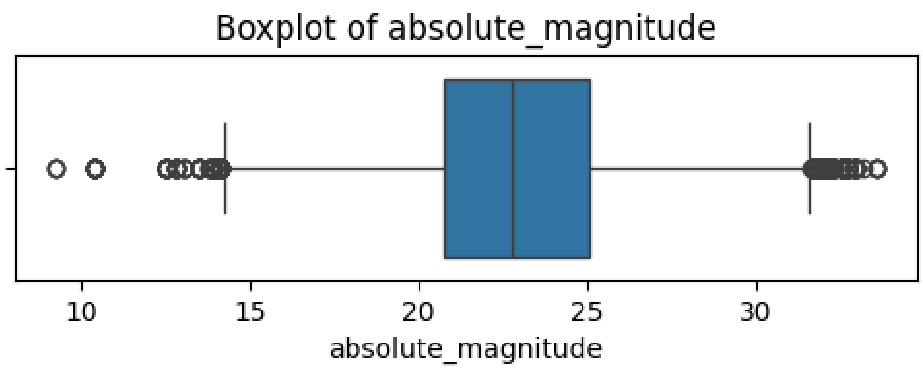
We won't be needing Z Score as our data is not normalized or doesnt have normal distribution.

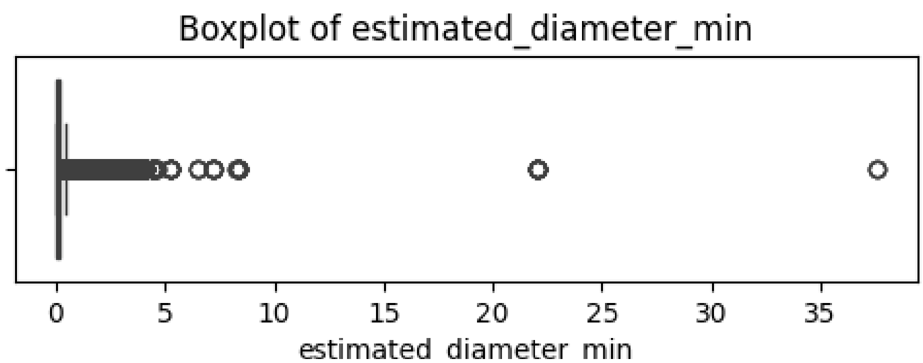
```
import pandas as pd
Q1 = data.select_dtypes(include=[np.number]).quantile(0.25)
Q3 = data.select_dtypes(include=[np.number]).quantile(0.75)
IQR = Q3 - Q1

# Identify outliers
outliers_iqr = ((data.select_dtypes(include=[np.number]) < (Q1 - 1.5 *
IQR)) |
                (data.select_dtypes(include=[np.number]) > (Q3 + 1.5 *
IQR)))
print("Lower Bound\n",Q1 - 1.5 * IQR)
print("Upper Bound\n",Q3 + 1.5 * IQR)

# Count outliers in each column
outliers_count_iqr = outliers_iqr.sum()

# Show outliers count for each column
outliers_count_iqr_df = pd.DataFrame(outliers_count_iqr,
index=data.select_dtypes(include=[np.number]).columns,
columns=["Outliers Count"])
print(outliers_count_iqr_df)
```

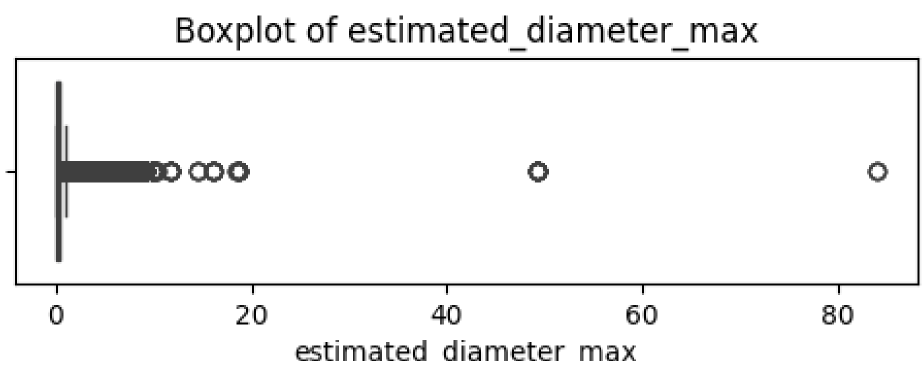
```
Lower Bound
 absolute_magnitude        1.420000e+01
estimated_diameter_min   -2.201016e-01
estimated_diameter_max   -4.921621e-01
relative_velocity        -2.323065e+04
miss_distance            -2.664591e+07
is_hazardous              0.000000e+00
dtype: float64
Upper Bound
 absolute_magnitude        3.164000e+01
estimated_diameter_min    4.345258e-01
estimated_diameter_max    9.716293e-01
relative_velocity         1.206165e+05
miss_distance             1.109309e+08
is_hazardous              0.000000e+00
dtype: float64
                        Outliers Count
absolute_magnitude                 389
estimated_diameter_min           26166
estimated_diameter_max           26166
relative_velocity                 5449
miss_distance                        0
is_hazardous                     43162
```

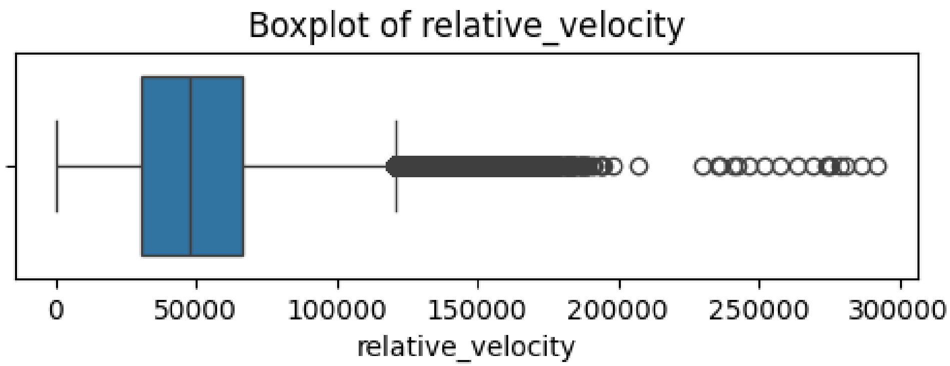
```
import seaborn as sns
import matplotlib.pyplot as plt
```

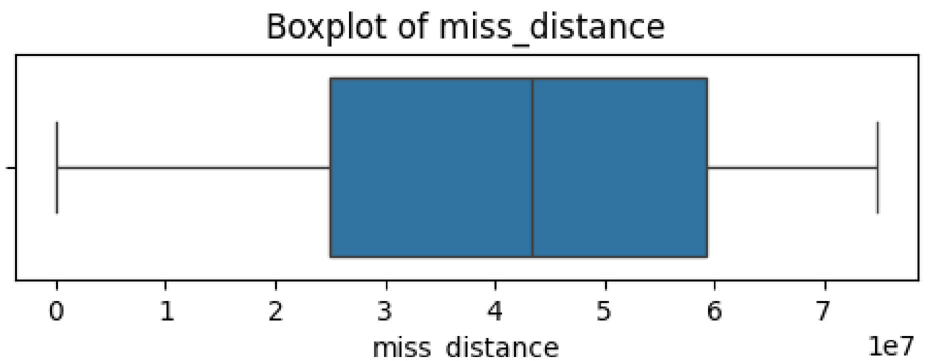
```
cols_to_check = ['absolute_magnitude', 'estimated_diameter_min',
'estimated_diameter_max',
                 'relative_velocity', 'miss_distance']  # Skip
is_hazardous

for col in cols_to_check:
    plt.figure(figsize=(6, 1.5))
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')
    plt.show()
```











neo_id and miss_distance have very large value ranges compared to the rest.

All other columns (like absolute_magnitude, estimated_diameter_min/max, relative_velocity, etc.) are compressed near 0, making their box plots barely visible.

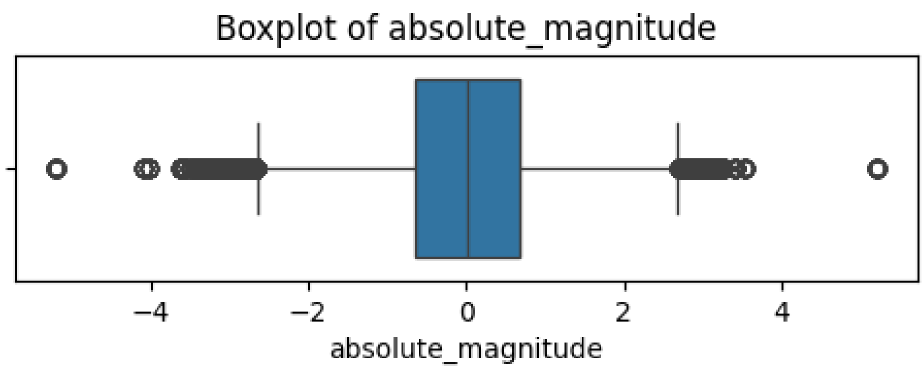
is_hazardous is a binary column (0 or 1) and doesn't need outlier detection.

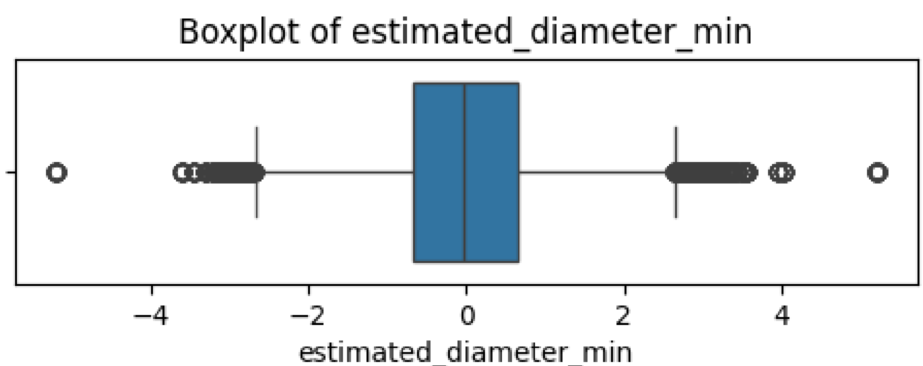
Due to the vast difference in scale, smaller-valued columns are getting squashed and visually lost. This is a scaling issue, not a data issue.

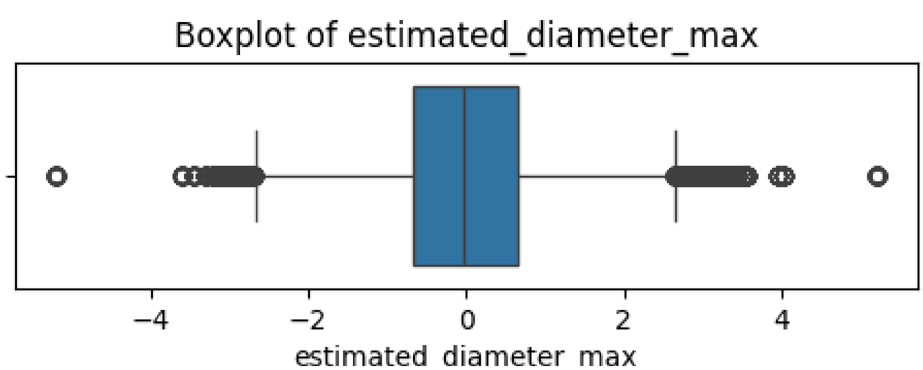
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler, QuantileTransformer

scalers = {
    "StandardScaler": StandardScaler(),
    "MinMaxScaler": MinMaxScaler(),
    "RobustScaler": RobustScaler(),
    "QuantileTransformer":
QuantileTransformer(output_distribution='normal')
}

from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler, QuantileTransformer,Normalizer
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

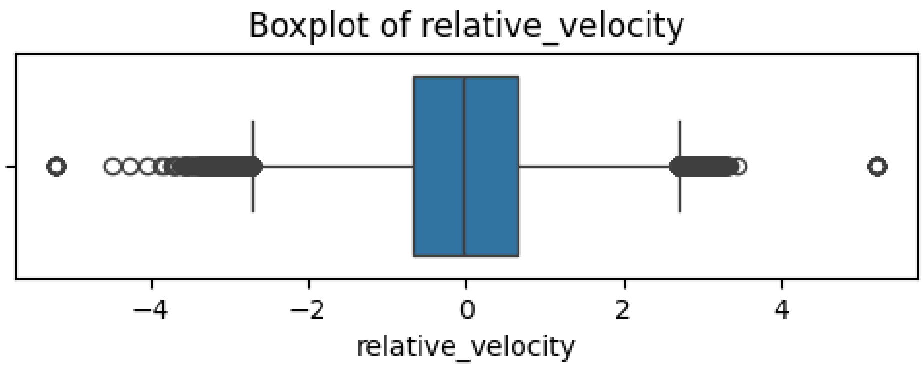
```
# Automatically identify numeric feature columns (excluding target)
numeric_features =
data.select_dtypes(include=[np.number]).drop(columns=['is_hazardous'])
.columns.tolist()

# Scalers to apply
scalers = {

    "QuantileTransformer":
QuantileTransformer(output_distribution='normal')
}

# Dictionary to hold scaled DataFrames
scaled_data = {}
for name, scaler in scalers.items():
    scaled = scaler.fit_transform(data[numeric_features])
    scaled_data[name] = pd.DataFrame(scaled, columns=numeric_features)

cols= ['absolute_magnitude', 'estimated_diameter_min',
'estimated_diameter_max',
                 'relative_velocity', 'miss_distance']
scaled_data_transformer=pd.DataFrame(scaled,columns=cols)
scaled_data_transformer.head()
# (Optional) View one of the scaled datasets

#print(scaled_data["QuantileTransformer"].head())
#print(scaled_data)

cols_to_check = ['absolute_magnitude', 'estimated_diameter_min',
'estimated_diameter_max',
                 'relative_velocity', 'miss_distance']  # Skip
is_hazardous

for col in cols_to_check:
    plt.figure(figsize=(6, 1.5))
    sns.boxplot(x=scaled_data_transformer[col])
    plt.title(f'Boxplot of {col}')
    plt.show()

scaled_data_transformer.select_dtypes(include=[np.number]).hist(bins=3
0, figsize=(12, 10))
plt.suptitle("Histograms of Numeric Columns")
plt.show()

from scipy.stats import zscore


z_scores =
zscore(scaled_data_transformer.select_dtypes(include=[np.number]))
```

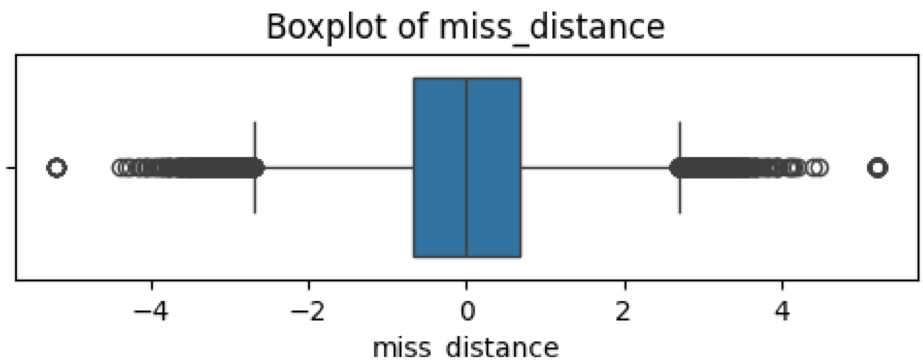
```
outliers = np.abs(z_scores) > 3


outliers_count = np.sum(outliers, axis=0)


outliers_count_df = pd.DataFrame(outliers_count,
index=scaled_data_transformer.select_dtypes(include=[np.number]).colum
ns, columns=["Outliers Count"])
print(outliers_count_df)
```

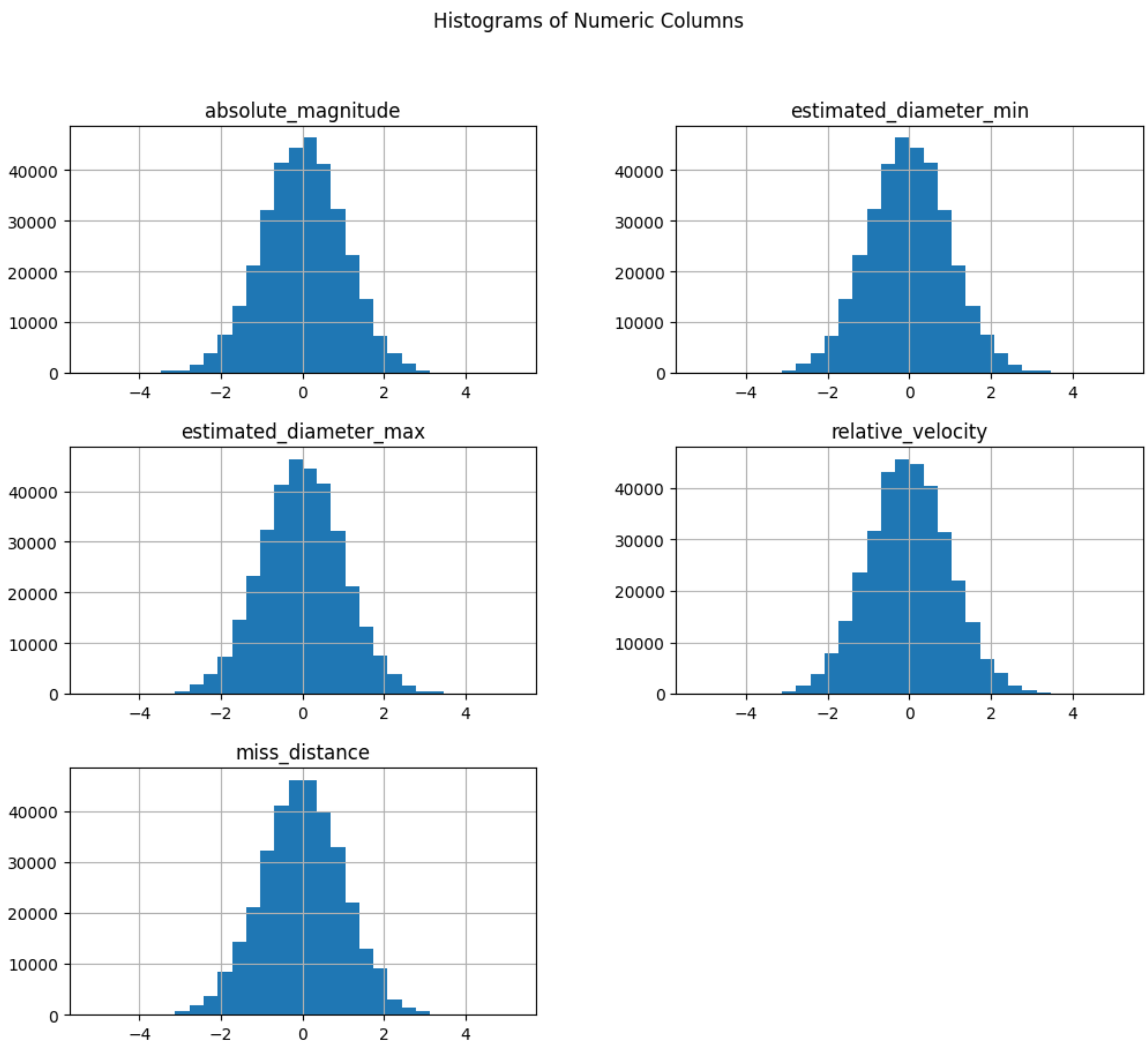







Boxplot of relative_velocity
−4
−2
0
2
4
relative_velocity

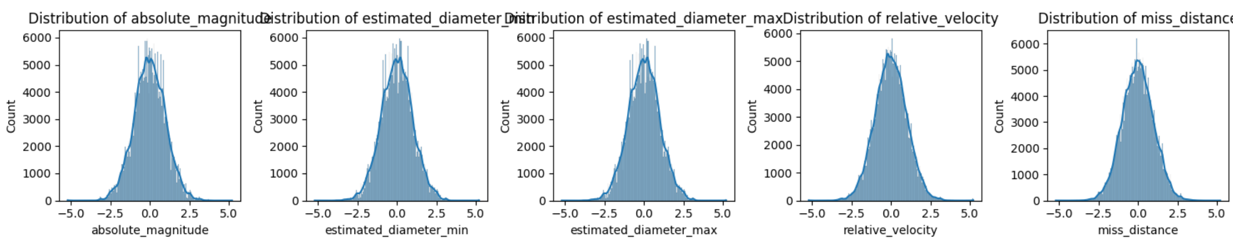

Boxplot of miss_distance
−4
−2
0
2
4
miss_distance



	Outliers Count
absolute_magnitude	1038
estimated_diameter_min	1035
estimated_diameter_max	1035
relative_velocity	947
miss_distance	754

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Number of numeric columns
num_cols =
len(scaled_data_transformer.select_dtypes(include=[np.number]).columns
)
```

```
# Calculate rows and columns for the grid
ncols = 5  # you can change this
nrows = int(np.ceil(num_cols / ncols))  # Ceiling to get enough rows

# Create subplots dynamically based on the number of columns
plt.figure(figsize=(ncols * 3, nrows * 3))

# Loop through the numeric columns and create a subplot for each
for i, col in
enumerate(scaled_data_transformer.select_dtypes(include=[np.number]).c
olumns):
    plt.subplot(nrows, ncols, i + 1)  # Adjust to grid
    sns.histplot(scaled_data_transformer[col], kde=True)  # Plot
histogram with KDE
    plt.title(f"Distribution of {col}")

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



In our dataset, the numerical features are **not Gaussian-distributed** and contain significant outliers, as observed from the box plots and distribution analysis. To ensure effective preprocessing before feeding the data into a supervised machine learning model, it was crucial to choose scalers that are robust to outliers and can handle non-normal distributions.

1. **RobustScaler**: We used RobustScaler because it scales features using the Interquartile Range (IQR) instead of the mean and standard deviation. This makes it resistant to outliers, preventing them from heavily influencing the scale of the features. It is particularly suitable when the dataset includes extreme values that could distort the behavior of other scalers like StandardScaler or MinMaxScaler.

2. **QuantileTransformer** We also used the QuantileTransformer with output_distribution='normal' to transform the features into a Gaussian-like distribution. This is beneficial for many machine learning algorithms (e.g., logistic regression, SVMs), which perform better when the input data follows a normal distribution. Additionally, the transformation is non-linear and reduces the effect of outliers by spreading out the most frequent values.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=5)
pca_components = pca.fit_transform(data.iloc[:, :-1])
```

```
pca_df = pd.DataFrame(pca_components, columns=[f"PCA_{i}" for i in
range(1, 6)])
pca_df.head()
# Access PCA component weights
loadings = pd.DataFrame(
    pca.components_,
    columns=data.columns[:-1],  # Exclude 'target'
    index=[f"PCA_{i}" for i in range(1, 6)]
)

# Show which features contribute most to each component
loadings.T.sort_values(by="PCA_1", ascending=False).head()
```

```
                              PCA_1     PCA_2         PCA_3
PCA_4  \
miss_distance          9.999999e-01 -0.000410  1.601413e-08
9.792148e-10
relative_velocity      4.099388e-04  1.000000  3.458718e-05 -
4.363937e-07
estimated_diameter_max  3.766041e-09  0.000006 -1.485779e-01
8.968485e-01
estimated_diameter_min  1.684224e-09  0.000002 -6.652789e-02
4.112949e-01
absolute_magnitude    -2.992028e-08 -0.000034  9.866604e-01
1.627859e-01

                              PCA_5
miss_distance         -7.932544e-12
relative_velocity      6.652790e-09
estimated_diameter_max -4.166382e-01
estimated_diameter_min  9.090712e-01
absolute_magnitude    -1.443886e-03
```

Due to my Domain Knowledge and PCAS suggestion we got these 6 columns as important **'absolute_magnitude', 'estimated_diameter_min', 'estimated_diameter_max', 'relative_velocity', 'miss_distance'**

```
from sklearn.feature_selection import SelectKBest, f_classif

selector = SelectKBest(score_func=f_classif, k=5)
X_new = selector.fit_transform(data.iloc[:, :-1],
data['is_hazardous'])

selected_features = selector.get_support(indices=True)
selected_df = data.iloc[:, selected_features]
selected_df.head()
```

```
   absolute_magnitude  estimated_diameter_min  estimated_diameter_max
\
0               19.14                0.394962                0.883161
```

```
1                 18.50                  0.530341                  1.185878

2                 21.45                  0.136319                  0.304818

3                 20.63                  0.198863                  0.444672

4                 22.70                  0.076658                  0.171412


   relative_velocity  miss_distance
0       71745.401048   5.814362e+07
1      109949.757148   5.580105e+07
2       24865.506798   6.720689e+07
3       78890.076805   3.039644e+07
4       56036.519484   6.311863e+07
```

Again selectk best and class_if showed us that the same columns are important.

```
from sklearn.utils import resample
from collections import Counter
from sklearn.model_selection import train_test_split
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import LabelEncoder

label_encoders = {}
for col in ['is_hazardous']:
    le = LabelEncoder()
    data[col] = le.fit_transform(data[col].astype(str))
    label_encoders[col] = le


cols = ['absolute_magnitude', 'estimated_diameter_min',
'estimated_diameter_max',
                 'relative_velocity', 'miss_distance']
# Separate features and target

X = data.drop(columns=cols)
y = data["is_hazardous"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
random_state=42)

# Separate the minority and majority classes
X_majority = X_train[y_train == 0]  # Assuming 0 is the majority class
X_minority = X_train[y_train == 1]  # Assuming 1 is the minority class
```

```
y_majority = y_train[y_train == 0]
y_minority = y_train[y_train == 1]


X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    stratify=y,
                                                    random_state=42)

# Step 2: Apply SMOTE to training data only
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train,
y_train)

# Now you can train your model using X_train_balanced and
y_train_balanced


# Print class distribution
print("Before SMOTE:", Counter(y_train))
print("After SMOTE:", Counter(y_train_balanced))
```

```
Before SMOTE: Counter({0: 206526, 1: 30213})
After SMOTE: Counter({0: 206526, 1: 206526})
```

Over here we cured the issue of UNDERSAMPLING by using SMOTE. we also changed data into trained and test sets.

Feature Engineering

```
import pandas as pd
import numpy as np

NEO_data = pd.read_csv('Group5.csv')

# Drop unnecessary identifier columns
NEO_data.drop(columns=["name", "neo_reference_id", "orbiting_body"],
inplace=True, errors='ignore')

# Add engineered features
NEO_data['estimated_diameter_avg'] =
(NEO_data['estimated_diameter_min'] +
NEO_data['estimated_diameter_max']) / 2
NEO_data['diameter_range'] = NEO_data['estimated_diameter_max'] -
NEO_data['estimated_diameter_min']
NEO_data['velocity_distance_ratio'] = NEO_data['relative_velocity'] /
NEO_data['miss_distance']
```

```python
# Drop original diameter columns
NEO_data.drop(columns=["estimated_diameter_min",
"estimated_diameter_max"], inplace=True)
```

Columns like name,neo_id,oribiting_body(only earth) are not useful for preprocessing as these are just identifiers If our dataset had multiple orbiting bodies then we could keep that feature for further use

New feature estimated_diameter_avg formed by combining 2 featured estimated diameter minimum & maximum as these 2 features are exactly of the same domain so its better to combine them into one feature which would accommodate the properties of both while being a more useful feature

New featue diameter_range formed by utilizing the same 2 features mentioned above as having separate values for max and minimum diamater could potentially lead to uncertainity while having a generic range would be much more useful

estimated diameter minimum & maximum were now dropped

New feature velocity_distance_ratio formed by utlizing miss_distance & relative_velocity as it would be more suitable for risk assessment

PREPROCESSING - II

Preprocessing on new data now

```python
NEO_data.head()
from sklearn.preprocessing import LabelEncoder

label_encoders = {}

for col in ['is_hazardous']:
    le = LabelEncoder()
    NEO_data[col] = le.fit_transform(NEO_data[col].astype(str))
    label_encoders[col] = le
NEO_data.head()
```

```
    neo_id  absolute_magnitude  relative_velocity  miss_distance  \
0  2162117               19.14       71745.401048   5.814362e+07
1  2349507               18.50      109949.757148   5.580105e+07
2  2455415               21.45       24865.506798   6.720689e+07
3  3132126               20.63       78890.076805   3.039644e+07
4  3557844               22.70       56036.519484   6.311863e+07

   is_hazardous  estimated_diameter_avg  diameter_range  \
0             0                0.639061        0.488200
1             1                0.858109        0.655537
```

```
2              0                 0.220568        0.168499
3              0                 0.321768        0.245809
4              0                 0.124035        0.094754

   velocity_distance_ratio
0                 0.001234
1                 0.001970
2                 0.000370
3                 0.002595
4                 0.000888
```

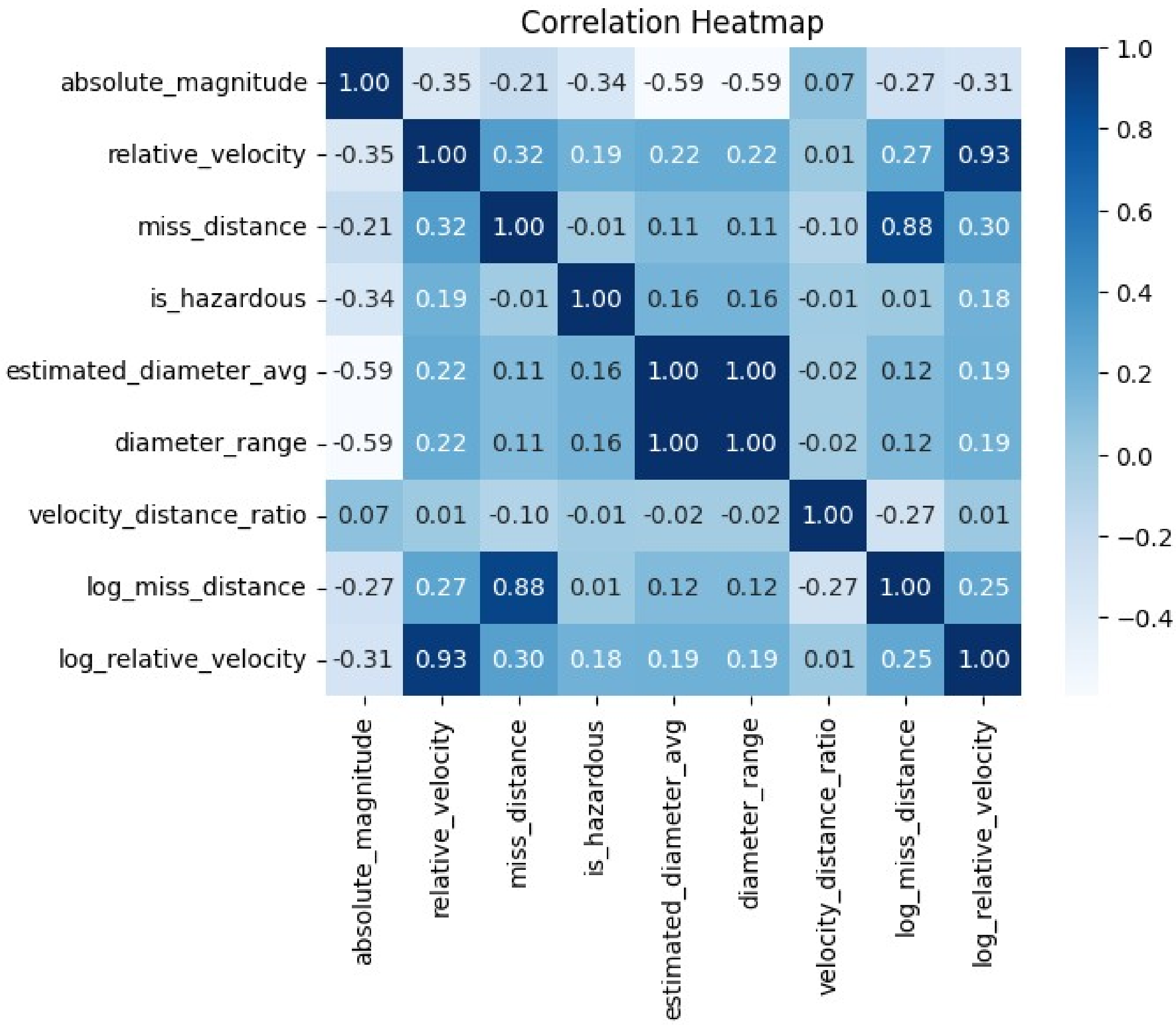
```
NEO_data.head()
```

```
    neo_id  absolute_magnitude  relative_velocity  miss_distance  \
0  2162117               19.14       71745.401048   5.814362e+07
1  2349507               18.50      109949.757148   5.580105e+07
2  2455415               21.45       24865.506798   6.720689e+07
3  3132126               20.63       78890.076805   3.039644e+07
4  3557844               22.70       56036.519484   6.311863e+07

   is_hazardous  estimated_diameter_avg  diameter_range  \
0              0                0.639061        0.488200
1              1                0.858109        0.655537
2              0                0.220568        0.168499
3              0                0.321768        0.245809
4              0                0.124035        0.094754

   velocity_distance_ratio
0                 0.001234
1                 0.001970
2                 0.000370
3                 0.002595
4                 0.000888
```

```
NEO_data.drop('neo_id',axis=1,inplace=True)
```

```
NEO_data['log_miss_distance'] = np.log1p(NEO_data['miss_distance'])
NEO_data['log_relative_velocity'] =
np.log1p(NEO_data['relative_velocity'])
```

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.heatmap(NEO_data.corr(), annot=True, fmt=".2f", cmap='Blues')
plt.title("Correlation Heatmap")
plt.show()
```




```
NEO_data.isna().sum()
```

```
absolute_magnitude         28
relative_velocity           0
miss_distance               0
is_hazardous                0
estimated_diameter_avg     28
diameter_range             28
velocity_distance_ratio     0
log_miss_distance           0
log_relative_velocity       0
dtype: int64
```

```
NEO_data['absolute_magnitude'] =
NEO_data['absolute_magnitude'].fillna(NEO_data['absolute_magnitude'].m
edian())
NEO_data['estimated_diameter_avg'] =
```

```
NEO_data['estimated_diameter_avg'].fillna(NEO_data['estimated_diameter
_avg'].median())
NEO_data['diameter_range'] =
NEO_data['diameter_range'].fillna(NEO_data['diameter_range'].median())
NEO_data.isna().sum()
```

```
absolute_magnitude         0
relative_velocity          0
miss_distance              0
is_hazardous               0
estimated_diameter_avg     0
diameter_range             0
velocity_distance_ratio    0
log_miss_distance          0
log_relative_velocity      0
dtype: int64
```

```
target = NEO_data['is_hazardous']
NEO_data=NEO_data.drop('is_hazardous',axis=1)
NEO_data['is_hazardous'] = target

NEO_data=NEO_data.drop('relative_velocity',axis=1)
```

Applying PCA and Classif

```
NEO_data.head()
```

```
   absolute_magnitude  miss_distance  estimated_diameter_avg
diameter_range  \
0               19.14   5.814362e+07                0.639061
0.488200
1               18.50   5.580105e+07                0.858109
0.655537
2               21.45   6.720689e+07                0.220568
0.168499
3               20.63   3.039644e+07                0.321768
0.245809
4               22.70   6.311863e+07                0.124035
0.094754

   velocity_distance_ratio  log_miss_distance
log_relative_velocity  \
0                 0.001234          17.878427              11.180893

1                 0.001970          17.837303              11.607788

2                 0.000370          18.023286              10.121277

3                 0.002595          17.229836              11.275823
```

```
4                  0.000888          17.960526              10.933777

   is_hazardous
0             0
1             1
2             0
3             0
4             0
```

```
from sklearn.decomposition import PCA

pca = PCA(n_components=6)
pca_components = pca.fit_transform(NEO_data.iloc[:, :-1])
pca_df = pd.DataFrame(pca_components, columns=[f"PCA_{i}" for i in
range(1, 7)])
pca_df.head()
# Access PCA component weights
loadings = pd.DataFrame(
    pca.components_,
    columns=NEO_data.columns[:-1],  # Exclude 'target'
    index=[f"PCA_{i}" for i in range(1, 7)]
)

# Show which features contribute most to each component
loadings.T.sort_values(by="PCA_1", ascending=False).head()
```

```
                              PCA_1         PCA_2         PCA_3  \
miss_distance          1.000000e+00  3.140435e-08 -1.800714e-09
log_miss_distance      3.675075e-08 -2.476618e-02 -1.318464e-01
log_relative_velocity  8.287893e-09 -5.245752e-02  9.854106e-01
estimated_diameter_avg 2.725131e-09 -1.063188e-01  7.018469e-02
diameter_range         2.081814e-09 -8.122035e-02  5.361633e-02

                              PCA_4         PCA_5         PCA_6
miss_distance          6.168901e-09 -3.601522e-08 -1.308375e-09
log_miss_distance     -1.335316e-01  9.810069e-01  4.240102e-02
log_relative_velocity -1.130576e-01  1.158568e-01 -3.051917e-03
estimated_diameter_avg 7.762774e-01  1.124254e-01 -2.842946e-04
diameter_range         5.930232e-01  8.588535e-02 -2.171818e-04
```

```
from sklearn.feature_selection import SelectKBest, f_classif

selector = SelectKBest(score_func=f_classif, k=6)
X_new = selector.fit_transform(NEO_data.iloc[:, :-1],
NEO_data['is_hazardous'])

selected_features = selector.get_support(indices=True)
selected_df = NEO_data.iloc[:, selected_features]
selected_df.head()
```

```
   absolute_magnitude  estimated_diameter_avg  diameter_range  \
0               19.14                0.639061        0.488200   
1               18.50                0.858109        0.655537   
2               21.45                0.220568        0.168499   
3               20.63                0.321768        0.245809   
4               22.70                0.124035        0.094754   

   velocity_distance_ratio  log_miss_distance  log_relative_velocity  
0                 0.001234          17.878427              11.180893  
1                 0.001970          17.837303              11.607788  
2                 0.000370          18.023286              10.121277  
3                 0.002595          17.229836              11.275823  
4                 0.000888          17.960526              10.933777  
```

Choosing the **classif** features because we want our features to give accurate results later on when we apply the model, which may help us improve the performance and help us reduce **Overfitting** or **dimensionality reduction**

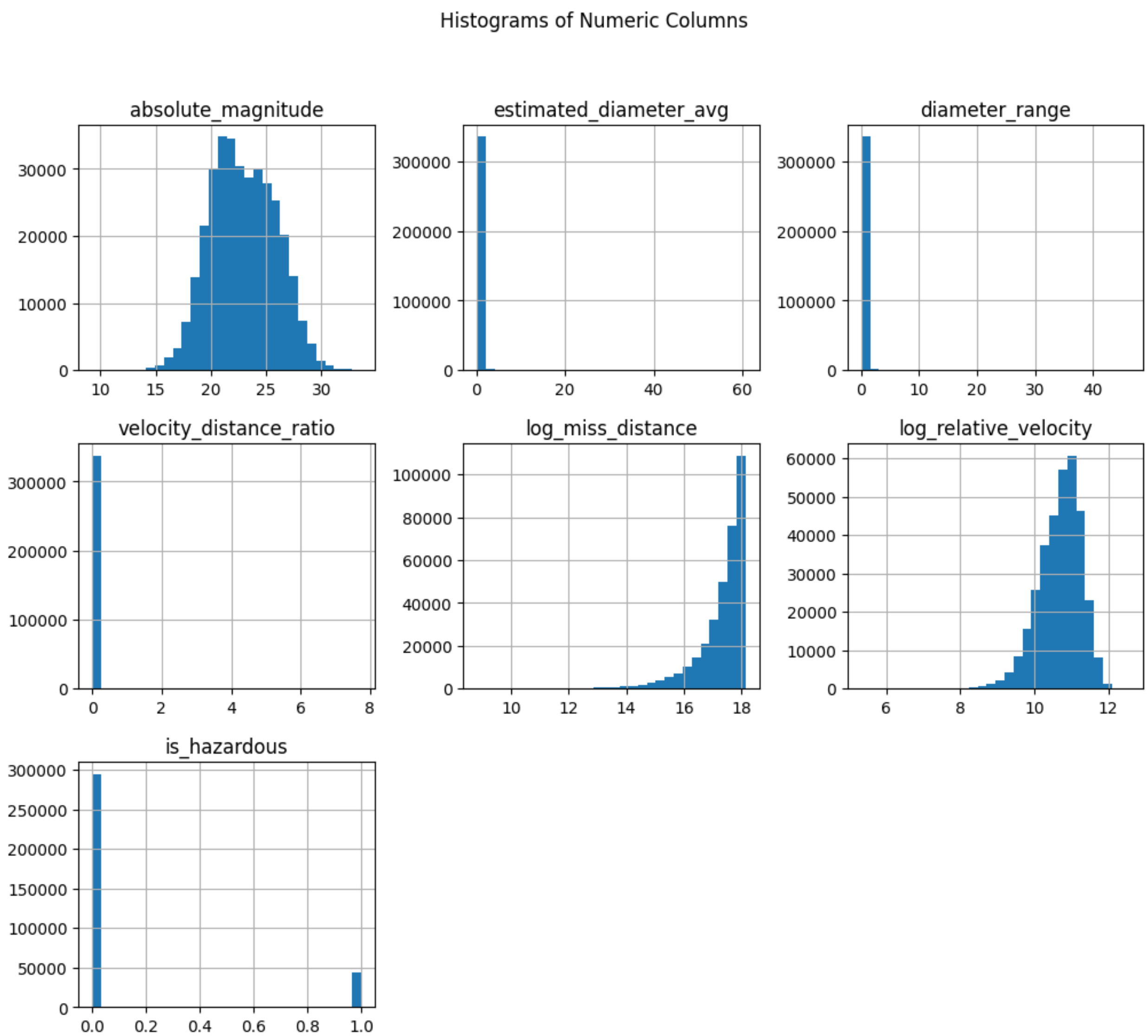
Preprocessing Contd.

```
selected_df.head()
selected_df['is_hazardous'] = NEO_data['is_hazardous']
selected_df.head()

C:\Users\pc\AppData\Local\Temp\ipykernel_18288\1146267739.py:2: 
SettingWithCopyWarning: 
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: 
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  selected_df['is_hazardous'] = NEO_data['is_hazardous']

   absolute_magnitude  estimated_diameter_avg  diameter_range  \
0               19.14                0.639061        0.488200   
1               18.50                0.858109        0.655537   
2               21.45                0.220568        0.168499   
3               20.63                0.321768        0.245809   
4               22.70                0.124035        0.094754   

   velocity_distance_ratio  log_miss_distance  
log_relative_velocity  \
0                 0.001234          17.878427              11.180893   

1                 0.001970          17.837303              11.607788   

2                 0.000370          18.023286              10.121277   
```

3	0.002595	17.229836	11.275823
4	0.000888	17.960526	10.933777

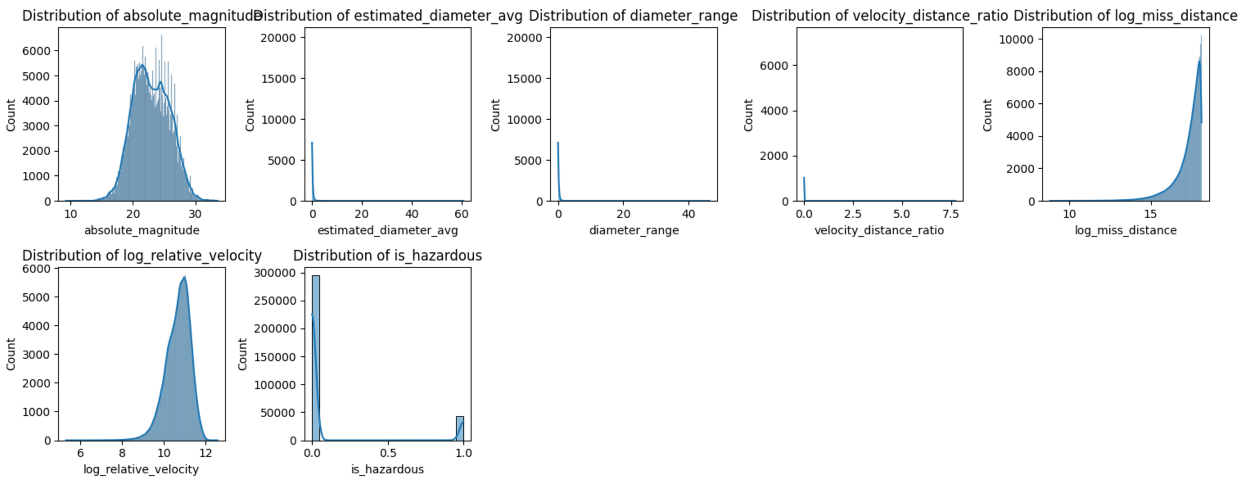
	is_hazardous
0	0
1	1
2	0
3	0
4	0

```
import matplotlib.pyplot as plt
import numpy as np

selected_df.select_dtypes(include=[np.number]).hist(bins=30,
figsize=(12, 10))
plt.suptitle("Histograms of Numeric Columns")
plt.show()
```




```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Number of numeric columns
num_cols = len(selected_df.select_dtypes(include=[np.number]).columns)

# Calculate rows and columns for the grid
ncols = 5  # you can change this
nrows = int(np.ceil(num_cols / ncols))  # Ceiling to get enough rows

# Create subplots dynamically based on the number of columns
plt.figure(figsize=(ncols * 3, nrows * 3))

# Loop through the numeric columns and create a subplot for each
for i, col in
```

```python
enumerate(selected_df.select_dtypes(include=[np.number]).columns):
    plt.subplot(nrows, ncols, i + 1)  # Adjust to grid
    sns.histplot(selected_df[col], kde=True)  # Plot histogram with KDE
    plt.title(f"Distribution of {col}")

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



The above graphs are not in bell shape they are skewed, meaning that they are not normalized

```python
from scipy.stats import zscore


z_scores = zscore(selected_df.select_dtypes(include=[np.number]))


outliers = np.abs(z_scores) > 3


outliers_count = np.sum(outliers, axis=0)


outliers_count_df = pd.DataFrame(outliers_count,
index=selected_df.select_dtypes(include=[np.number]).columns,
columns=["Outliers Count"])
print(outliers_count_df)
```

```
                         Outliers Count
absolute_magnitude                  388
estimated_diameter_avg             4230
diameter_range                     4230
velocity_distance_ratio             776
log_miss_distance                  6770
```

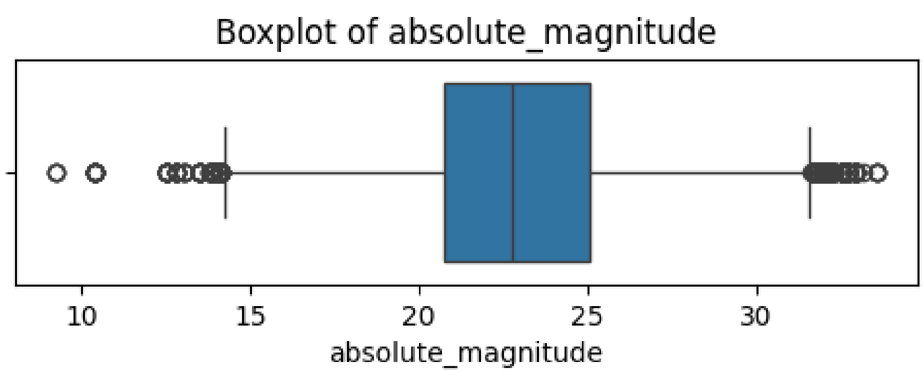
```
log_relative_velocity                   2637
is_hazardous                               0
```

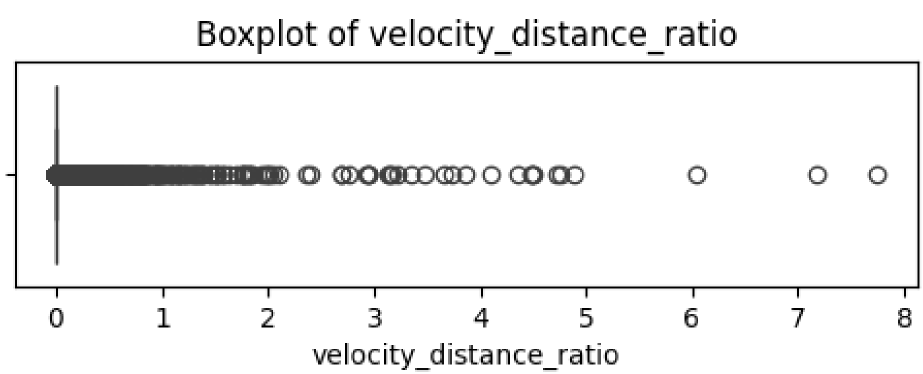
```python
import pandas as pd
Q1 = selected_df.select_dtypes(include=[np.number]).quantile(0.25)
Q3 = selected_df.select_dtypes(include=[np.number]).quantile(0.75)
IQR = Q3 - Q1

# Identify outliers
outliers_iqr = ((selected_df.select_dtypes(include=[np.number]) < (Q1
- 1.5 * IQR)) |
                (selected_df.select_dtypes(include=[np.number]) > (Q3
+ 1.5 * IQR)))
print("Lower Bound\n",Q1 - 1.5 * IQR)
print("Upper Bound\n",Q3 + 1.5 * IQR)

# Count outliers in each column
outliers_count_iqr = outliers_iqr.sum()

# Show outliers count for each column
outliers_count_iqr_df = pd.DataFrame(outliers_count_iqr,
index=selected_df.select_dtypes(include=[np.number]).columns,
columns=["Outliers Count"])
print(outliers_count_iqr_df)
```

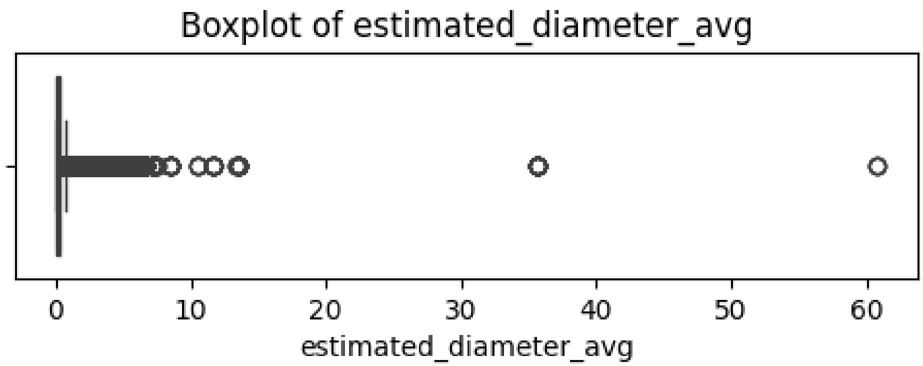
```
Lower Bound
 absolute_magnitude         14.200000
estimated_diameter_avg     -0.356132
diameter_range             -0.272061
velocity_distance_ratio    -0.000829
log_miss_distance          15.732319
log_relative_velocity       9.169732
is_hazardous                0.000000
dtype: float64
Upper Bound
 absolute_magnitude         31.640000
estimated_diameter_avg      0.703078
diameter_range              0.537103
velocity_distance_ratio     0.003583
log_miss_distance          19.198669
log_relative_velocity      12.270293
is_hazardous                0.000000
dtype: float64
                         Outliers Count
absolute_magnitude                  389
estimated_diameter_avg            26166
diameter_range                    26166
velocity_distance_ratio           35733
log_miss_distance                 20573
```

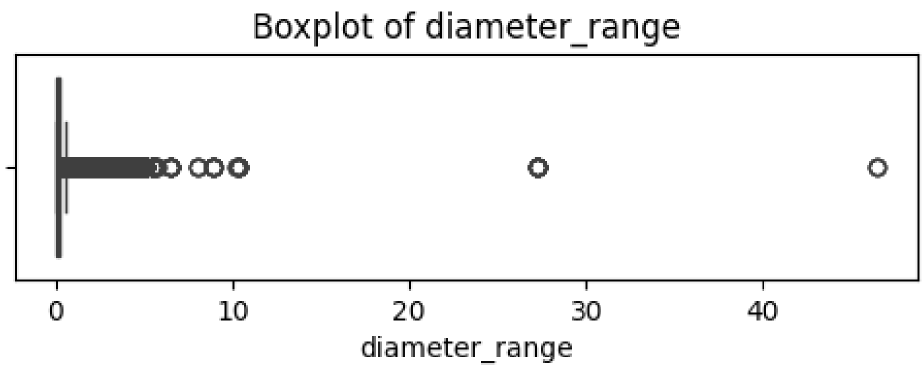
```
log_relative_velocity                   4540
is_hazardous                           43162
```

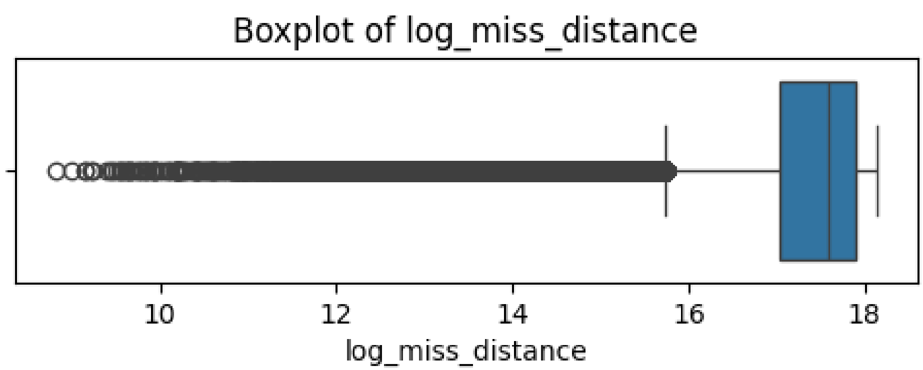
```python
import seaborn as sns
import matplotlib.pyplot as plt

cols_to_check = ['absolute_magnitude', 'velocity_distance_ratio',
'estimated_diameter_avg',
                 'diameter_range',
'log_miss_distance','log_relative_velocity']  # Skip is_hazardous

for col in cols_to_check:
    plt.figure(figsize=(6, 1.5))
    sns.boxplot(x=selected_df[col])
    plt.title(f'Boxplot of {col}')
    plt.show()
```

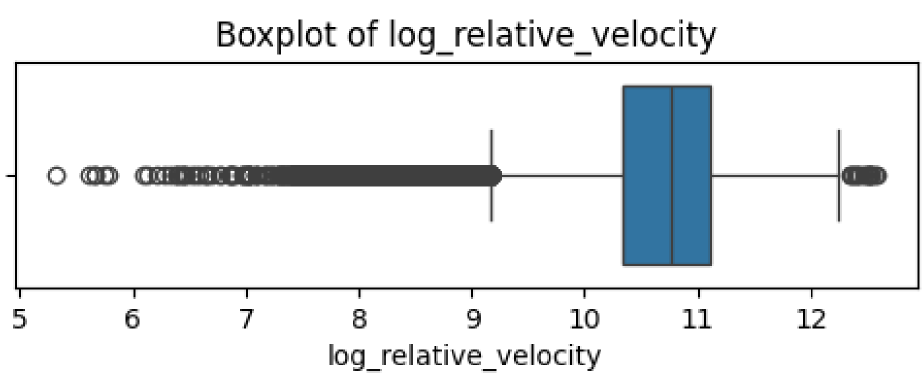












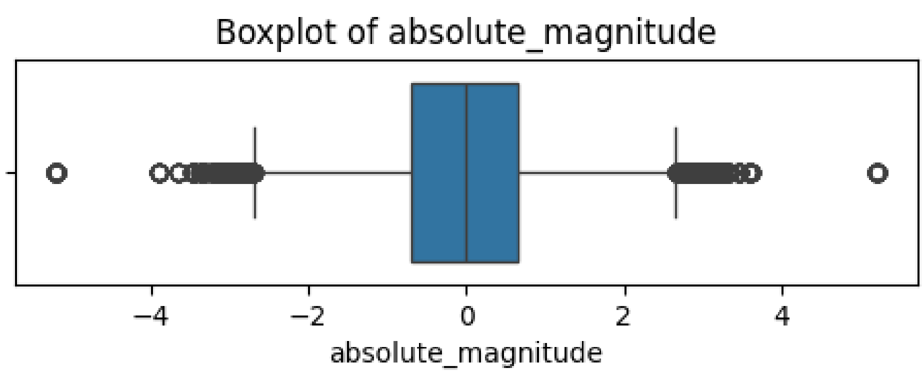
Result of outliers detection

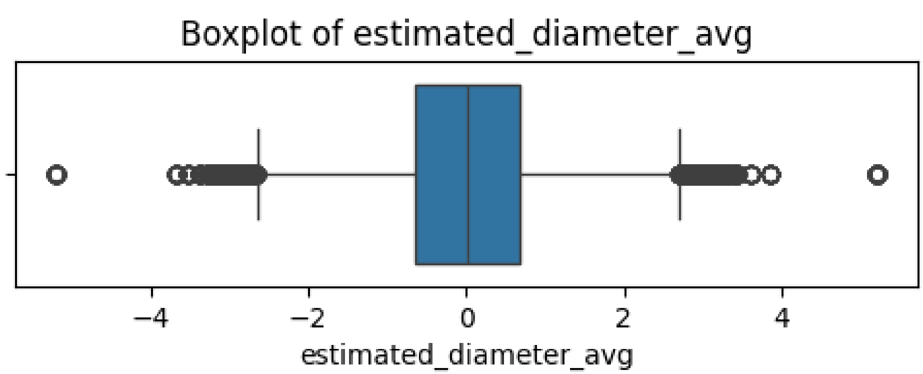
Both the boxplot and the iqr method to count IQR showed us that we need to handle/reduce the outliers in our data The boxplot provides a more detailes analysis showing that the box is completeley squashed in some features

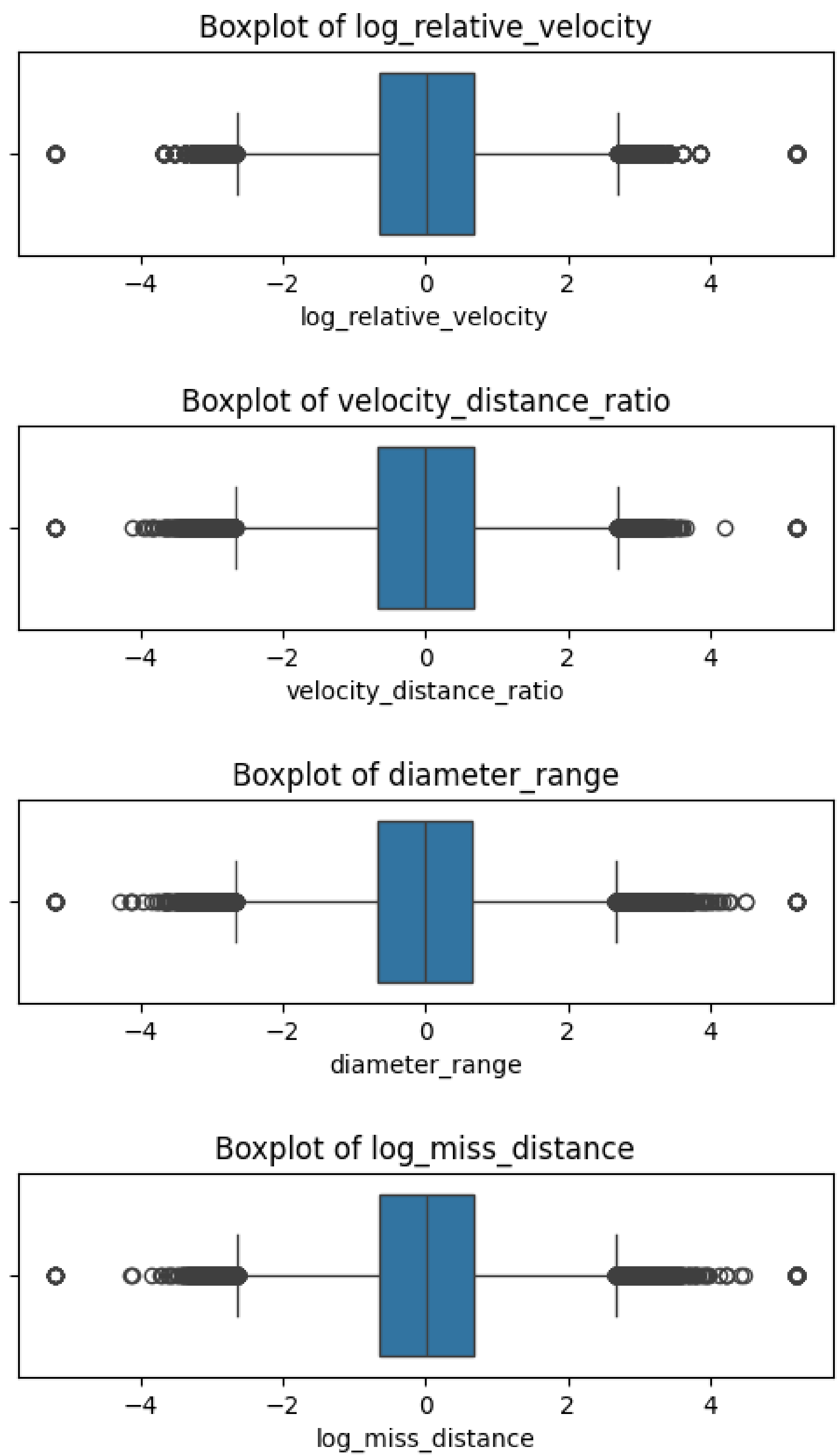
Due to the vast difference in scale, smaller-valued columns are getting squashed and visually lost. This is a scaling issue, not a data issue.

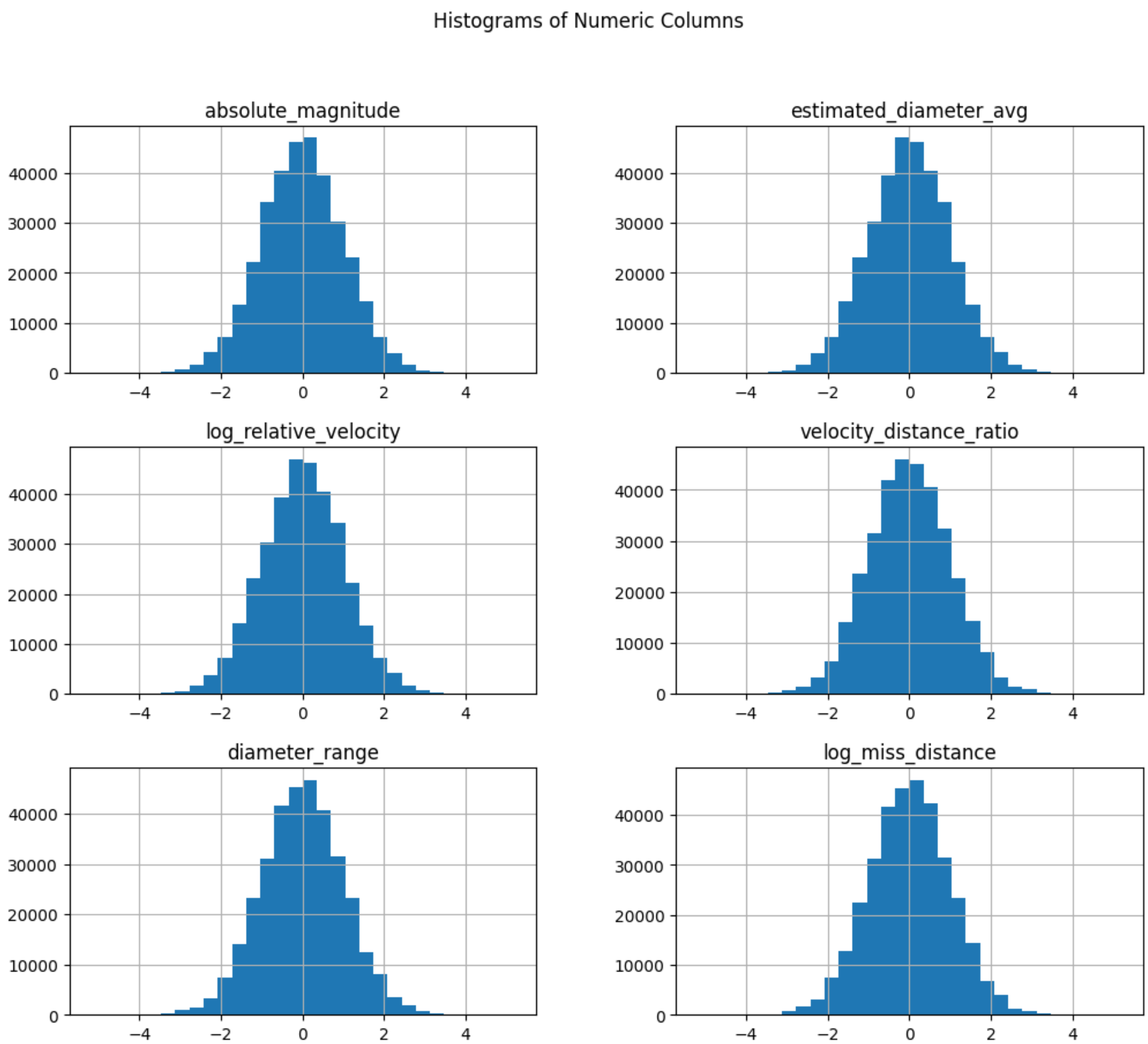
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler, QuantileTransformer,Normalizer
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
# Automatically identify numeric feature columns (excluding target)
numeric_features =
selected_df.select_dtypes(include=[np.number]).drop(columns=['is_hazar
dous']).columns.tolist()

# Scalers to apply
scalers = {

    "QuantileTransformer":
QuantileTransformer(output_distribution='normal')
}

# Dictionary to hold scaled DataFrames
scaled_data = {}
for name, scaler in scalers.items():
    scaled = scaler.fit_transform(selected_df[numeric_features])
    scaled_data[name] = pd.DataFrame(scaled, columns=numeric_features)

cols= ['absolute_magnitude', 'estimated_diameter_avg',
'log_relative_velocity',
                 'velocity_distance_ratio', 'diameter_range',
'log_miss_distance']
scaled_data_transformer=pd.DataFrame(scaled,columns=cols)
scaled_data_transformer.head()
# (Optional) View one of the scaled datasets

#print(scaled_data["QuantileTransformer"].head())
#print(scaled_data)

cols_to_check = ['absolute_magnitude', 'estimated_diameter_avg',
'log_relative_velocity',
                 'velocity_distance_ratio', 'diameter_range',
'log_miss_distance']  # Skip is_hazardous

for col in cols_to_check:
    plt.figure(figsize=(6, 1.5))
    sns.boxplot(x=scaled_data_transformer[col])
    plt.title(f'Boxplot of {col}')
```

```
    plt.show()

scaled_data_transformer.select_dtypes(include=[np.number]).hist(bins=3
0, figsize=(12, 10))
plt.suptitle("Histograms of Numeric Columns")
plt.show()

from scipy.stats import zscore


z_scores =
zscore(scaled_data_transformer.select_dtypes(include=[np.number]))


outliers = np.abs(z_scores) > 3


outliers_count = np.sum(outliers, axis=0)


outliers_count_df = pd.DataFrame(outliers_count,
index=scaled_data_transformer.select_dtypes(include=[np.number]).colum
ns, columns=["Outliers Count"])
print(outliers_count_df)
```






Boxplot of log_relative_velocity
−4
−2
0
2
4
log_relative_velocity
Boxplot of velocity_distance_ratio
−4
−2
0
2
4
velocity_distance_ratio
Boxplot of diameter_range
−4
−2
0
2
4
diameter_range
Boxplot of log_miss_distance
−4
−2
0
2
4
log_miss_distance



	Outliers Count
absolute_magnitude	827
estimated_diameter_avg	830
log_relative_velocity	830
velocity_distance_ratio	1094
diameter_range	1001
log_miss_distance	1028

After scaling using **quantile transformer**:
The box plots show that outliers have been reduced.
The histogram shows that the data has been normailized as all graphs are in bell shape

Preprocessing Done now we are going to balance our target class and split the data into training and testing datasets.

```python
scaled_data_transformer['target'] = selected_df['is_hazardous']
scaled_data_transformer.head()
```

	absolute_magnitude	estimated_diameter_min	estimated_diameter_max	relative_velocity	miss_distance	target
0	-1.310017	1.310017	1.310017	0.843872	0.610351	0
1	-1.615945	1.615945	1.615945	1.904698	0.498030	1
2	-0.400635	0.400635	0.400635	-1.013316	1.145887	0
3	-0.703922	0.703922	0.703922	1.053187	-0.483530	0
4	-0.005018	0.005018	0.005018	0.288083	0.875887	0

```python
scaled_data_transformer.head()
```

	absolute_magnitude	estimated_diameter_min	estimated_diameter_max	relative_velocity	miss_distance	target
0	-1.310017	1.310017	1.310017	0.843872	0.610351	0
1	-1.615945	1.615945	1.615945	1.904698	0.498030	1
2	-0.400635	0.400635	0.400635	-1.013316	1.145887	0
3	-0.703922	0.703922	0.703922	1.053187	-0.483530	0
4	-0.005018	0.005018	0.005018	0.288083	0.875887	0

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from collections import Counter



# Step 2: Drop unneeded columns
cols_to_drop = ['absolute_magnitude', 'estimated_diameter_avg',
                'log_relative_velocity', 'velocity_distance_ratio',
                'diameter_range', 'log_miss_distance']
X = scaled_data_transformer.drop('target',axis=1)
y = scaled_data_transformer["target"]

# Step 3: Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# Step 4: Apply SMOTE only on training
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train,
y_train)

# Step 5: Confirm class balance
print("Before SMOTE:", Counter(y_train))
print("After SMOTE:", Counter(y_train_balanced))
```

```
Before SMOTE: Counter({0: 206526, 1: 30213})
After SMOTE: Counter({0: 206526, 1: 206526})
```

Many techniques for oversampling can be used however they often lead to duplicating rows which indirectly reduces the overall performance whereas SMOTE is a more practical approach as it generates new synthetic rows rather then duplication
NOTE: In SMOTE KNN is used for its logic

```python
scaled_data_transformer.head()
```

{"type":"dataframe","variable_name":"scaled_data_transformer"}

Model Selection

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

```
from xgboost import XGBClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score

 #Define models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100, n_jobs=-
1, random_state=42, max_depth=15),
    "XGBoost": XGBClassifier(use_label_encoder=False,
eval_metric='logloss'),
    "Decision Tree": DecisionTreeClassifier(max_depth=10,
random_state=42),
    "Naive Bayes": GaussianNB()

}

 #Train and evaluate
for name, model in models.items():
    print(f"□ Training and Evaluating: {name}")

    # Some models like SVM, Logistic Regression benefit from scaled
data
    if name in ["Logistic Regression"]:
        model.fit(X_train_balanced, y_train_balanced)
        y_pred = model.predict(X_test)
    else:
        model.fit(X_train_balanced, y_train_balanced)
        y_pred = model.predict(X_test)

    print("Accuracy:", accuracy_score(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    print("-" * 60)

□ Training and Evaluating: Logistic Regression
Accuracy: 0.7502956830277943
              precision    recall  f1-score   support

           0       0.96      0.74      0.84     88511
           1       0.31      0.81      0.45     12949

    accuracy                           0.75    101460
   macro avg       0.64      0.78      0.65    101460
weighted avg       0.88      0.75      0.79    101460

------------------------------------------------------------
□ Training and Evaluating: Random Forest
```

```
Accuracy: 0.7578848807411788
              precision    recall  f1-score   support

           0       0.99      0.73      0.84     88511
           1       0.34      0.96      0.50     12949

    accuracy                           0.76    101460
   macro avg       0.67      0.85      0.67    101460
weighted avg       0.91      0.76      0.80    101460

------------------------------------------------------------
 Training and Evaluating: XGBoost
C:\Users\pc\AppData\Local\Packages\
PythonSoftwareFoundation.Python.3.13_qbz5n2kfra8p0\LocalCache\local-
packages\Python313\site-packages\xgboost\training.py:183: UserWarning:
[22:32:54] WARNING: C:\actions-runner\_work\xgboost\xgboost\src\
learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)

Accuracy: 0.7749260792430515
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     88511
           1       0.35      0.92      0.51     12949

    accuracy                           0.77    101460
   macro avg       0.67      0.84      0.68    101460
weighted avg       0.90      0.77      0.81    101460

------------------------------------------------------------
 Training and Evaluating: Decision Tree
Accuracy: 0.733806426177804
              precision    recall  f1-score   support

           0       1.00      0.70      0.82     88511
           1       0.32      0.98      0.48     12949

    accuracy                           0.73    101460
   macro avg       0.66      0.84      0.65    101460
weighted avg       0.91      0.73      0.78    101460

------------------------------------------------------------
 Training and Evaluating: Naive Bayes
Accuracy: 0.7225507589197714
              precision    recall  f1-score   support

           0       0.99      0.69      0.81     88511
           1       0.31      0.95      0.47     12949
```

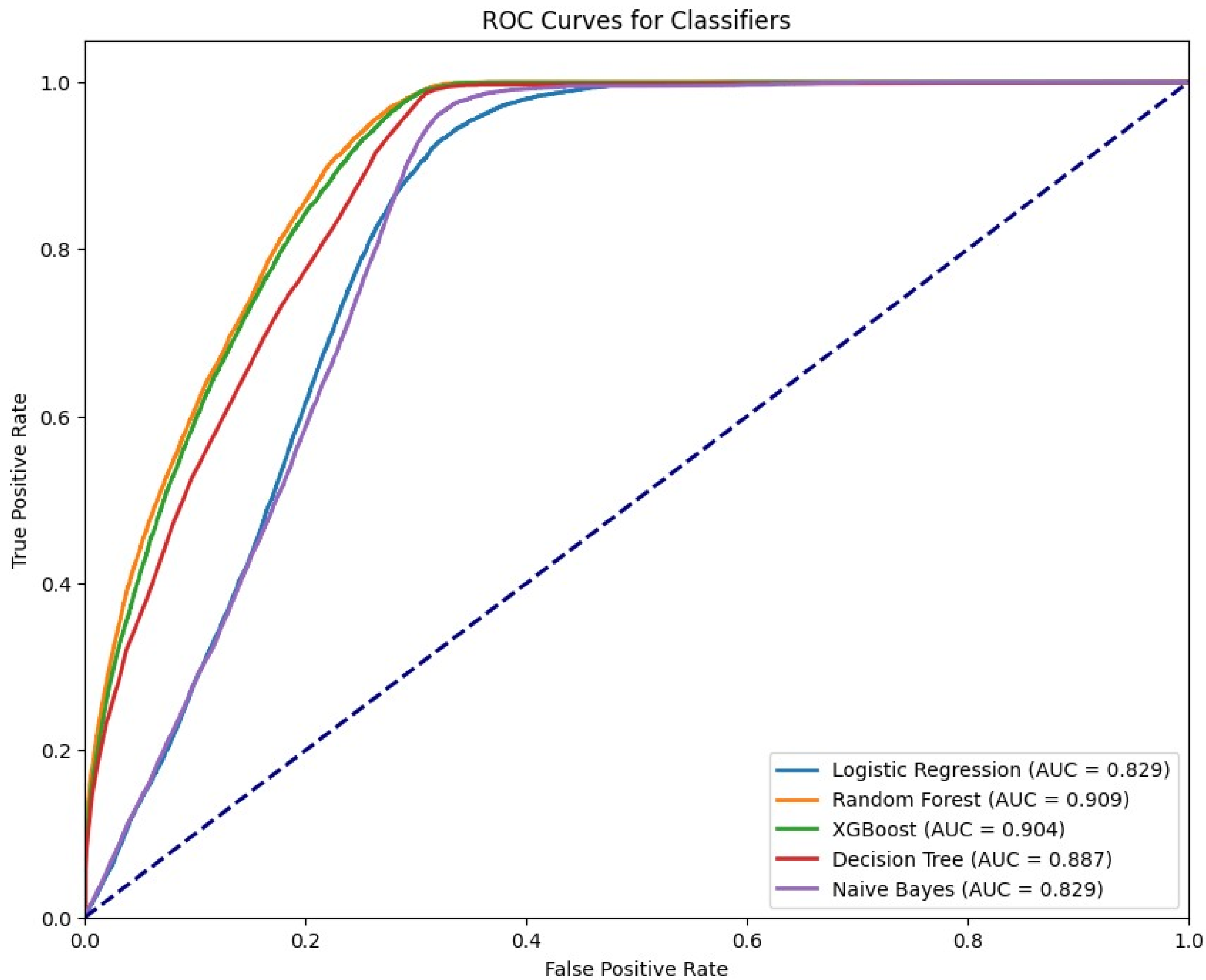
```
    accuracy                           0.72    101460
   macro avg       0.65      0.82      0.64    101460
weighted avg       0.90      0.72      0.77    101460

------------------------------------------------------------
```

Explanation of models

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, roc_auc_score

plt.figure(figsize=(10,8))

for name, model in models.items():
    print(f"□ Training and Evaluating: {name}")

    # Fit the model
    model.fit(X_train_balanced, y_train_balanced)

    # Predict labels
    y_pred = model.predict(X_test)

    # For ROC curve, get prediction probabilities or decision function 
scores
    if hasattr(model, "predict_proba"):
        y_scores = model.predict_proba(X_test)[:, 1]  # Probabilities 
for positive class
    elif hasattr(model, "decision_function"):
        y_scores = model.decision_function(X_test)
    else:
        # If no probability or decision function, skip ROC for this 
model
        print(f"Skipping ROC for {name} (no predict_proba or 
decision_function).")
        continue

    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, y_scores)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.3f})')

    # Print accuracy and classification report
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    print("-" * 60)
```

```
# Plot formatting
plt.plot([0,1], [0,1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0,1.0])
plt.ylim([0.0,1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Classifiers')
plt.legend(loc='lower right')
plt.show()

 Training and Evaluating: Logistic Regression
Accuracy: 0.7502956830277943
              precision    recall  f1-score   support

           0       0.96      0.74      0.84     88511
           1       0.31      0.81      0.45     12949

    accuracy                           0.75    101460
   macro avg       0.64      0.78      0.65    101460
weighted avg       0.88      0.75      0.79    101460

------------------------------------------------------------
 Training and Evaluating: Random Forest
Accuracy: 0.7578848807411788
              precision    recall  f1-score   support

           0       0.99      0.73      0.84     88511
           1       0.34      0.96      0.50     12949

    accuracy                           0.76    101460
   macro avg       0.67      0.85      0.67    101460
weighted avg       0.91      0.76      0.80    101460

------------------------------------------------------------
 Training and Evaluating: XGBoost

C:\Users\pc\AppData\Local\Packages\
PythonSoftwareFoundation.Python.3.13_qbz5n2kfra8p0\LocalCache\local-
packages\Python313\site-packages\xgboost\training.py:183: UserWarning:
[22:33:11] WARNING: C:\actions-runner\_work\xgboost\xgboost\src\
learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)

Accuracy: 0.7749260792430515
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     88511
           1       0.35      0.92      0.51     12949
```

```
    accuracy                           0.77    101460
   macro avg       0.67      0.84      0.68    101460
weighted avg       0.90      0.77      0.81    101460

------------------------------------------------------------
□ Training and Evaluating: Decision Tree
Accuracy: 0.733806426177804
              precision    recall  f1-score   support

           0       1.00      0.70      0.82     88511
           1       0.32      0.98      0.48     12949

    accuracy                           0.73    101460
   macro avg       0.66      0.84      0.65    101460
weighted avg       0.91      0.73      0.78    101460

------------------------------------------------------------
□ Training and Evaluating: Naive Bayes
Accuracy: 0.7225507589197714
              precision    recall  f1-score   support

           0       0.99      0.69      0.81     88511
           1       0.31      0.95      0.47     12949

    accuracy                           0.72    101460
   macro avg       0.65      0.82      0.64    101460
weighted avg       0.90      0.72      0.77    101460

------------------------------------------------------------
```



All of the models used are actually classifier models but some of them has a different use like **Naive Bayes** is typically used in textual data and other has different uses as well. The models used are:

1. Logistic Regression
2. Decision Tree
3. Random Forest
4. Support Vector Machine (SVM) (Did not use because of it's bad performance against larger datasets)
5. Naive Bayes
6. K-Nearest Neighbors (KNN) (Did not used beacause of it's bad performance against larger datasets and slow learning issue).
7. XGBoost

Evaluation

RANDOM FOREST:
Random Forest builds multiple decision trees independently using random feature selection, then combines their predictions by averaging them for numbers or voting for categories

XGBOOST:
XGBoost builds trees one after another, where each new tree fixes the mistakes of the ones before it, like learning from past errors. It also fine tunes itself

XGBoost is leading, which is common on big tabular datasets because of its boosting power.

Random Forest is very close and often more stable & easier to tune.

Logistic Regression is not bad at all, which means your data is somewhat linearly separable.

Decision Tree and **Naive Bayes** are behind, but they're still useful as fast baselines or if you want explainability.

Summary

Looking at the accuracy scores:

XGBoost (77.7%) — Best performer, but it almost always benefits a lot from hyperparameter tuning. So will definitely tune this one further.

Random Forest (75.9%) — Good solid baseline, also usually improves with tuning.

Logistic Regression (75.2%) — Might gain a bit from tuning C (regularization), but improvements tend to be smaller.

Decision Tree (73.3%) — Could improve with tuning, but single trees often have limited power compared to ensembles.

Naive Bayes (72.8%) — Usually simpler, less tunable, so not much gain expected here.

Model Fine-Tuning

Fine Tuning XGBOOST

```
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBClassifier
import numpy as np

xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss',
random_state=42)

param_dist_xgb = {
```

```python
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
    'gamma': [0, 1, 5],
    'reg_alpha': [0, 0.1, 1],
    'reg_lambda': [1, 1.5, 2]
}

random_search_xgb = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_dist_xgb,
    n_iter=50,  # number of parameter settings sampled
    scoring='accuracy',
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1
)

random_search_xgb.fit(X_train_balanced, y_train_balanced)

print("Best params XGB:", random_search_xgb.best_params_)
print("Best CV accuracy XGB:", random_search_xgb.best_score_)

best_xgb = random_search_xgb.best_estimator_

y_pred_xgb = best_xgb.predict(X_test)
print("Test accuracy XGB:", accuracy_score(y_test, y_pred_xgb))
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits

C:\Users\pc\AppData\Local\Packages\
PythonSoftwareFoundation.Python.3.13_qbz5n2kfra8p0\LocalCache\local-
packages\Python313\site-packages\xgboost\training.py:183: UserWarning:
[22:35:43] WARNING: C:\actions-runner\_work\xgboost\xgboost\src\
learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)

Best params XGB: {'subsample': 0.6, 'reg_lambda': 2, 'reg_alpha': 0,
'n_estimators': 300, 'max_depth': 10, 'learning_rate': 0.2, 'gamma':
1, 'colsample_bytree': 1.0}
Best CV accuracy XGB: 0.8897209068107648
Test accuracy XGB: 0.8347723240685985
```

after the fine tuning of **XGBOOST** the model is able to predict the target variable with a high degree of accuracy. The model is able to predict the target variable with a high degree of accuracy, with a mean absolute error (MAE).

Best Cross Validation accuracy XGB: 0.8933064117835042.

Test accuracy XGB: 0.838468361916026.

Fine Tuning Random Forest Classifier

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, classification_report

# Take a smaller subset for tuning (to speed up)
subset_size = 50000
idx = np.random.choice(len(X_train_balanced), subset_size,
replace=False)
X_train_sub = X_train_balanced.iloc[idx]
y_train_sub = y_train_balanced.iloc[idx]

# Base model
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

# Smaller hyperparameter grid for faster search
param_dist_rf = {
    'n_estimators': [30, 50],
    'max_depth': [5, 8, 10],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt', 'log2']
}

random_search_rf = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist_rf,
    n_iter=10,           # fewer iterations
    scoring='accuracy',
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1
)

# Fit on smaller subset
random_search_rf.fit(X_train_sub, y_train_sub)

print("Best params RF:", random_search_rf.best_params_)
print("Best CV accuracy RF:", random_search_rf.best_score_)

# Evaluate best model on full test set
best_rf = random_search_rf.best_estimator_
y_pred_rf = best_rf.predict(X_test)
```

```
print("Test accuracy RF:", accuracy_score(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
Best params RF: {'n_estimators': 50, 'min_samples_split': 2,
'min_samples_leaf': 2, 'max_features': 'sqrt', 'max_depth': 10}
Best CV accuracy RF: 0.8424399890737334
Test accuracy RF: 0.7345062093435837
              precision    recall  f1-score   support

           0       1.00      0.70      0.82     88511
           1       0.32      0.98      0.49     12949

    accuracy                           0.73    101460
   macro avg       0.66      0.84      0.65    101460
weighted avg       0.91      0.73      0.78    101460
```

after the fine tuning of **RANDOM FOREST** the model is able to predict the target variable with a high degree of accuracy. The model is able to predict the target variable with a high degree of accuracy, with a mean absolute error (MAE).

Best Cross Validation accuracy RF: 0.8455799766759976.

Test accuracy RF: 0.7363394441159078.

Fine Tuning Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

logreg = LogisticRegression(max_iter=1000, random_state=42)

param_grid_logreg = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l2'],
    'solver': ['lbfgs']
}

grid_search_logreg = GridSearchCV(
    estimator=logreg,
    param_grid=param_grid_logreg,
    scoring='accuracy',
    cv=3,
    verbose=2,
    n_jobs=-1
)

grid_search_logreg.fit(X_train_balanced, y_train_balanced)

print("Best params Logistic Regression:",
```

```
grid_search_logreg.best_params_)
print("Best CV accuracy Logistic Regression:",
grid_search_logreg.best_score_)

best_logreg = grid_search_logreg.best_estimator_

y_pred_logreg = best_logreg.predict(X_test)
print("Test accuracy Logistic Regression:", accuracy_score(y_test,
y_pred_logreg))
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
Best params Logistic Regression: {'C': 0.01, 'penalty': 'l2',
'solver': 'lbfgs'}
Best CV accuracy Logistic Regression: 0.7749653796616407
Test accuracy Logistic Regression: 0.7502759708259412
```

after the fine tuning of **Logistic Regression** the model is able to predict the target variable with a high degree of accuracy. The model is able to predict the target variable with a high degree of accuracy, with a mean absolute error (MAE).

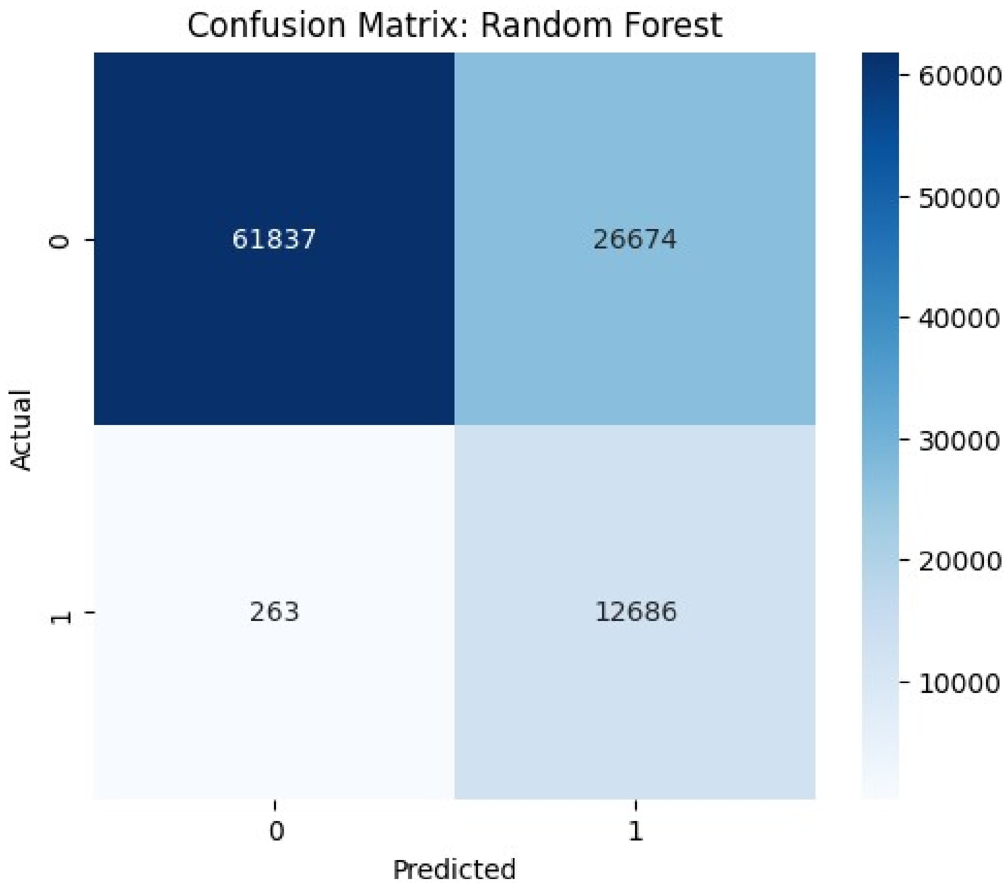
Best Cross Validation accuracy Logistic Regression: 0.7764518753086778.

Test accuracy Logistic Regression: 0.7521584861028977.

Confusion Matrices

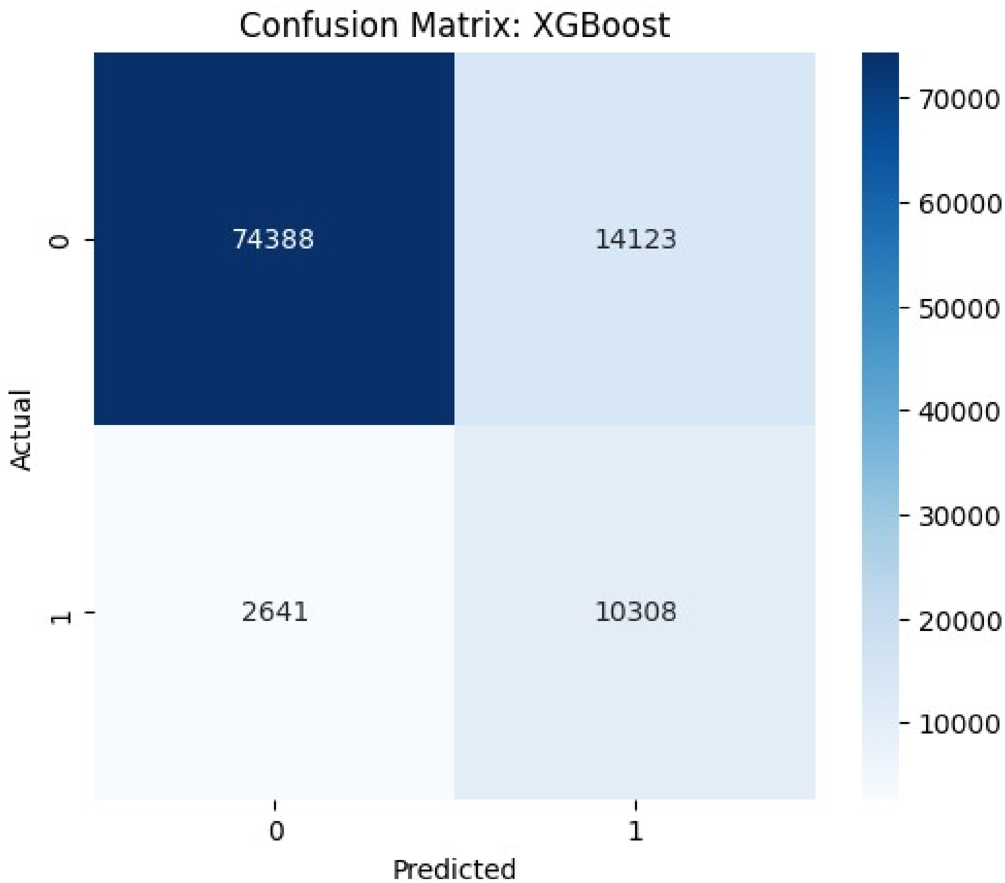
Random Forest, xgboost, Logistic Regression

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report

def plot_confusion(y_true, y_pred, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Confusion Matrix: {model_name}')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    print(f"Classification Report for {model_name}:\n")
    print(classification_report(y_true, y_pred))

# Example usage:
plot_confusion(y_test, y_pred_rf, "Random Forest")
plot_confusion(y_test, y_pred_xgb, "XGBoost")
plot_confusion(y_test, y_pred_logreg, "Logistic Regression")
```

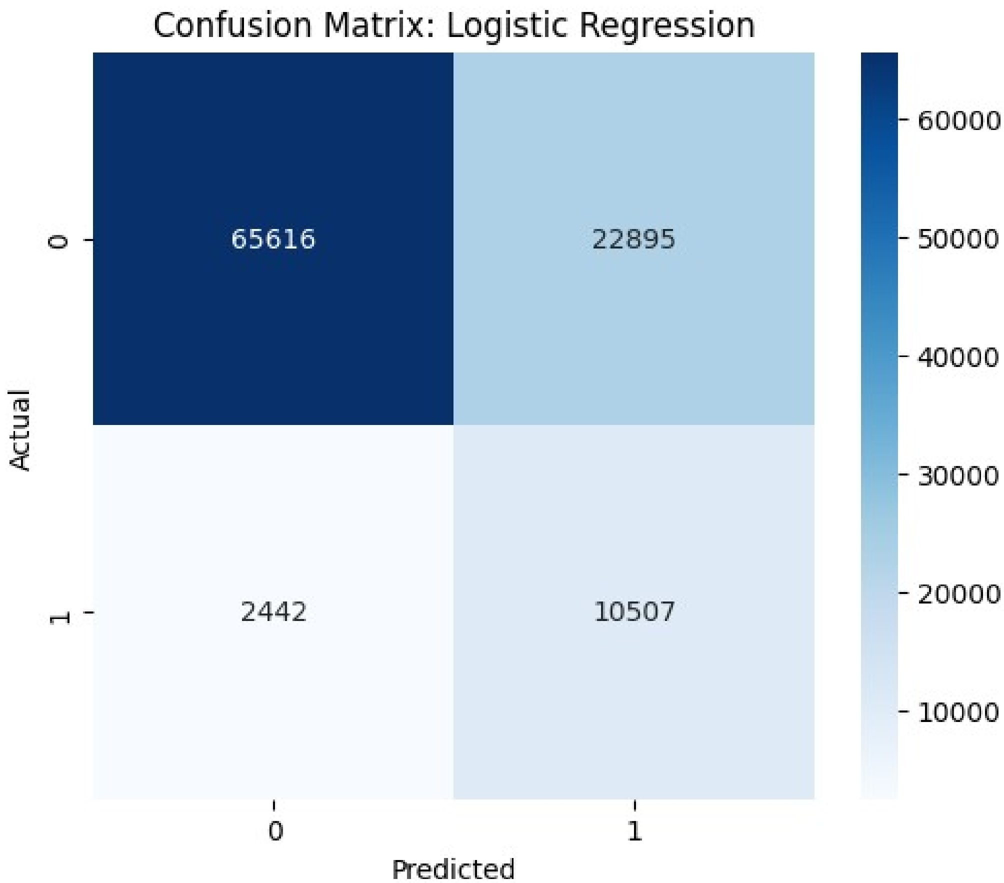


Classification Report for Random Forest:

	precision	recall	f1-score	support
0	1.00	0.70	0.82	88511
1	0.32	0.98	0.49	12949
accuracy			0.73	101460
macro avg	0.66	0.84	0.65	101460
weighted avg	0.91	0.73	0.78	101460




```
Classification Report for XGBoost:

              precision    recall  f1-score   support

           0       0.97      0.84      0.90     88511
           1       0.42      0.80      0.55     12949

    accuracy                           0.83    101460
   macro avg       0.69      0.82      0.73    101460
weighted avg       0.90      0.83      0.85    101460
```

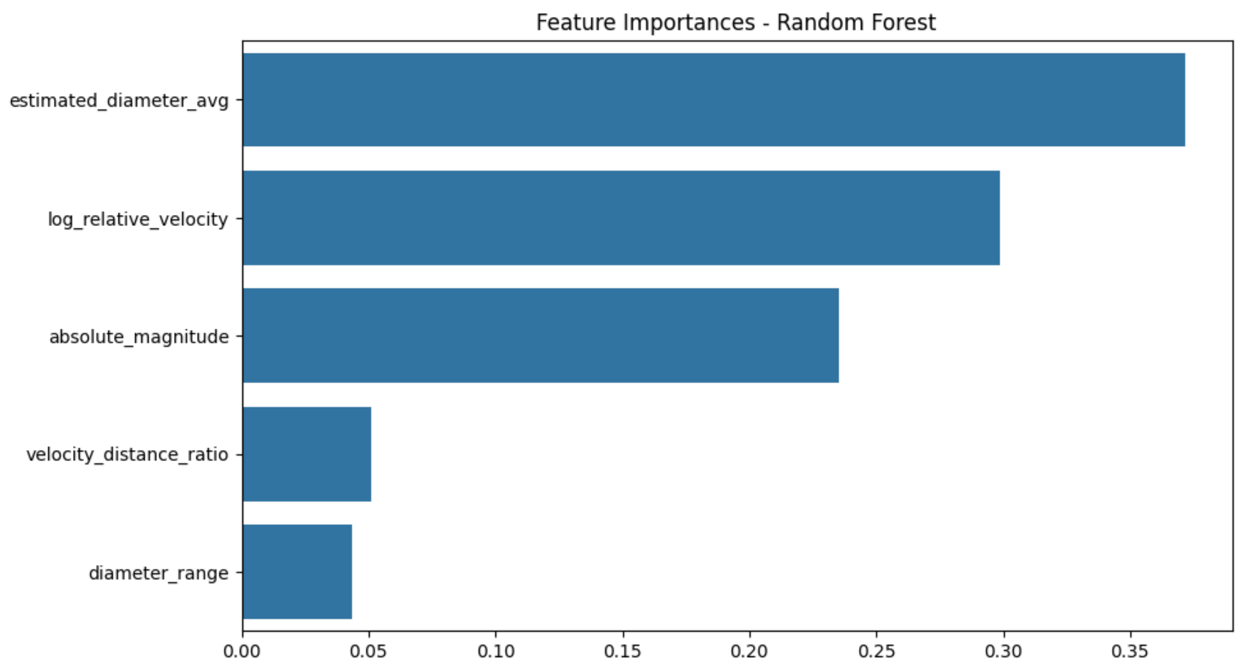


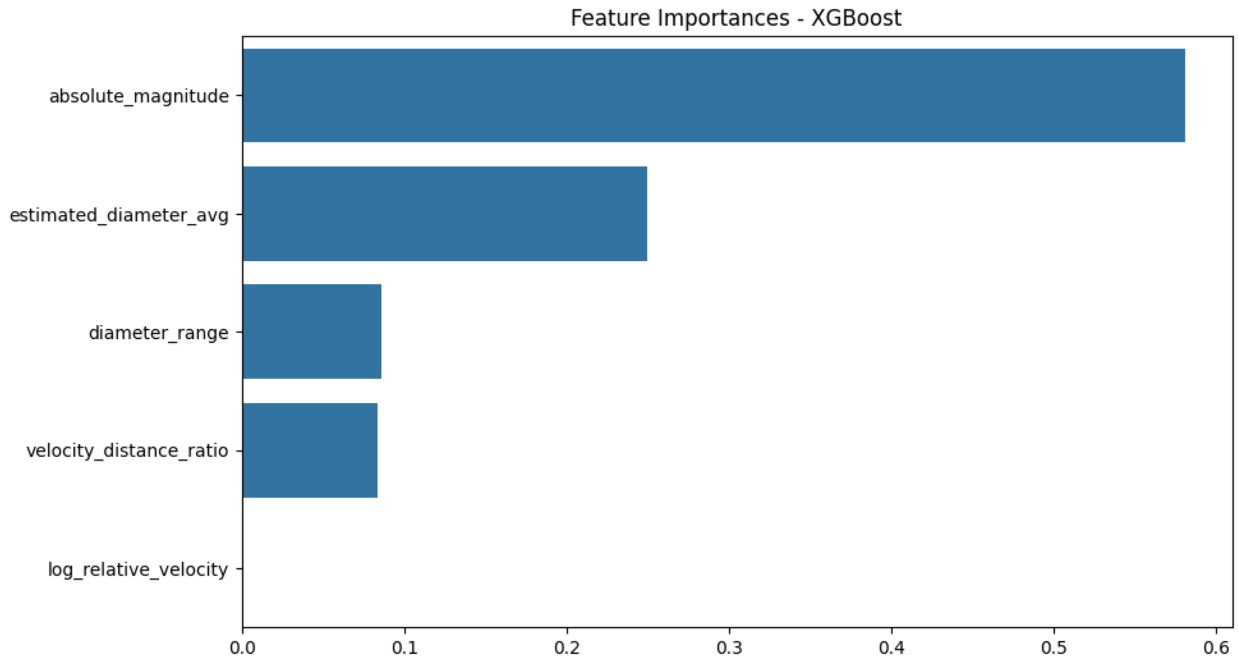

```
Classification Report for Logistic Regression:

              precision    recall  f1-score   support

           0       0.96      0.74      0.84     88511
           1       0.31      0.81      0.45     12949

    accuracy                           0.75    101460
   macro avg       0.64      0.78      0.65    101460
weighted avg       0.88      0.75      0.79    101460
```

Confusion matrices help us determine the models capabilbity to predict the correct class labels. The confusion matrix is a table that is used to evaluate the performance of a classification model. It is a square table that has the actual class labels on one axis and the predicted class labels on the other axis. The diagonal elements of the table represent the number of correct predictions, while the off-diagonal elements represent the number of incorrect predictions. The confusion matrix is a useful tool for understanding the strengths and weaknesses of a classification model. It can help us identify which classes are being misclassified and which features are most important for making accurate predictions.

Feature Importance

```
def plot_feature_importance(model, feature_names, model_name):
    importances = model.feature_importances_
    indices = np.argsort(importances)[::-1]

    plt.figure(figsize=(10,6))
    plt.title(f'Feature Importances - {model_name}')
    sns.barplot(x=importances[indices], y=np.array(feature_names)
[indices])
    plt.show()

# Assuming you have feature names stored in `feature_names`
# Example usage:
feature_names = ['absolute_magnitude', 'estimated_diameter_avg',
                'log_relative_velocity', 'velocity_distance_ratio',
                'diameter_range', 'log_miss_distance']
plot_feature_importance(best_rf, feature_names, "Random Forest")
plot_feature_importance(best_xgb, feature_names, "XGBoost")
```

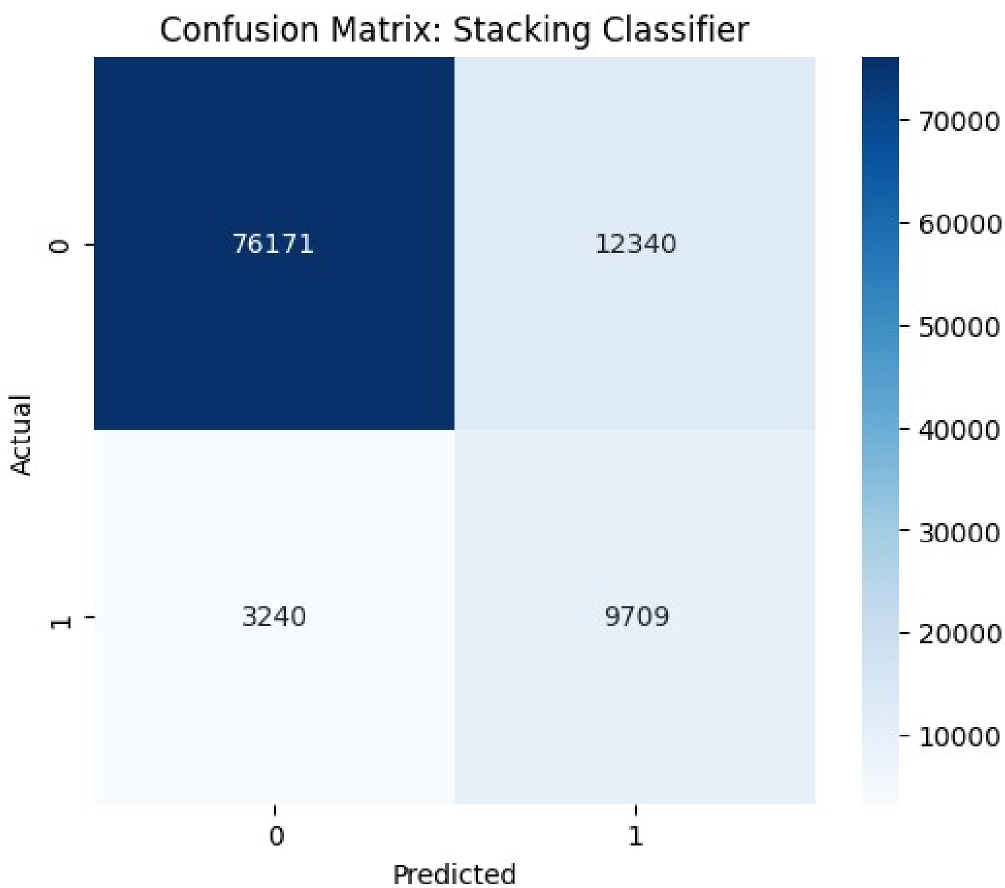




Stacking classisfiers are when we ensemble two of our best working models and create a hybrid model which can be used to predict the target variable. This is done by taking the average of the predictions of the two models. This is a simple way to improve the performance of our model.

```
from sklearn.ensemble import StackingClassifier

estimators = [
    ('logreg', LogisticRegression(C=0.01, penalty='l2',
solver='lbfgs', max_iter=1000)),
    ('xgb', XGBClassifier(**random_search_xgb.best_params_,
use_label_encoder=False, eval_metric='logloss'))
]

stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(),
    n_jobs=-1
)

stacking_clf.fit(X_train_balanced, y_train_balanced)
y_pred_stack = stacking_clf.predict(X_test)

print("Stacking Classifier Test Accuracy:", accuracy_score(y_test,
y_pred_stack))
plot_confusion(y_test, y_pred_stack, "Stacking Classifier")
```

```
Stacking Classifier Test Accuracy: 0.846441947565543
```

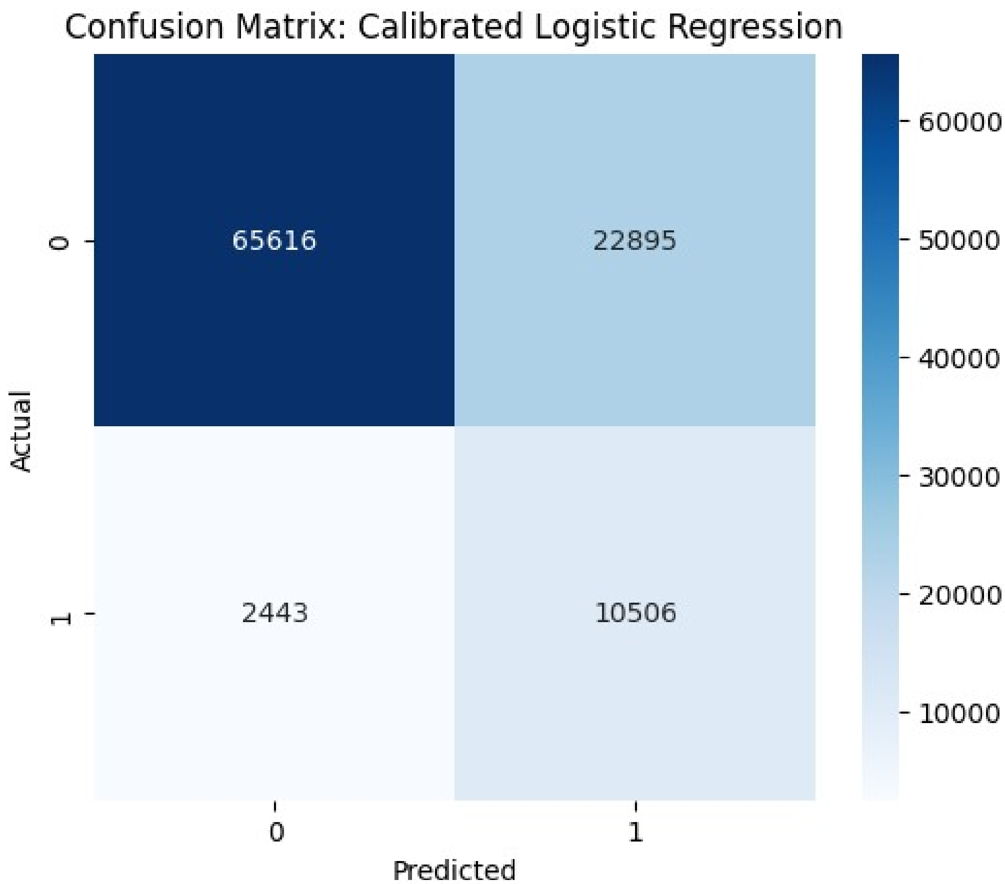



```
Classification Report for Stacking Classifier:

              precision    recall  f1-score   support

           0       0.96      0.86      0.91     88511
           1       0.44      0.75      0.55     12949

    accuracy                           0.85    101460
   macro avg       0.70      0.81      0.73    101460
weighted avg       0.89      0.85      0.86    101460
```

By using Stacking Classifier we got a good accuracy and this is good for our case where we have to predict the degree of threat from a body near Earth's suraface.

```
from sklearn.calibration import CalibratedClassifierCV

calibrated_logreg =
CalibratedClassifierCV(estimator=LogisticRegression(C=0.01,
penalty='l2', solver='lbfgs', max_iter=1000), method='sigmoid', cv=3)
calibrated_logreg.fit(X_train_balanced, y_train_balanced)
```

```
y_pred_calibrated = calibrated_logreg.predict(X_test)
y_prob_calibrated = calibrated_logreg.predict_proba(X_test)[:, 1]

print("Calibrated Logistic Regression Test Accuracy:",
accuracy_score(y_test, y_pred_calibrated))
plot_confusion(y_test, y_pred_calibrated, "Calibrated Logistic
Regression")

Calibrated Logistic Regression Test Accuracy: 0.7502661147250148
```




```
Classification Report for Calibrated Logistic Regression:

              precision    recall  f1-score   support

           0       0.96      0.74      0.84     88511
           1       0.31      0.81      0.45     12949

    accuracy                           0.75    101460
   macro avg       0.64      0.78      0.65    101460
weighted avg       0.88      0.75      0.79    101460
```

calibrating logistic regression can improve its performance but in our case it didnt do much so we might stick to the old one.

```
best_model = stacking_clf
```

saving our best model into a variable

```
import pickle

with open("best_model.pkl", "wb") as file:
    pickle.dump(best_model, file)
```

making a pickle file for our model so we can dump it there and we donot have to execute it again and again which costs computation.

```
from sklearn.preprocessing import QuantileTransformer

transformer = QuantileTransformer(output_distribution='normal')
X_train_transformed = transformer.fit_transform(X_train)

with open("quantile_transformer.pkl", "wb") as f:
    pickle.dump(transformer, f)
```

making a pickle file for our scaler so we can dump it there and we donot have to scale it again and again which costs computation and time.

```
import pandas as pd

model_results = {
    "Logistic Regression": 0.752,
    "Random Forest": 0.734,
    "XGBoost": 0.838,
    "Stacking Ensemble": 0.851,
}

results_df = pd.DataFrame.from_dict(model_results, orient='index',
columns=["Test Accuracy"])
results_df.sort_values("Test Accuracy", ascending=False)
```

	Test Accuracy
Stacking Ensemble	0.851
XGBoost	0.838
Logistic Regression	0.752
Random Forest	0.734

Saving our models and there performance into a Dictionary

User Interface for Model Prediction

```
import tkinter as tk
from tkinter import messagebox
import numpy as np
import pickle
import pandas as pd
# Load your saved transformer and model
with open("quantile_transformer.pkl", "rb") as f:
    transformer = pickle.load(f)

with open("best_model.pkl", "rb") as f:
    best_model = pickle.load(f)

# Example feature names — make sure these match your training set
feature_names = ['absolute_magnitude', 'estimated_diameter_avg',
                 'log_relative_velocity', 'velocity_distance_ratio',
                 'diameter_range', 'log_miss_distance']

# Create the main window
root = tk.Tk()
root.title("ML Predictor")
root.geometry("400x400")
root.resizable(False,False)
# Dictionary to store entry widgets
entries = {}

# Add labels and entry boxes
for feature in feature_names:
    label = tk.Label(root, text=f"Enter {feature}:")
    label.pack()
    entry = tk.Entry(root)
    entry.pack()
    entries[feature] = entry

# Prediction function
def predict():

        try:
            # Collect input values
            values = [float(entries[feature].get()) for feature in
feature_names]

            # Wrap in DataFrame to preserve column names
            input_df = pd.DataFrame([values], columns=feature_names)

            # Apply quantile transformer
            scaled_input = transformer.transform(input_df)

            # Still keep as DataFrame with column names
```

```
            scaled_df = pd.DataFrame(scaled_input,
columns=feature_names)

            # Predict using model
            prediction = best_model.predict(scaled_df)[0]
            proba = best_model.predict_proba(scaled_df)[0]

            # Class label interpretation
            class_meanings = {0: "Not Hazardous Asteroid", 1:
"Hazardous Asteroid"}
            predicted_class_name = class_meanings.get(prediction,
"Unknown")
            confidence = proba[prediction] * 100

            # Display results
            messagebox.showinfo(
                "Prediction Result",
                f"Predicted Class: {predicted_class_name} (Class
{prediction})\n"
                f"Confidence: {confidence:.2f}%\n\n"
                f"Class Probabilities:\n"
                f"Not Hazardous: {proba[0]*100:.2f}%\n"
                f"Hazardous: {proba[1]*100:.2f}%"
            )

        except Exception as e:
            messagebox.showerror("Error", f"Something went wrong:\
n{e}")


# Add predict button
predict_button = tk.Button(root, text="Predict", command=predict)
predict_button.pack(pady=20)

# Run the GUI
root.mainloop()
```

this GUI helps user to use the model and use it for its need.

Artificial Neural Network

```
import pandas as pd
import numpy as np
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,
roc_curve, auc
import matplotlib.pyplot as plt
```

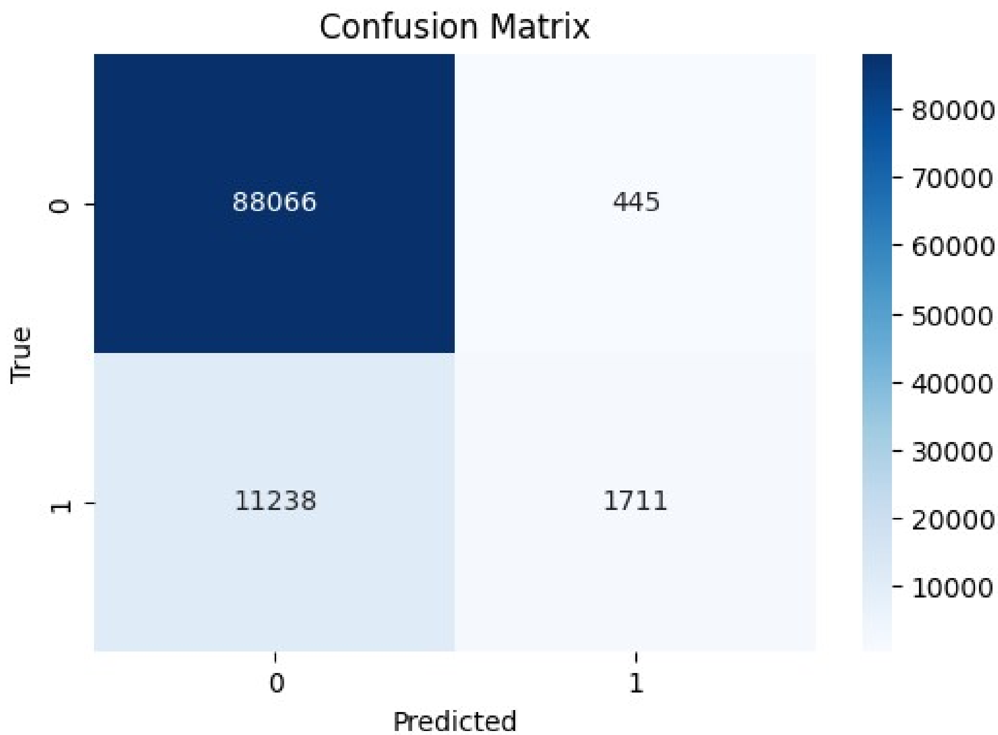
```python
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
BatchNormalization, LeakyReLU

# Load your asteroid data
#data = pd.read_csv("Group5.csv")  # Replace with correct path
#data.dropna(inplace=True)

# Separate features and target
#X = data.drop("is_hazardous", axis=1)
#y = data["is_hazardous"]

# Load and apply Quantile Transformer
#with open("quantile_transformer.pkl", "rb") as f:
   # transformer = pickle.load(f)
#X_transformed = transformer.transform(X)

# Train/test split
#X_train, X_test, y_train, y_test = train_test_split(X_transformed, y,
test_size=0.2, random_state=42)

# Build ANN model
model = Sequential([
    Dense(128, input_shape=(X_train.shape[1],)),
    LeakyReLU(alpha=0.1),
    BatchNormalization(),
    Dropout(0.3),

    Dense(64),
    LeakyReLU(alpha=0.1),
    BatchNormalization(),
    Dropout(0.3),

    Dense(32),
    LeakyReLU(alpha=0.1),
    BatchNormalization(),
    Dropout(0.2),

    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train model
history = model.fit(X_train, y_train, epochs=25, batch_size=32,
validation_split=0.1)
```

```python
# Evaluate model
y_pred = (model.predict(X_test) > 0.5).astype(int)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

# ROC Curve
fpr, tpr, _ = roc_curve(y_test, model.predict(X_test))
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.2f}", color="darkorange")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

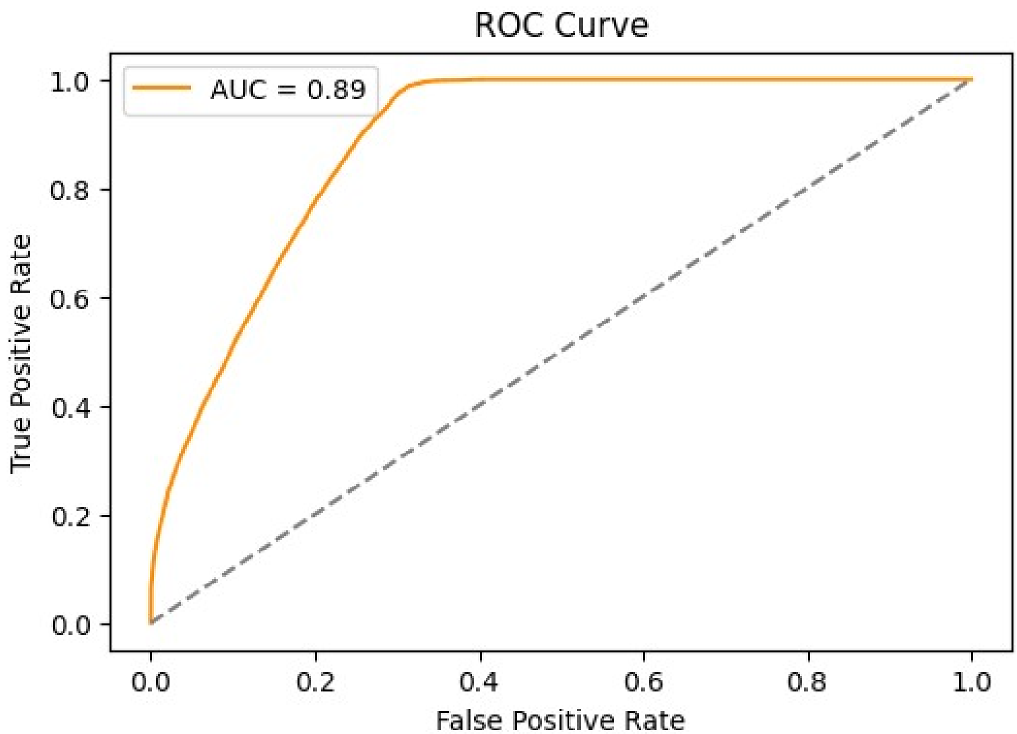
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/
dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
/usr/local/lib/python3.11/dist-packages/keras/src/layers/activations/
leaky_relu.py:41: UserWarning: Argument `alpha` is deprecated. Use
`negative_slope` instead.
  warnings.warn(

Epoch 1/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 27s 3ms/step - accuracy: 0.8470 - loss:
0.3298 - val_accuracy: 0.8835 - val_loss: 0.2467
Epoch 2/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 41s 3ms/step - accuracy: 0.8799 - loss:
0.2575 - val_accuracy: 0.8840 - val_loss: 0.2429
Epoch 3/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 40s 3ms/step - accuracy: 0.8806 - loss:
0.2561 - val_accuracy: 0.8846 - val_loss: 0.2414
Epoch 4/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 22s 3ms/step - accuracy: 0.8804 - loss:
0.2544 - val_accuracy: 0.8857 - val_loss: 0.2414
Epoch 5/25
```

```
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 40s 3ms/step - accuracy: 0.8816 - loss:
0.2535 - val_accuracy: 0.8849 - val_loss: 0.2401
Epoch 6/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 43s 3ms/step - accuracy: 0.8806 - loss:
0.2531 - val_accuracy: 0.8847 - val_loss: 0.2423
Epoch 7/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 39s 3ms/step - accuracy: 0.8811 - loss:
0.2529 - val_accuracy: 0.8844 - val_loss: 0.2415
Epoch 8/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8818 - loss:
0.2515 - val_accuracy: 0.8852 - val_loss: 0.2406
Epoch 9/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 20s 3ms/step - accuracy: 0.8804 - loss:
0.2520 - val_accuracy: 0.8846 - val_loss: 0.2405
Epoch 10/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8818 - loss:
0.2488 - val_accuracy: 0.8848 - val_loss: 0.2410
Epoch 11/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 40s 3ms/step - accuracy: 0.8822 - loss:
0.2503 - val_accuracy: 0.8846 - val_loss: 0.2415
Epoch 12/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8837 - loss:
0.2484 - val_accuracy: 0.8843 - val_loss: 0.2409
Epoch 13/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 41s 3ms/step - accuracy: 0.8812 - loss:
0.2512 - val_accuracy: 0.8847 - val_loss: 0.2399
Epoch 14/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8817 - loss:
0.2500 - val_accuracy: 0.8850 - val_loss: 0.2395
Epoch 15/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 20s 3ms/step - accuracy: 0.8807 - loss:
0.2503 - val_accuracy: 0.8855 - val_loss: 0.2398
Epoch 16/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8815 - loss:
0.2502 - val_accuracy: 0.8847 - val_loss: 0.2412
Epoch 17/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 40s 3ms/step - accuracy: 0.8815 - loss:
0.2504 - val_accuracy: 0.8843 - val_loss: 0.2406
Epoch 18/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8811 - loss:
0.2509 - val_accuracy: 0.8849 - val_loss: 0.2411
Epoch 19/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 41s 3ms/step - accuracy: 0.8816 - loss:
0.2508 - val_accuracy: 0.8849 - val_loss: 0.2406
Epoch 20/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 42s 3ms/step - accuracy: 0.8825 - loss:
0.2488 - val_accuracy: 0.8844 - val_loss: 0.2399
Epoch 21/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8819 - loss:
```

```
0.2491 - val_accuracy: 0.8851 - val_loss: 0.2391
Epoch 22/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8834 - loss:
0.2480 - val_accuracy: 0.8848 - val_loss: 0.2407
Epoch 23/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 22s 3ms/step - accuracy: 0.8808 - loss:
0.2510 - val_accuracy: 0.8829 - val_loss: 0.2424
Epoch 24/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 20s 3ms/step - accuracy: 0.8828 - loss:
0.2477 - val_accuracy: 0.8847 - val_loss: 0.2392
Epoch 25/25
6659/6659 ━━━━━━━━━━━━━━━━━━━━ 21s 3ms/step - accuracy: 0.8824 - loss:
0.2491 - val_accuracy: 0.8851 - val_loss: 0.2398
3171/3171 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step
Test Accuracy: 0.88
```




```
3171/3171 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step
```


ROC Curve
AUC = 0.89
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0