DATA ANALYTICS: Unveiling the Cosmos

TEAM SATURN RINGS

Ishan Ray | Adwit Barun | Jatin Batra | Gohil Happy

INTRODUCTION:

The problem statement for this Data Analytics event revolves around exploring and analyzing the Exoplanet dataset provided by NASA. Exoplanets are of great interest to scientists and astronomers as they provide valuable insights into the existence of potentially habitable worlds beyond our solar system.

The dataset, "phl_exoplanet_catalog_2019" contains a wealth of information about various exoplanets discovered up to the year 2019. This data encompasses a wide range of attributes related to these exoplanets, including their physical characteristics, orbital parameters, and potential habitability.

Database Field Descriptions

P_NAME - planet name

P_STATUS - planet status (confirmed = 3)

P_MASS - planet mass (earth masses)

P_MASS_ERROR_MIN - planet mass error min (earth masses)

P_MASS_ERROR_MAX - planet mass error max (earth masses)

P_RADIUS - planet radius (earth radii)

P_RADIUS_ERROR_MIN - planet radius error min (earth radii)

P_RADIUS_ERROR_MAX - planet radius error max (earth radii)

P_YEAR - planet discovered year

P_UPDATED - planet data last update date

P_PERIOD - planet period (days)

Some of the data description. Complete descriptions of data were obtained from [here](#).

We observe a few interesting things from the data. The 'P_HABITABLE' column contains the target variable with 3 nominal values.
- 0 for inhabitable
- 1 for conservatively habitable
- 2 for optimistically habitable

So, at first glance, it looks like a Supervised classification problem with 3 result classes.
We have used Google Colab to write the code for the project.

To start off, we start by importing some packages and the csv file of the data.
We start by importing packages for analysis and prediction

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Fig 1: Importing Libraries

We use pandas to operate the dataframe by importing the data using the read_csv method. This is how it looks.

```
[ ] df=pd.read_csv('phl_exoplanet_catalog_2019 - phl_exoplanet_catalog_2019.csv.csv')
    df
```

	P_NAME	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN	P_RADIUS_ERROR_MAX	P_YEAR	P_UPDATED	...
0	11 Com b	3	6165.86330	-476.74200	476.74200	NaN	NaN	NaN	2007	2014-05-14	...
1	11 UMi b	3	4684.78480	-794.57001	794.57001	NaN	NaN	NaN	2009	2018-09-06	...
2	14 And b	3	1525.57440	NaN	NaN	NaN	NaN	NaN	2008	2014-05-14	...
3	14 Her b	3	1481.07850	-47.67420	47.67420	NaN	NaN	NaN	2002	2018-09-06	...
4	16 Cyg B b	3	565.73385	-25.42624	25.42624	NaN	NaN	NaN	1996	2018-09-06	...
...	...	...	...	...	...	...	...	...	...	...	...
4043	K2-296 b	3	NaN	NaN	NaN	1.87	0.45	0.2	2019	5/23/19	...
4044	K2-296 c	3	NaN	NaN	NaN	2.76	NaN	NaN	2019	5/23/19	...
4045	GJ 1061 b	3	1.38000	0.15000	0.16000	NaN	NaN	NaN	2019	9/3/19	...

Fig 2: Primary data visualization

```
[ ] df.dtypes

P_NAME                    object
P_STATUS                   int64
P_MASS                   float64
P_MASS_ERROR_MIN         float64
P_MASS_ERROR_MAX         float64
                           ...
S_CONSTELLATION_ABR       object
S_CONSTELLATION_ENG       object
P_RADIUS_EST             float64
P_MASS_EST               float64
P_SEMI_MAJOR_AXIS_EST    float64
Length: 112, dtype: object
```

Fig 2: A brief overview of the data

1. Visualization and Analysis of Dataset:

1.1.1 Print Range, Mean, Median, and Standard Deviation of the Dataset.

We can use **df. describe()** to get the mean and std deviation of the dataset.

```
df.describe()
```

	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN	P_RADIUS_ERROR_MAX	P_YEAR	P_PERIOD	P_PERIOD_ERROR_MIN	...	S_SNOW_LINE
count	4048.0	1598.000000	1467.000000	1467.000000	3139.000000	3105.000000	3105.000000	4048.000000	3.938000e+03	3.807000e+03	...	3786.000000
mean	3.0	798.384920	-152.292232	190.289692	4.191426	-0.483990	0.621867	2014.212945	2.309342e+03	-1.073631e+03	...	3.513348
std	0.0	1406.808654	783.366353	1082.061976	4.776830	1.409048	2.007592	3.704839	1.167012e+05	5.943181e+04	...	5.463171
min	3.0	0.019070	-24965.390000	0.000000	0.336300	-54.592700	0.000000	1989.000000	9.070629e-02	-3.650000e+06	...	0.002405
25%	3.0	26.548968	-79.457001	4.449592	1.569400	-0.526870	0.145730	2014.000000	4.497336e+00	-1.129000e-03	...	1.740762
50%	3.0	273.332080	-24.154928	25.108412	2.331680	-0.235410	0.325090	2016.000000	1.187053e+01	-9.390000e-05	...	2.568600
75%	3.0	806.488560	-4.392383	85.813561	3.553570	-0.134520	0.661390	2016.000000	4.186661e+01	-1.595000e-05	...	3.661581
max	3.0	17668.059000	0.270000	26630.808000	77.349000	0.450000	68.919080	2019.000000	7.300000e+06	3.200000e-02	...	104.112780

8 rows × 98 columns

Fig 3: Mean & std of data set by .describe method in pandas

Additionally, we can even do these separately using the following methods.

The range can be calculated by the following process.

```
[ ] for (columnName, columnData) in df.iteritems():
      if(df[columnName].dtype!=object):
        print(columnName," ",df[columnName].max()-df[columnName].min())
```

Fig 4: Calculating the range of each feature

```
P_STATUS   0
P_MASS   17668.039930320003
P_MASS_ERROR_MIN   24965.66000001
P_MASS_ERROR_MAX   26630.808
P_RADIUS   77.01270000000001
P_RADIUS_ERROR_MIN   55.0427
P_RADIUS_ERROR_MAX   68.91908
P_YEAR   30
P_PERIOD   7299999.90929371
P_PERIOD_ERROR_MIN   3650000.032
P_PERIOD_ERROR_MAX   3650000.0
P_SEMI_MAJOR_AXIS   2499.9956
P_SEMI_MAJOR_AXIS_ERROR_MIN   200.001
P_SEMI_MAJOR_AXIS_ERROR_MAX   200.0
P_ECCENTRICITY   0.95
P_ECCENTRICITY_ERROR_MIN   0.48600000099999996
P_ECCENTRICITY_ERROR_MAX   0.41
P_INCLINATION   125.3
P_INCLINATION_ERROR_MIN   25.0
P_INCLINATION_ERROR_MAX   56.232
P_OMEGA   628.341
P_OMEGA_ERROR_MIN   405.9
P_OMEGA_ERROR_MAX   354.3
P_TPERI   2464881.0
```

Fig 5: Result of code in Fig 4

For the median, we can use **df. median().** The same goes for mean and standard deviation too.

```
[ ]  df.median()
```

```
df.median()
P_STATUS                     3.000000
P_MASS                     273.332080
P_MASS_ERROR_MIN           -24.154928
P_MASS_ERROR_MAX            25.108412
P_RADIUS                     2.331680
                              ...
P_HABITABLE                  0.000000
P_ESI                        0.271192
P_RADIUS_EST                 2.667980
P_MASS_EST                   7.815324
P_SEMI_MAJOR_AXIS_EST        0.102199
Length: 98, dtype: float64
```

Fig 6: Median of features by .median() method

```
df.mean()
```

```
df.mean()
P_STATUS                     3.000000
P_MASS                     798.384920
P_MASS_ERROR_MIN          -152.292232
P_MASS_ERROR_MAX           190.289692
P_RADIUS                     4.191426
                              ...
P_HABITABLE                  0.021986
P_ESI                        0.261252
P_RADIUS_EST                 5.588647
P_MASS_EST                 323.089993
P_SEMI_MAJOR_AXIS_EST        4.011385
Length: 98, dtype: float64
```

Fig 7: Mean of features by .mean() method

```
[ ]  df.std()
```

```
  df.std()
P_STATUS                    0.000000
P_MASS                   1406.808654
P_MASS_ERROR_MIN          783.366353
P_MASS_ERROR_MAX         1082.061976
P_RADIUS                    4.776830
                             ...
P_HABITABLE                 0.195731
P_ESI                       0.131333
P_RADIUS_EST                5.392733
P_MASS_EST                965.084290
P_SEMI_MAJOR_AXIS_EST      62.389968
Length: 98, dtype: float64
```

Fig 7: Standard deviation of features by .std() method

1.1.2 Does the Dataset require Normalisation?

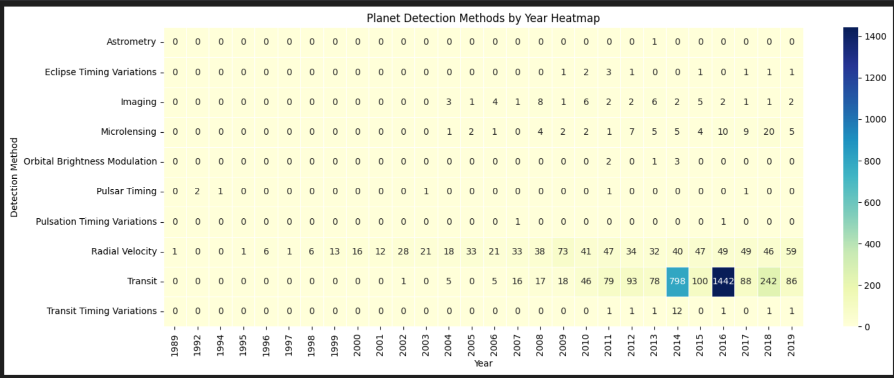
Yes, different ranges are present from 0 to 7299999. Continuing with the present ranges will result in problems in the model training stage, taking a long time or being unable to reach the optimal solution. Hence we need to normalise the dataset.

1.1.3

The described method gives us a very good understanding of the data present and helps us get an overview of how to proceed in the project. It tells us about the ranges, minimum, and maximum values, and much more.

1.2

```python
# Count the occurrences of each combination of method and year
heatmap_data = df.groupby(['P_DETECTION', 'P_YEAR']).size().unstack(fill_value=0)

# Create the heatmap
plt.figure(figsize=(16, 6))
sns.heatmap(heatmap_data, cmap='YlGnBu', annot=True, fmt='d', linewidths=0.5)
plt.xlabel('Year')
plt.ylabel('Detection Method')
plt.title('Planet Detection Methods by Year Heatmap')
plt.show()
```



1.2.1

Inferences of heat map: Transit as a detection method is quite co-related with the years 2014-2018. Whereas Astrometry as a detection method doesn't seem to have much relation/role over the years.

1.3

Creating a data frame df3 consisting only of `'P_HABITABLE'`and`'P_DETECTION'`

```python
df3=df[['P_HABITABLE','P_DETECTION']]
df3
```

	P_HABITABLE	P_DETECTION
0	0	Radial Velocity
1	0	Radial Velocity
2	0	Radial Velocity
3	0	Radial Velocity
4	0	Radial Velocity

Now, checking for Uninhabitable planets (0)

```python
df_01= df3[df3['P_HABITABLE'] == 0]
df_01.P_DETECTION.value_counts()
```

```
Transit                          3076
Radial Velocity                   748
Microlensing                       78
Imaging                            47
Transit Timing Variations          18
Eclipse Timing Variations          11
Orbital Brightness Modulation       6
Pulsar Timing                       6
Pulsation Timing Variations         2
Astrometry                          1
```

Hence, **Transit** has identified the most Uninhabitable planets.

Checking for Conservatively habitable planets (1)

```
df_02= df3[df3['P_HABITABLE'] == 1]
df_02.P_DETECTION.value_counts()
```

```
Radial Velocity    12
Transit             9
Name: P_DETECTION, dtype: int64
```

Hence, **Radial velocity** has identified the most Conservatively habitable planets.

Checking for Optimistically habitable planets (2)

```
df_03= df3[df3['P_HABITABLE'] == 2]
df_03.P_DETECTION.value_counts()
```

```
Transit            29
Radial Velocity     5
Name: P_DETECTION, dtype: int64
```

Hence, **Transit** has identified the most Optimistically habitable planets.

1.4

```
def find_iqr(x):
  return np.subtract(*np.percentile(x, [75, 25]))

#calculate IQR for all columns
df2.apply(find_iqr)
```

```
P_STATUS                  0.000000
P_MASS                         NaN
P_MASS_ERROR_MIN               NaN
P_MASS_ERROR_MAX               NaN
P_RADIUS                       NaN
                            ...
P_HABITABLE               0.000000
P_ESI                          NaN
P_RADIUS_EST             10.066580
P_MASS_EST              145.751059
P_SEMI_MAJOR_AXIS_EST          NaN
Length: 98, dtype: float64
```

The following code finds the Interquartile Range of all the columns with int and float data types.

```
<ipython-input-148-c789b15bf834>:1: Futu
  df.skew(axis = 0, skipna = True)
P_STATUS                 0.000000
P_MASS                   3.709352
P_MASS_ERROR_MIN       -23.147053
P_MASS_ERROR_MAX        19.064529
P_RADIUS                 2.957998
                          ...
P_HABITABLE              9.321263
P_ESI                    1.039325
P_RADIUS_EST             1.545462
P_MASS_EST               5.797200
P_SEMI_MAJOR_AXIS_EST   28.395487
Length: 98, dtype: float64
```

The following code finds the Skewness of all the columns with int and float data types.

1.5

```
Class=0, n=3993 (98.641%)
Class=2, n=34 (0.840%)
Class=1, n=21 (0.519%)
```

Here we can see n is very high for class 0 whereas very low for class 1 and 2.
So we need to use undersampling as well as oversampling to balance it.

Over-sampling: duplicate or create new synthetic examples in the minority class
Under-sampling: delete or merge models in the majority class

SMOTEENN combines over- and under-sampling using SMOTE and Edited Nearest-Neighbors.

After resolving the imbalance:

```
Class=0, n=3825 (32.847%)
Class=1, n=3992 (34.281%)
Class=2, n=3828 (32.872%)
```

1.6

The star metallicity (often denoted as [Fe/H]) is a measure of the abundance of elements heavier than hydrogen and helium in a star's atmosphere. It is usually expressed as the logarithm of the ratio of the number of iron (Fe) atoms to the number of hydrogen (H) atoms relative to the same ratio in the Sun.
Hence we have converted the logarithm to ratio using the following code

```
df['Fe to H Ratio']=10**df['S_METALLICITY']
df['Fe to H Ratio']
df
```

Fe to H Ratio
0.446684
0.954993
0.575440

These are some of the examples after the ratio has been calculated.

2. Interpretation and calculation of physical parameters:

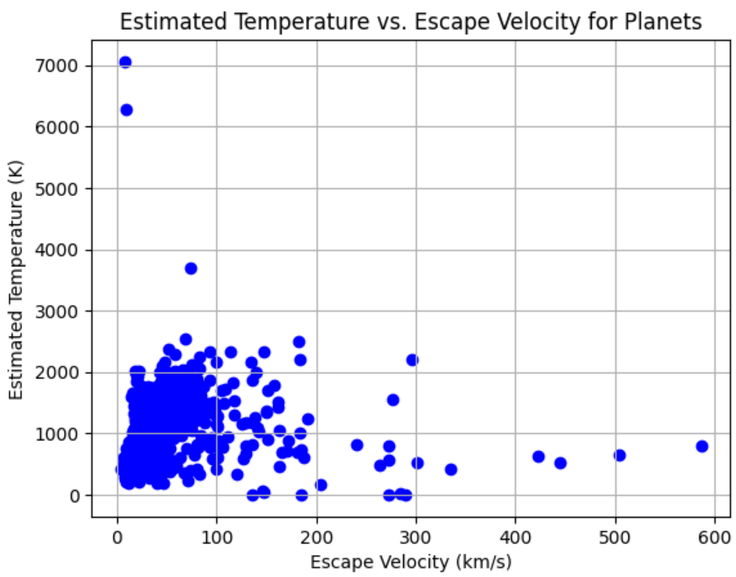
2.1

Extracting the escape velocity(given in earth units) with the planet's temperature.

```
[ ]  df4=df[['P_ESCAPE','P_TEMP_EQUIL']].dropna()
     df4['P_ESCAPE']=df4['P_ESCAPE']*11.2
     df4
```

The following code will create a scatter plot between the escape velocity(given in Earth units) and the planet's temperature.

```
plt.scatter(df4['P_ESCAPE'],df4['P_TEMP_EQUIL'], c='blue', marker='o', label='Planets')

# Add labels and a title
plt.xlabel('Escape Velocity (km/s)')
plt.ylabel('Estimated Temperature (K)')
plt.title('Estimated Temperature vs. Escape Velocity for Planets')

# Add a legend (if applicable)
# plt.legend()

# Customize the plot further as needed

# Display the plot
plt.grid(True)
plt.show()
```



Resulting scatter plot.

2.1.1

The majority of data points on the plot are concentrated in an area characterized by both low temperatures and low escape velocities. This clustering suggests that these planets might face challenges in retaining their atmospheres because their escape velocities fall below the threshold required to counteract the thermal velocities of gas molecules at their respective temperatures. Consequently, this situation implies that gas molecules are prone to gradually escaping into space over time.

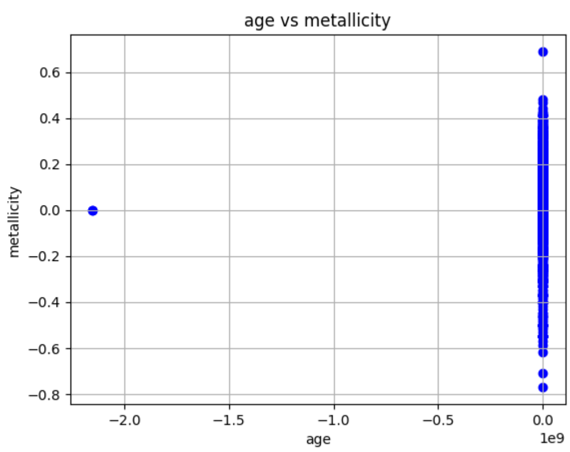
2.2

```
[ ]  df5=df[['S_AGE','S_METALLICITY']].dropna()
     df5.S_AGE=df5.S_AGE
```

```
hm = sns.heatmap(data=df5,
                 annot=True)

# displaying the plotted heatmap
plt.show()
```

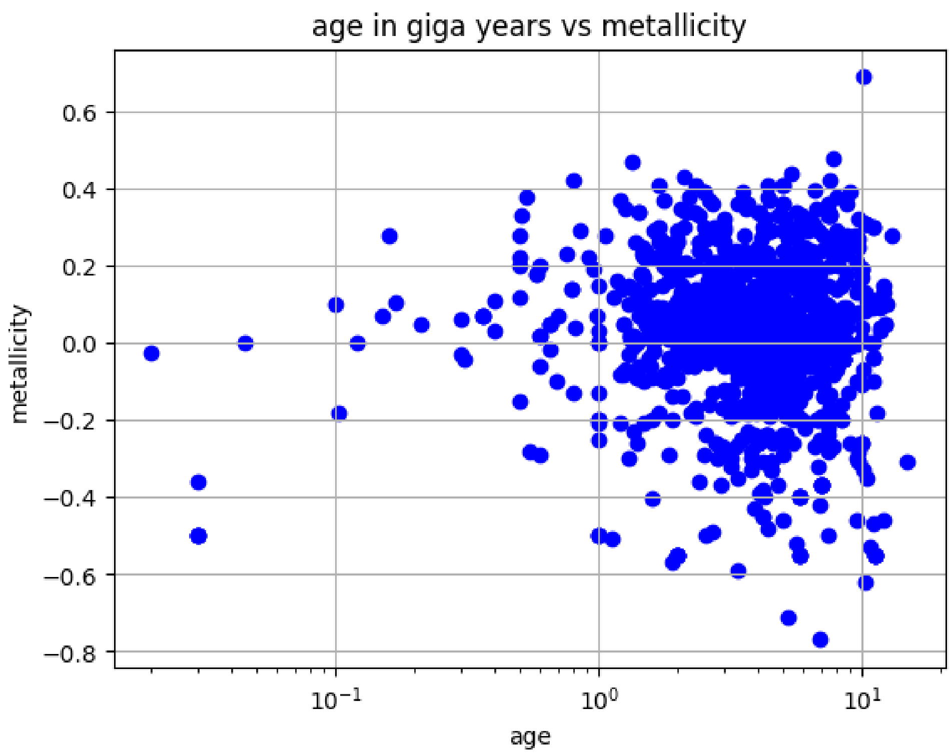
Creating a scatter plot of host star ages with the metallicity of their associated exoplanets.

```
[ ]  plt.scatter(df5['S_AGE'],df5['S_METALLICITY'], c='blue', marker='o', label='Planets')

     # Add labels and a title
     plt.xlabel('age')
     plt.ylabel('metallicity')
     plt.title('age vs metallicity')

     # Add a legend (if applicable)
     # plt.legend()

     # Customize the plot further as needed

     # Display the plot
     plt.grid(True)
     plt.show()
```



This is the resulting graph. We can observe that due to a few datasets, the graph doesn't give a very good representation of the observed data.
Hence we use the log of age to provide a better visualisation.

```python
plt.scatter(df5['S_AGE'],df5['S_METALLICITY'], c='blue', marker='o', label='Planets')

plt.xscale('log')
# Add labels and a title
plt.xlabel('age')
plt.ylabel('metallicity')
plt.title('age in giga years vs metallicity')

# Add a legend (if applicable)
# plt.legend()

# Customize the plot further as needed

# Display the plot
plt.grid(True)
plt.show()
```



2.2.1

We can see there is a positive correlation that the older stars have more metalicity that goes with stellar evolution where with time metallic behavior gets accumulated due to the heavier elements.

2.3

- Magnetic fields can influence the retention or loss of a planet's atmosphere by deflecting or capturing charged particles from the star's solar wind.
- High magnetic field strengths may provide protection from harmful stellar radiation, affecting the composition and stability of the atmosphere.
- Magnetic interactions might influence the temperature distribution in the exoplanet's atmosphere, affecting its climate and weather patterns.

2.3.1

- Magnetic shielding can help retain volatile gases like hydrogen and helium.
- Interaction with the stellar wind can lead to atmospheric escape, particularly for smaller exoplanets without strong magnetic fields.
- Consider the role of magnetic fields in preventing or causing atmospheric erosion, which can affect the long-term viability of an atmosphere.

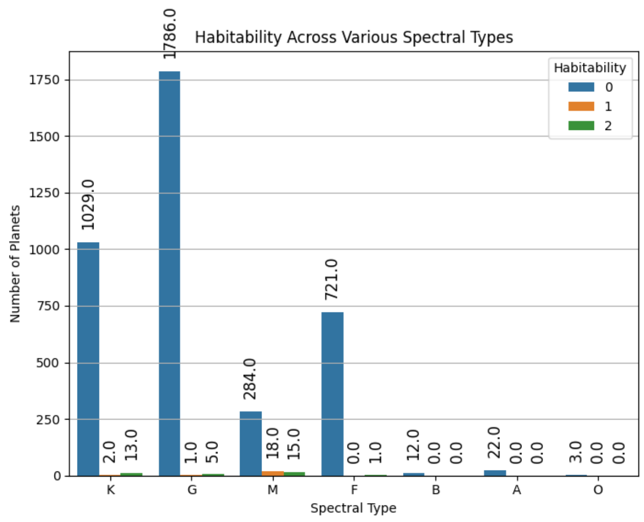
2.4

Spectral types are a classification system used by astronomers to categorize stars based on the characteristics of their spectra, which is the distribution of light emitted or absorbed by a star at different wavelengths. The spectral type of a star provides valuable information about its temperature, chemical composition, and other physical properties.

There are seven major spectral types, which are often remembered using the mnemonic "O, B, A, F, G, K, M." These spectral types are ordered from hottest to coolest

2.4.1

```
# Set a custom color palette for Habitable and Non-Habitable categories
palette = {"Habitable": "green", "Non-Habitable": "red"}

# Create a categorical plot (bar plot)
plt.figure(figsize=(8, 6))
ax=sns.countplot(data=df, x='S_TYPE_TEMP', hue='P_HABITABLE',)
for p in ax.patches:
    ax.annotate(f'{p.get_height():.1f}', (p.get_x() + p.get_width() / 2., p.get_h

# Add labels and a title
plt.xlabel('Spectral Type')
plt.ylabel('Number of Planets')
plt.title('Habitability Across Various Spectral Types')

# Show the plot
plt.legend(title='Habitability')
plt.grid(axis='y')
plt.show()
```



2.4.2

O-type, B-type, A-type, F-type, G-type, K-type, and M-type is the descending order of the temperature of the planets. We can see in the plot that O is the most inhabitable whereas M is the most habitable.

2.4.3

The spectral characteristics of a star, including its spectral type, have a significant impact on the size, density, and habitability of the exoplanets within its planetary system. Factors such as the star's metallicity, radiation output, and lifespan all play a role in shaping the properties of the planets orbiting it. For instance, a star's metallicity affects the composition of its planets, while its radiation can influence their atmospheres. The spectral type also determines the location of the habitable zone, which affects the orbital distances, sizes, and densities of exoplanets within that zone. Understanding this relationship is crucial for studying exoplanets and their potential for habitability.

3. Feature Engineering:

3.1

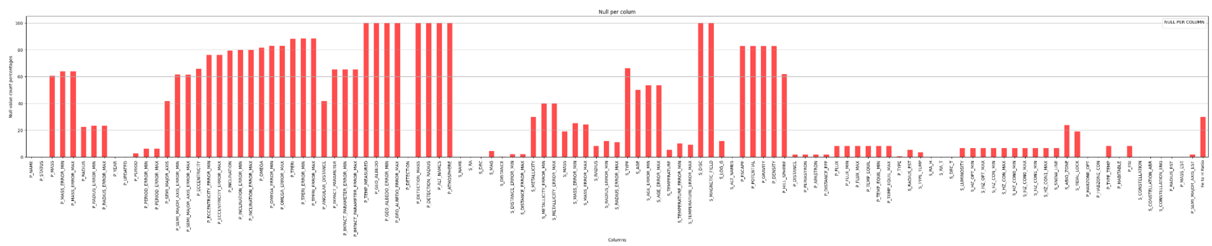
```
df.isnull().sum()
```

```
P_NAME                      0
P_STATUS                    0
P_MASS                   2450
P_MASS_ERROR_MIN         2581
P_MASS_ERROR_MAX         2581
                          ...
S_CONSTELLATION_ENG         0
P_RADIUS_EST                0
P_MASS_EST                  0
P_SEMI_MAJOR_AXIS_EST      70
Fe to H Ratio            1206
Length: 113, dtype: int64
```

Checking the number of null values in each column

Plotting the graph for the number of NULL values in each column.

```
# Create a categorical plot (bar plot)
null_counts=df.isnull().sum()/4048*100

plt.figure(figsize=(50, 6))
null_counts.plot(kind='bar',color='red',alpha=0.7)

# Add labels and a title
plt.xlabel('Columns')
plt.ylabel('Null value count percentages')
plt.title('Null per colum')

# Show the plot
plt.legend(title='NULL PER COLUMN')
plt.grid(axis='y')
plt.show()
```



3.2 Feature Reduction

3.2.1

Checking for the columns with maximum NULL values

```python
count = df.isnull().sum().sort_values(ascending=False)

#Return the fraction of a column which is filled with missing values
percent = ((df.isnull().sum()/df.isnull().count())*100).\
sort_values(ascending=False)

#Merge count and percent to display
missing = pd.concat([count, percent], axis = 1, keys = ['Count', '%'])

missing.head(60)
```

	Count	%
P_GEO_ALBEDO	4048	100.000000
S_DISC	4048	100.000000
S_MAGNETIC_FIELD	4048	100.000000
P_ATMOSPHERE	4048	100.000000
P_ALT_NAMES	4048	100.000000
P_DETECTION_RADIUS	4048	100.000000
P_DETECTION_MASS	4048	100.000000
P_GEO_ALBEDO_ERROR_MAX	4043	99.876482
P_TEMP_MEASURED	4043	99.876482
P_GEO_ALBEDO_ERROR_MIN	4043	99.876482
P_TPERI_ERROR_MAX	3576	88.339921
P_TPERI_ERROR_MIN	3576	88.339921
P_TPERI	3567	88.117589
P_OMEGA_ERROR_MIN	3355	82.880435

3.2.2

Dropping data with more than 50% missing data

```python
#Drop columns containing more than 50% missing data
df_clean = df.drop(['P_DETECTION_MASS', 'P_GEO_ALBEDO',\
'S_MAGNETIC_FIELD', 'S_DISC', 'P_ATMOSPHERE', 'P_ALT_NAMES', \
'P_DETECTION_RADIUS', 'P_GEO_ALBEDO_ERROR_MIN', 'P_TEMP_MEASURED',\
'P_GEO_ALBEDO_ERROR_MAX', 'P_TPERI_ERROR_MAX', 'P_TPERI_ERROR_MIN', \
'P_TPERI', 'P_OMEGA_ERROR_MIN', 'P_OMEGA_ERROR_MAX', 'P_DENSITY', \
'P_POTENTIAL', 'P_GRAVITY', 'P_OMEGA', \
'P_INCLINATION_ERROR_MAX', 'P_INCLINATION_ERROR_MIN', 'P_INCLINATION',\
'P_ECCENTRICITY_ERROR_MAX', 'P_ECCENTRICITY_ERROR_MIN', 'S_TYPE', \
'P_ECCENTRICITY','P_IMPACT_PARAMETER_ERROR_MIN', \
'P_IMPACT_PARAMETER_ERROR_MAX', 'P_IMPACT_PARAMETER', 'P_MASS_ERROR_MAX',\
'P_MASS_ERROR_MIN', 'P_HILL_SPHERE', 'P_SEMI_MAJOR_AXIS_ERROR_MIN',\
'P_SEMI_MAJOR_AXIS_ERROR_MAX', 'P_MASS', 'S_AGE_ERROR_MAX', \
'S_AGE_ERROR_MIN'], \

axis = 1)
```

df_clean contains the resulting dataframe.

```python
[ ] df_corr=df_clean.drop(df_clean.select_dtypes(include = ['object']).columns,axis=1)
```

Plotting the heatmap for non-object columns

```
corr_mat=df_corr.corr()
mask = np.triu(np.ones_like(corr_mat, dtype=np.bool_))

plt.figure(figsize=(12, 10))
sns.heatmap(corr_mat, center = 0,vmax = None,annot=False, cmap='coolwarm', fmt='.2f', linewidths=0.5, mask=mask,cbar_kws = {"shrink": 0.9})

# Add a title
plt.title('Correlation Plot of the Dataset')

# Show the plot
plt.show()
```

Now, we can remove one of the two columns which have a correlation greater than 0.95.

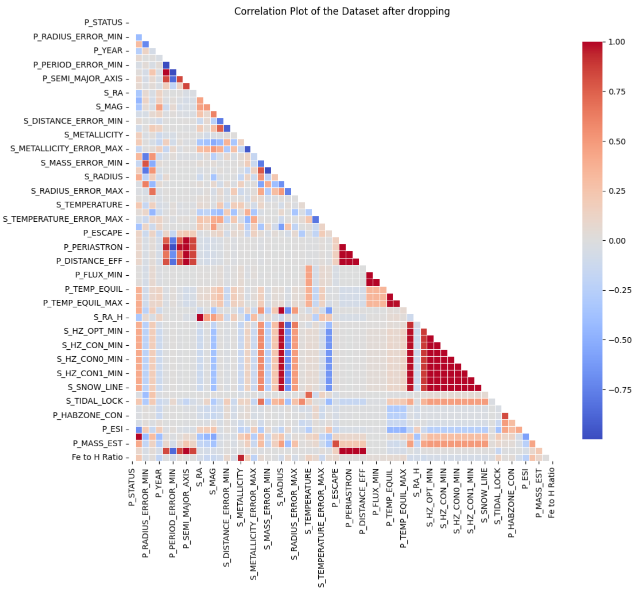
```
upper_triangle = corr_mat.where(np.triu(np.ones\
(corr_mat.shape),k = 1).astype(np.bool_))

#Set up an array of the columns to be dropped (correlation greater than 95%)
to_drop = [column for column in upper_triangle.columns if \
any(upper_triangle[column] > .95)]

#Print the list of the columns to be dropped
print(to_drop)
```

```
[ ]  df_processed=df_clean.drop(to_drop,axis=1)
```

3.2.3

Creating heatmap of features after dropping high correlation greater

```
corr_mat1=df_processed.corr()
mask = np.triu(np.ones_like(corr_mat, dtype=np.bool_))

plt.figure(figsize=(12, 10))
sns.heatmap(corr_mat, center = 0,vmax = None,annot=False, cmap='coolwarm', fmt='.2f', linewidths=0.5, mask=mask,cbar_kws = {"shrink": 0.9})

# Add a title
plt.title('Correlation Plot of the Dataset after dropping')

# Show the plot
plt.show()
```



3.3

We have to divide the process for objects and non-objects.

Objects: Check how many NaN values are in each of the columns.

```
count = object_cols.isnull().sum().sort_values(ascending = False)
percent = ((object_cols.isnull().sum()/object_cols.isnull().count())*\
100).sort_values(ascending = False)
missing = pd.concat([count, percent], axis = 1, keys = ['Count', '%'])
missing
```

	Count	%
P_TYPE_TEMP	327	8.078063
S_TYPE_TEMP	136	3.359684
P_TYPE	17	0.419960
P_NAME	0	0.000000
P_UPDATED	0	0.000000
P_DETECTION	0	0.000000
S_NAME	0	0.000000
S_ALT_NAMES	0	0.000000
S_RA_T	0	0.000000
S_DEC_T	0	0.000000
S_CONSTELLATION	0	0.000000

Filling the null values with the mode(most frequent) in each column.

```
df_processed['P_TYPE_TEMP'] = df_processed['P_TYPE_TEMP']\
.fillna(df_processed['P_TYPE_TEMP'].mode()[0])
df_processed['S_TYPE_TEMP'] = df_processed['S_TYPE_TEMP'].\
fillna(df_processed['S_TYPE_TEMP'].mode()[0])
df_processed['P_TYPE'] = df_processed['P_TYPE'].fillna\
(df_processed['P_TYPE'].mode()[0])
```

Non-Objects: We use MICE (Multivariate Imputation by Chained Equations) to fill the missing values. MICE makes an educated guess about the true values of missing data by looking at other data samples.

```
#Impute in the missing data with MICE
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

#Deep copy current dataset into Mice_temp_data
imputed_data = df_corr.copy(deep = True)

#Set the estimator to estimate features based on other features
mice_imputer = IterativeImputer()

#Fit-transform the imputed columns in the dataset
imputed_data.iloc[:, :] = mice_imputer.fit_transform(df_corr)

imputed_data
```

4. Habitability classification:

4.1

Firstly we use label encoding to convert the object variables into int counterparts.

```
[ ] #Convert categorical values to numeric values
    from sklearn.preprocessing import LabelEncoder

    #Define a dictionaryfor encoded labels
    label_encoder = LabelEncoder()
    #Encode each member of encoders dictionary
    for column in object_cols.columns:
        if object_cols[column].dtype == 'object':
            object_cols[column] = label_encoder.fit_transform(object_cols[column])
```

Concatenating the labeled data to the non-object data to get df_final.

```
[ ] df_final=pd.concat([imputed_data,object_cols],axis=1)
    df_final
```

A few of the df_final elements are given below.

	P_STATUS	P_RADIUS	P_RADIUS_ERROR_MIN	P_RADIUS_ERROR_MAX	P_YEAR	P_PERIOD	P_PERIOD_ERROR_MIN	P_PERIOD_ERROR_MAX	P_SEMI_MAJOR_AXIS	P_ANGULAR_DISTANCE	...	S_NAME	S_ALT_
0	3.0	4.794721	-0.433851	0.608203	2007.0	326.030000	-0.3200	0.3200	1.29000	13.8	...	0	
1	3.0	4.794721	-0.433851	0.608203	2009.0	516.219970	-3.2000	3.2000	1.53000	12.2	...	1	
2	3.0	4.794721	-0.433851	0.608203	2008.0	185.840000	-0.2300	0.2300	0.83000	11.0	...	2	
3	3.0	4.794721	-0.433851	0.608203	2002.0	1773.400000	-2.5000	2.5000	2.93000	163.0	...	3	
4	3.0	4.794721	-0.433851	0.608203	1996.0	798.500000	-1.0000	1.0000	1.66000	78.5	...	4	
...	...	...	...	...	...	...	...	...	...	...	...	...	
4043	3.0	1.870000	0.450000	0.200000	2019.0	28.165600	0.0028	0.0027	0.13456	0.0	...	928	
4044	3.0	2.760000	-0.433851	0.608203	2019.0	7.906961	0.0000	0.0000	0.05769	0.0	...	928	
4045	3.0	4.794721	-0.433851	0.608203	2019.0	3.204000	0.0010	0.0010	0.02100	0.0	...	107	

Sampling

We observed in question 1.5 that there is an imbalance in the dataset. Hence we do sampling.

```
[ ] from collections import Counter
    counter_ = Counter(df_final['P_HABITABLE'])
    for class_label_, example_num_ in counter_.items():
        percentage_ = example_num_ / len(df_final['P_HABITABLE']) * 100
        print('Class=%d, n=%d (%.3f%%)' % (class_label_, example_num_, percentage_))

    Class=0, n=3993 (98.641%)
    Class=2, n=34 (0.840%)
    Class=1, n=21 (0.519%)
```

We use the SMOTEENN module to achieve the required results.

```python
#Resolve the imbalance
from imblearn.combine import SMOTEENN

#Split the dataset
X, y = df_final.drop(['P_HABITABLE'], axis = 1), df_final.P_HABITABLE

#Apply sampling method and fit the resampled into data
smt = SMOTEENN(random_state=0)
X, y = smt.fit_resample(X, y)

#The distribution after applying SMOTEENN
from collections import Counter
counter = Counter(y)
for class_label, example_num in counter.items():
    percentage = example_num / len(y) * 100
    print('Class=%d, n=%d (%.3f%%)' % (class_label, example_num, percentage))
```

```
Class=0, n=3825 (32.847%)
Class=1, n=3992 (34.281%)
Class=2, n=3828 (32.872%)
```

Feature selection:

Currently we have 76 columns consisting of 75 features and one target column.
Now, not all of these features are as important as others. Selecting the most important features gives a better estimation of the model and also solves the overfitting problem.

Here we have used the `RandomForestClassifier` and `AdaBoostRegressor` to find the most important features and took the union of them. We are now left with 17 features.

```python
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier as rf

#Use split data: feature_mat and target
estimator = rf(n_estimators = 1000, random_state = 0)
selector = SelectFromModel(estimator)
selector.fit(X,y)
#Display which columns are selected
status = selector.get_support()
print("Status: ", status)

#Display selected column list
features = X.loc[:, status].columns.tolist()
print(features)

#Disply the importances
print(rf(n_estimators = 1000, random_state = 0).fit(X,y).feature_importances_)
```

```python
#Feature Selection using AdaBoost
from sklearn.ensemble import AdaBoostRegressor as Ada

#Use split data: feature_mat and target
estimator = Ada(random_state = 0, n_estimators = 50)
selector = SelectFromModel(estimator)
selector.fit(X,y)
#Display which columns are selected
status = selector.get_support()
print("Status: ", status)

#Display selected column list
features = X.loc[:, status].columns.tolist()
print(features)

#Disply the importances
print(estimator.fit(X,y).feature_importances_)
```

Only keeping the required fields.

```
#The feature_mat has to consist of only the features
#we have selected in Feature Selection phase
feature = X[['S_MASS', 'S_TEMPERATURE', 'P_FLUX', 'P_FLUX_MIN', 'P_FLUX_MAX', 'P_TEMP_EQUIL','S_LOG_G', 'P_PERIASTRON', 'P_TEMP_EQUIL_MIN', 'P_TEMP_EQUIL_MAX', 'S_ABIO_ZONE', 'P_HABZC

#The target column to test with
target = y
```

Min-Max scales the features to bring them to the same range.

```
[ ]  #Normalize the training set
     from sklearn.preprocessing import MinMaxScaler

     scaler = MinMaxScaler()
     X_train = scaler.fit_transform(feature)
```

MODEL TRAINING

K-NEAREST NEIGHBOURS

We start with applying the **K-Nearest neighbors** model with Kfold to obtain the following result

```python
from sklearn.model_selection import KFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X = pd.DataFrame(scaler.fit_transform(feature))

# Values of K to loop through (2-10)
k_values = range(2, 11)

# Loop through different values of K
for k in k_values:
    # Initialize K-Fold cross-validator with the current value of K
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # List to store accuracy for each fold
    fold_accuracies = []

    # Iterate through the folds
    for train_index, test_index in kf.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Create a KNN classifier
        knn = KNeighborsClassifier(n_neighbors=int(11645/3))  # You can adjust the value of k

        # Train the KNN classifier
        knn.fit(X_train, y_train)

        # Make predictions on the test data
        y_pred = knn.predict(X_test)

        # Calculate accuracy for this fold and append it to the fold_accuracies list
        fold_accuracy = accuracy_score(y_test, y_pred)
        fold_accuracies.append(fold_accuracy)

    # Calculate the mean accuracy for this K value
    mean_accuracy = np.mean(fold_accuracies)

    # Print the mean accuracy for each K-fold
    print(f"K-Fold={k}, Mean Accuracy={mean_accuracy}")
```

```
K-Fold=2, Mean Accuracy=0.36900074586657006
K-Fold=3, Mean Accuracy=0.7878941485317333
K-Fold=4, Mean Accuracy=0.8656927493667446
K-Fold=5, Mean Accuracy=0.8794332331472734
K-Fold=6, Mean Accuracy=0.8908541669986244
K-Fold=7, Mean Accuracy=0.8933446852305843
K-Fold=8, Mean Accuracy=0.8960063937351308
K-Fold=9, Mean Accuracy=0.9013296872324976
K-Fold=10, Mean Accuracy=0.903821143607215
```

DECISION TREE CLASSIFIER:

(answer to **4.3**)

We use the **Grid search method** on a training dataset to get the best hyperparameters that give the best result on the result dataset.

```
[ ] X_train, X_test, y_train, y_test = train_test_split(feature, target,test_size = 0.33, random_state = 42)
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

#Find the best parameters for Decision Tree using GridSearchCV
#Hyperparameters' range
param_grid = {'max_depth': np.arange(2, 10, 1),
              'max_leaf_nodes': np.arange(2, 100, 10),
              'random_state': [0, 1, 2, 3, 4, 5],
              'splitter': ['best', 'random']}

#Conduct the Grid Search
grid_search = GridSearchCV(DecisionTreeClassifier(),
                           param_grid = param_grid,
                           refit = True, verbose = 0)
#Fit the dataset
grid_search.fit(X_train, y_train)

# print best parameter after tuning
print(f"Best Parameters: {grid_search.best_params_}")
```

```
Best Parameters: {'max_depth': 8, 'max_leaf_nodes': 22, 'random_state': 0, 'splitter': 'random'}
```

Creating a function to create a Confusion matrix

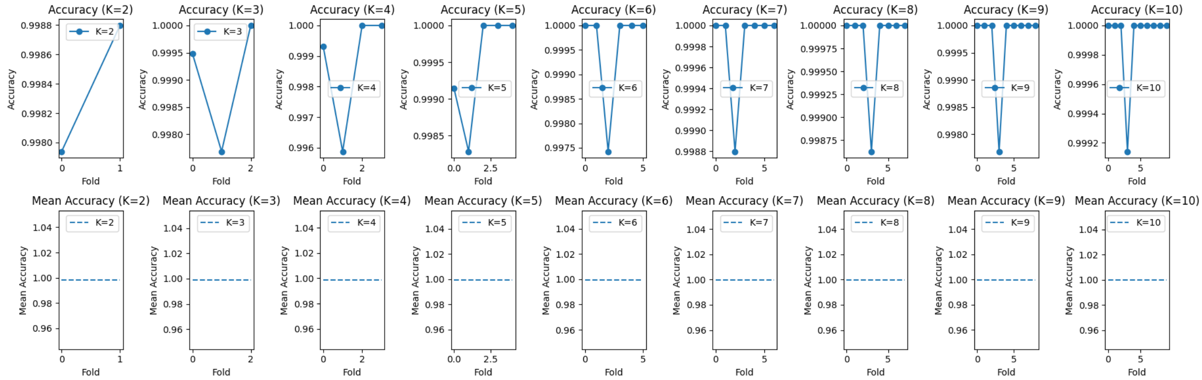
```
from sklearn.metrics import confusion_matrix

def plot_confusion_mat(ytest, ypred):
    #Create  a confusion matrix, which compares the y_test and prediction made
    conf_mat = confusion_matrix(ytest, ypred)

    #Create a dataframe for a array-formatted Confusion matrix,so it will be easy for plotting.
    #Assign corresponding names to labels
    confusion_mat_df = pd.DataFrame(conf_mat,
                    index = ['Inhabitable', 'Consevatively Habitable','Optimistically Habitable'],
                    columns = ['Inhabitable', 'Consevatively Habitable','Optimistically Habitable'])

    #Plot the confusion matrix
    plt.figure(figsize = (2,1))
    sns.heatmap(confusion_mat_df, annot = True)
    plt.title('Habitability Confusion Matrix')
    plt.ylabel('Actal Values')
    plt.xlabel('Predicted Values')
    plt.show()

    return conf_mat
```

Applying k-fold validation from 2 to 10 times

```python
from sklearn.metrics import accuracy_score

# Define a function to create and train the Decision Tree model
def train_decision_tree(X_train, y_train, max_depth=None, max_leaf_nodes=None):
    clf = DecisionTreeClassifier(max_depth=max_depth, max_leaf_nodes=max_leaf_nodes, random_state=1, splitter='random')
    clf.fit(X_train, y_train)
    return clf

# Values of K to test
k_values = range(2, 11)

# Create subplots for loss and accuracy
fig, axes = plt.subplots(nrows=2, ncols=len(k_values), figsize=(18, 6))

# Perform K-Fold Cross-Validation for different K values
for i, k in enumerate(k_values):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    fold_losses = []
    fold_accuracies = []

    for train_index, val_index in kf.split(feature):
        X_train, X_val = feature[train_index], feature[val_index]
        y_train, y_val = y[train_index], y[val_index]

        # Train the Decision Tree model
        clf = train_decision_tree(X_train, y_train,7,12)
```

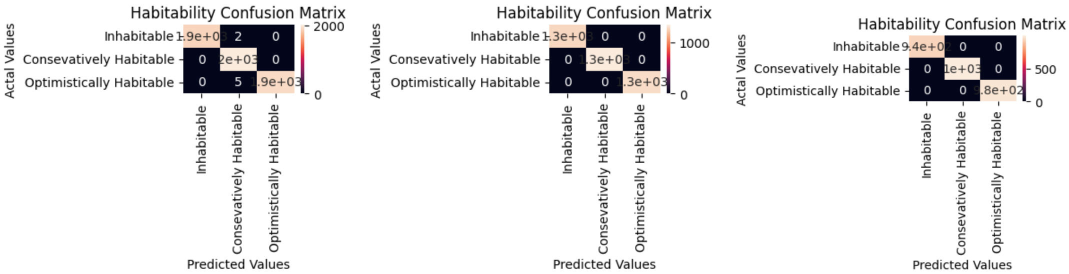
```python
        # Make predictions on the validation set
        y_pred = clf.predict(X_val)

        accuracy = accuracy_score(y_val, y_pred)
        fold_accuracies.append(accuracy)

    # Plot accuracy curves
    axes[0, i].plot(range(k), fold_accuracies, marker='o', label=f'K={k}')
    axes[0, i].set_title(f'Accuracy (K={k})')
    axes[0, i].set_xlabel('Fold')
    axes[0, i].set_ylabel('Accuracy')
    axes[0, i].legend()

    # Plot mean accuracy across folds
    axes[1, i].plot(range(k), [np.mean(fold_accuracies)] * k, linestyle='--', label=f'K={k}')
    axes[1, i].set_title(f'Mean Accuracy (K={k})')
    axes[1, i].set_xlabel('Fold')
    axes[1, i].set_ylabel('Mean Accuracy')
    axes[1, i].legend()


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

We get the following result

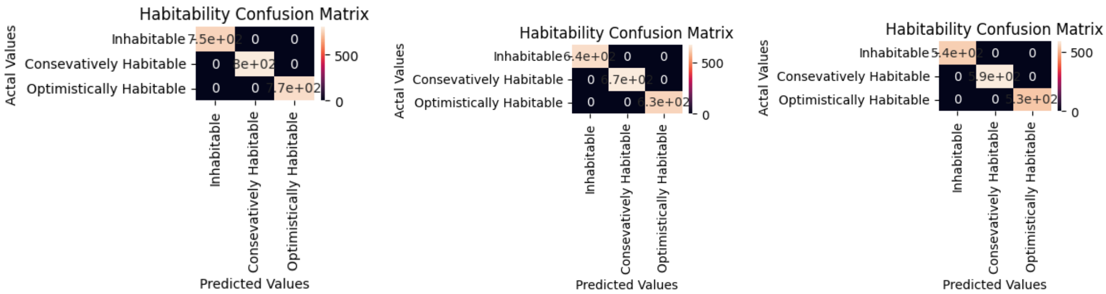


We can see that the mean accuracy comes out to be 1 using the DecisionTreeClassifier

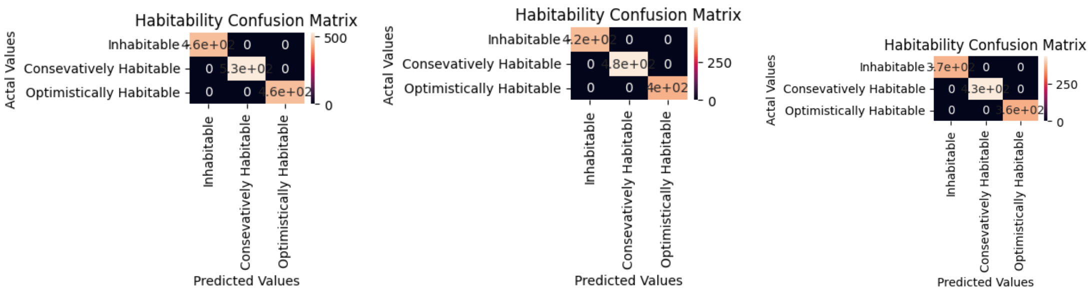
Printing the Confusion matrix for each k-value prediction



2,3,4



5,6,7



8,9,10

NEURAL NETWORK

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from tensorflow import keras
from tensorflow.keras import layers

# Define a function to create and train the Neural Network model
def train_neural_network(X_train, y_train, num_epochs=50):
    model = keras.Sequential([
        layers.Input(shape=(X_train.shape[1],)),
        layers.Dense(64, activation='relu'),
        layers.Dense(1, activation='sigmoid')  # Binary classification, adjust output units as needed
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=num_epochs, verbose=0)

    return model, history

# Values of K to test
k_values = range(2, 11)

# Create subplots for loss and accuracy versus epoch
fig, axes = plt.subplots(nrows=2, ncols=len(k_values), figsize=(25, 6))
```

```python
# Perform K-Fold Cross-Validation for different K values
for i, k in enumerate(k_values):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    fold_losses = []
    fold_accuracies = []

    for train_index, val_index in kf.split(feature):
        X_train, X_val = feature[train_index], feature[val_index]
        y_train, y_val = y[train_index], y[val_index]

        # Train the Neural Network model
        model, history = train_neural_network(X_train, y_train, num_epochs=50)

        # Record loss and accuracy at each epoch
        fold_losses.append(history.history['loss'])
        fold_accuracies.append(history.history['accuracy'])

    # Calculate mean loss and accuracy across folds
    mean_loss = np.mean(fold_losses, axis=0)
    mean_accuracy = np.mean(fold_accuracies, axis=0)

    # Plot loss versus epoch
    axes[0, i].plot(range(1, len(mean_loss) + 1), mean_loss, label=f'K={k}')
    axes[0, i].set_title(f'Loss vs. Epochs (K={k})')
    axes[0, i].set_xlabel('Epochs')
    axes[0, i].set_ylabel('Loss')
    axes[0, i].legend()
```

```python
    # Plot accuracy versus epoch
    axes[1, i].plot(range(1, len(mean_accuracy) + 1), mean_accuracy, label=f'K={k}')
    axes[1, i].set_title(f'Accuracy vs. Epochs (K={k})')
    axes[1, i].set_xlabel('Epochs')
    axes[1, i].set_ylabel('Accuracy')
    axes[1, i].legend()

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

Output: loss and accuracy vs epoch plots for different k values