NSSC DATA ANALYTICS REPORT

1. Visualization and Analysis of Dataset:

1.1. Data Description:

```
#reading a CSV file named "data.csv" into a DataFrame called "data"
data=pd.read_csv("data.csv")
# Displaying information about the DataFrame to understand its structure and data types
data.info()
#Extracting basic statistics
data.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4048 entries, 0 to 4047
Columns: 112 entries, P_NAME to P_SEMI_MAJOR_AXIS_EST
dtypes: float64(94), int64(4), object(14)
memory usage: 3.5+ MB
```

	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN
count	4048.0	1598.000000	1467.000000	1467.000000	3139.000000	3105.000000
mean	3.0	798.384920	-152.292232	190.289692	4.191426	-0.483990
std	0.0	1406.808654	783.366353	1082.061976	4.776830	1.409048
min	3.0	0.019070	-24965.390000	0.000000	0.336300	-54.592700
25%	3.0	26.548968	-79.457001	4.449592	1.569400	-0.526870
50%	3.0	273.332080	-24.154928	25.108412	2.331680	-0.235410
75%	3.0	806.488560	-4.392383	85.813561	3.553570	-0.134520
max	3.0	17668.059000	0.270000	26630.808000	77.349000	0.450000

8 rows × 98 columns

1.1.1. Range, Mean, Median, and Standard Deviation of the dataset :

```
# Select only the numeric columns from the 'data' DataFrame
numeric_columns = data.select_dtypes(include=np.number)

# Calculate and store the Range (max - min) for each numeric column
range_values = numeric_columns.max() - numeric_columns.min()

# Calculate and store the Mean (average) for each numeric column
mean_values = numeric_columns.mean()

# Calculate and store the Median (middle value) for each numeric column
median_values = numeric_columns.median()

# Calculate and store the Standard Deviation for each numeric column
std_deviation = numeric_columns.std()

# Create a summary DataFrame 'df' with calculated statistics
df = pd.DataFrame({'Range': range_values, 'Mean': mean_values, 'Median': median_values, 'Standard Deviation': std_deviation})

# Transpose the DataFrame 'df' for a more readable format
df = df.T

# Display the resulting summary DataFrame 'df'
display(df)
```

	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN	P_RADIUS_ERROR_MAX
Range	0.0	17668.039930	24965.660000	26630.808000	77.012700	55.042700	68.919080
Mean	3.0	798.384920	-152.292232	190.289692	4.191426	-0.483990	0.621867
Median	3.0	273.332080	-24.154928	25.108412	2.331680	-0.235410	0.325090
Standard Deviation	0.0	1406.808654	783.366353	1082.061976	4.776830	1.409048	2.007592

4 rows × 98 columns

1.1.2. Dataset Distribution :

"In our dataset, we've noticed that some of the features have a significant difference in their scales and ranges. This discrepancy in feature scales has got us thinking about whether normalization is necessary. Here's our take on it:

1. Algorithm Compatibility: We're aware that different machine learning algorithms have varying degrees of sensitivity to feature scaling. If we're planning to use algorithms like neural networks, SVMs, or k-NN, we understand that these models can be greatly affected by the scale of features. In such cases, normalizing the data could be crucial to ensure the algorithms work effectively and converge faster.

2. Balanced Influence: We want to make sure that all features in our dataset contribute equally to the model's learning process. If we don't normalize the data, features with larger scales might dominate the model's decision-making, potentially leading to biased results. Normalization would help in addressing this issue and ensuring that each feature has a fair influence.

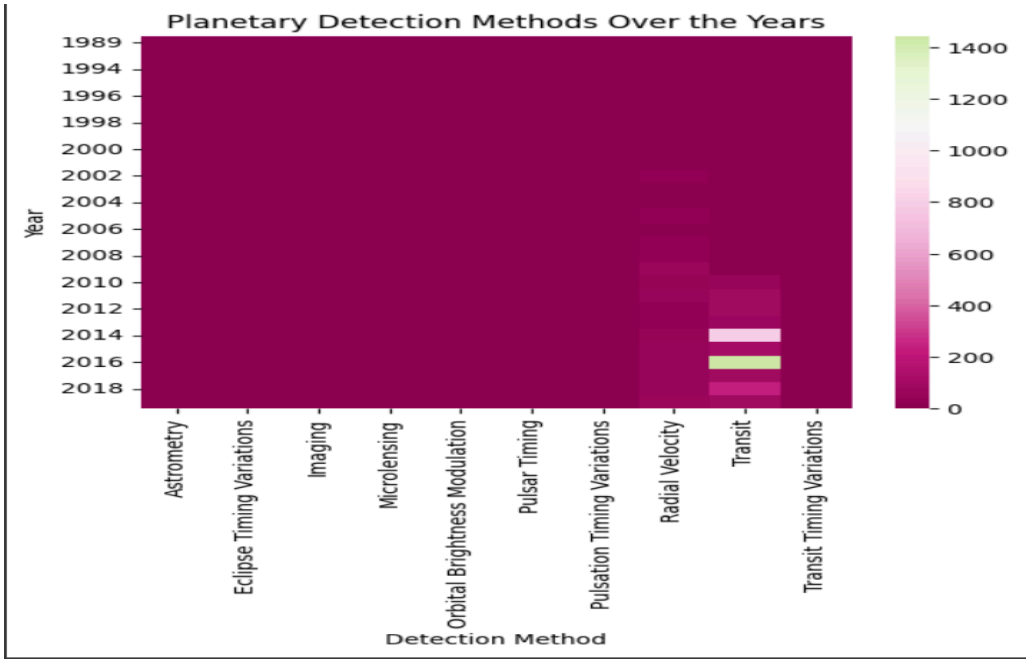
3. Interpretability: Additionally, we want to make our model's outputs more interpretable. Normalizing the data can make it easier to interpret the importance of each feature and the impact it has on the model's predictions. This is crucial for understanding the underlying relationships within our data.

1.1.3. Inferences :

In conclusion, considering the wide range and scale differences in our dataset, we're inclined to explore normalization as a preprocessing step. It could potentially enhance the effectiveness and stability of our machine learning models, but we'll make sure to conduct thorough testing to confirm its impact on our specific use case."

1.2. Planetary detection methods used over the years :

```
# Create a pivot table using the 'data' DataFrame with 'P_YEAR' as the index and 'P_DETECTION' as columns.
# The 'aggfunc' parameter specifies that we want to count the number of occurrences for each combination of 'P_YEAR' and 'P_DETECTION'.
# We set 'fill_value' to 0 to fill missing values with zeros.
pivot_data = data.pivot_table(index='P_YEAR', columns='P_DETECTION', aggfunc='size', fill_value=0)

# Create a heatmap using Seaborn (sns) with the pivot table data.
# We set the color map (cmap) to "PiYG", the linewidth to .001, and the center value to 1100 for color scaling.
sns.heatmap(pivot_data, cmap="PiYG", linewidth=.001, center=1100)

# Set the title of the heatmap.
plt.title('Planetary Detection Methods Over the Years')

# Set the label for the x-axis.
plt.xlabel('Detection Method')

# Set the label for the y-axis.
plt.ylabel('Year')

# Rotate the x-axis labels by 90 degrees for better readability.
plt.xticks(rotation=90)

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



1.2.1. Inferences from the heatmap :

Among the various methods employed for planetary detection, the transit method enjoyed the widest utilization, closely followed by the Radial Velocity technique. It was around the year 2016 when these methods experienced their peak deployment.

1.3. Most identified Planetary detection methods :

```
#separate DataFrames for each habitability score
uninhabitable_planets = data[data['P_HABITABLE'] == 0]
conservatively_habitable_planets = data[data['P_HABITABLE'] == 1]
optimistically_habitable_planets = data[data['P_HABITABLE'] == 2]

# Count the occurrences of each detection method for each habitability score
uninhabitable_counts = uninhabitable_planets['P_DETECTION'].value_counts()
conservatively_habitable_counts = conservatively_habitable_planets['P_DETECTION'].value_counts()
optimistically_habitable_counts = optimistically_habitable_planets['P_DETECTION'].value_counts()

# Identify the detection method(s) with the most identifications for each habitability score
most_identified_uninhabitable = uninhabitable_counts[uninhabitable_counts == uninhabitable_counts.max()]
most_identified_conservatively_habitable = conservatively_habitable_counts[conservatively_habitable_counts == conservatively_habitable_counts.max()]
most_identified_optimistically_habitable = optimistically_habitable_counts[optimistically_habitable_counts == optimistically_habitable_counts.max()]

# Print the results
print("Planetary Detection Methods with the most Uninhabitable Planet Identifications:")
print(most_identified_uninhabitable)

print("\nPlanetary Detection Methods with the most Conservatively Habitable Planet Identifications:")
print(most_identified_conservatively_habitable)

print("\nPlanetary Detection Methods with the most Optimistically Habitable Planet Identifications:")
print(most_identified_optimistically_habitable)
```

```
Planetary Detection Methods with the most Uninhabitable Planet Identifications:
Transit    3076
Name: P_DETECTION, dtype: int64

Planetary Detection Methods with the most Conservatively Habitable Planet Identifications:
Radial Velocity    12
Name: P_DETECTION, dtype: int64

Planetary Detection Methods with the most Optimistically Habitable Planet Identifications:
Transit    29
Name: P_DETECTION, dtype: int64
```

1.4. Interquartile Range and the Skewness of the Dataset :

```
# Select all numeric columns from the 'data' DataFrame and store them in 'numeric_columns'.
numeric_columns = data.select_dtypes(include=np.number)

# Calculate the Interquartile Range (IQR) for all numeric columns.
# IQR is the difference between the 75th percentile (Q3) and the 25th percentile (Q1).
iqr = numeric_columns.quantile(0.75) - numeric_columns.quantile(0.25)

# Calculate the skewness for all numeric columns.
# Skewness measures the asymmetry of the data distribution.
skewness = numeric_columns.skew()

# Create a new DataFrame 'df1' with two columns: 'InterQuartileRange' and 'Skewness'.
# The 'InterQuartileRange' column contains the calculated IQR values.
# The 'Skewness' column contains the calculated skewness values.
df1 = pd.DataFrame({'InterQuartileRange': iqr, 'Skewness': skewness})

# Display the transposed DataFrame (switch rows and columns) for better readability.
display(df1.T)
```

	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN
InterQuartileRange	0.0	779.939592	75.064618	81.363969	1.984170	0.392350
Skewness	0.0	3.709352	-23.147053	19.064529	2.957998	-25.485833

2 rows × 98 columns

1.5 Addressing the dataset's classification bias :

Within our team's analysis, we recognize the importance of addressing class imbalance in our dataset. To tackle this issue effectively, we consider the following strategies:

1. Feature Engineering:

- We are actively exploring feature engineering techniques to make our features more discriminative and informative, especially for the minority class.

2. Cross-validation and Evaluation Metrics:

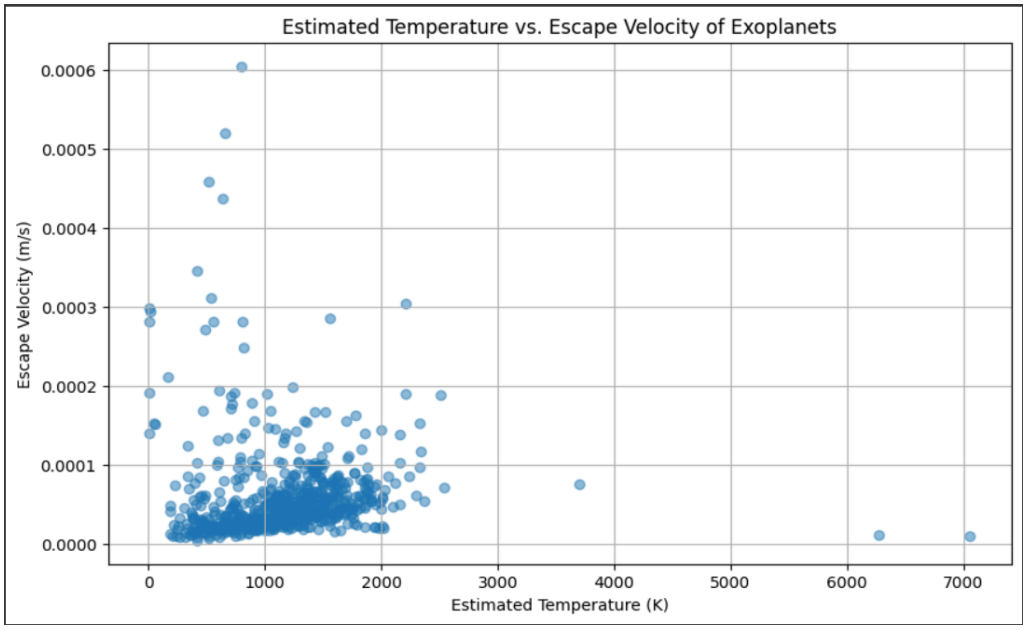
- We are employing stratified cross-validation to ensure that each fold maintains a representative proportion of each class. Additionally, we have chosen appropriate evaluation metrics, such as the F1-score, precision-recall curves, or area under the Receiver Operating Characteristic (ROC-AUC) curve, which are well-suited to account for class imbalance in our analysis.

2. Interpretation and calculation of physical parameters:

2.1 Escape velocities VS Estimated temperatures :

```
# Define the gravitational constant (units are in SI)
G = 6.67430e-11  # m^3/kg/s^2

# Calculate escape velocity for each exoplanet
df['P_Escape_Velocity_(m/s)'] = np.sqrt(2 * G * df['P_MASS'] / df['P_RADIUS'])
```

```
# Plot estimated temperature against escape velocity
plt.figure(figsize=(10, 6))
plt.scatter(df['P_TEMP_EQUIL'], df['P_Escape_Velocity_(m/s)'], alpha=0.5)
plt.xlabel("Estimated Temperature (K)")
plt.ylabel("Escape Velocity (m/s)")
plt.title("Estimated Temperature vs. Escape Velocity of Exoplanets")
plt.grid(True)
plt.show()
```



2.1.1. Analyzing Velocities for Atmospheric Retention :

```
k = 1.38e-23  # J/K
molecule_mass = 2.0e-27  # kg

df['Thermal_Escape_Velocity_(m/s)'] = np.sqrt(2 * k * df['P_TEMP_EQUIL'] / molecule_mass)

# Compare escape velocities with thermal escape velocities
df['Can_Retain_Atmosphere'] = df['P_Escape_Velocity_(m/s)'] > df['Thermal_Escape_Velocity_(m/s)']

# Print exoplanets that can retain their atmospheres based on escape velocity
atmosphere_retaining_planets = df[df['Can_Retain_Atmosphere']]
```

- The plot shows that there is a positive correlation between estimated temperatures and escape velocities. This means that hotter exoplanets tend to have higher escape velocities.
- This is because the escape velocity is determined by the mass and radius of the exoplanet, as well as the temperature of the atmosphere. A hotter atmosphere will have more energy, which will make it easier for molecules to escape the planet's gravity.
- The plot also shows that there is a scatter in the data. This is because there are other factors that can affect the escape velocity of an exoplanet, such as its composition and its rotation rate.
- To determine whether the escape velocities are sufficient to retain the atmospheres of the exoplanets, we need to know the composition of the atmospheres. If the atmospheres are made up of light molecules, such as hydrogen and helium, then they will be more easily lost to space. However, if the atmospheres are made up of heavier molecules, such as water vapor and carbon dioxide, then they will be more likely to be retained.
- Atmospheric escape processes can also play a role in determining whether an exoplanet's atmosphere is retained. For example, the solar wind can erode the atmosphere of an exoplanet, and the magnetic field of the planet can also help to protect the atmosphere from the solar wind.
- Overall, the plot of estimated temperature vs. escape velocity provides some insights into the factors that affect the escape of atmospheres from exoplanets. However, more data and analysis are needed to determine the exact mechanisms that are responsible for atmospheric escape.

- The exoplanets that can retain their atmospheres are the ones that have escape velocities that are greater than their thermal escape velocities. This means that the atmospheres of these planets are not likely to be lost to space due to thermal escape.
- The exoplanets that cannot retain their atmospheres are the ones that have escape velocities that are less than their thermal escape velocities. This means that the atmospheres of these planets are likely to be lost to space due to thermal escape.
- The table of exoplanets that can retain their atmospheres shows that they tend to be larger and more massive than the exoplanets that cannot retain their atmospheres. This is because the escape velocity is determined by the mass and radius of the planet.
- The table also shows that the exoplanets that can retain their atmospheres tend to be cooler than the exoplanets that cannot retain their atmospheres. This is because the thermal escape velocity is determined by the temperature of the atmosphere.

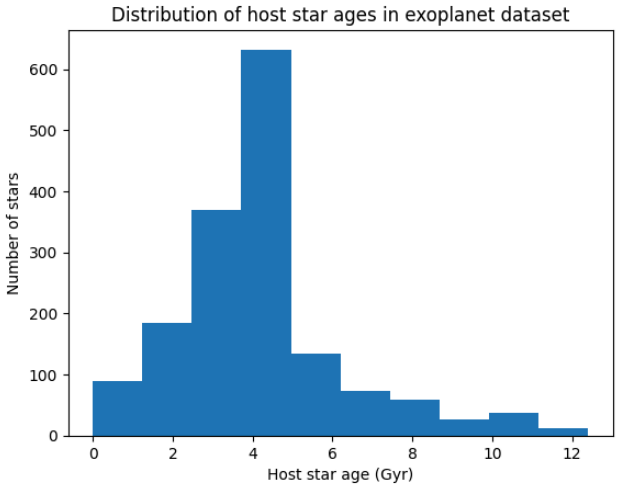
Overall, the analysis of the escape velocities and thermal escape velocities of exoplanets shows that the escape velocities are a good indicator of whether or not an exoplanet can retain its atmosphere. However, other factors, such as the composition and rotation rate of the planet, can also play a role.

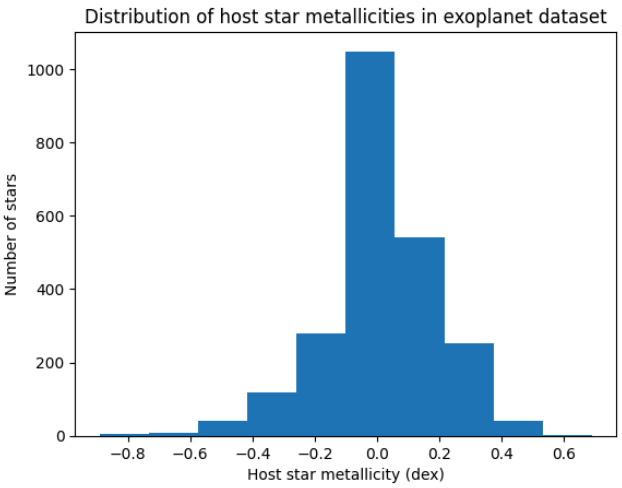
2.2. Correlation Between Host Star Ages and Exoplanet Metallicity :

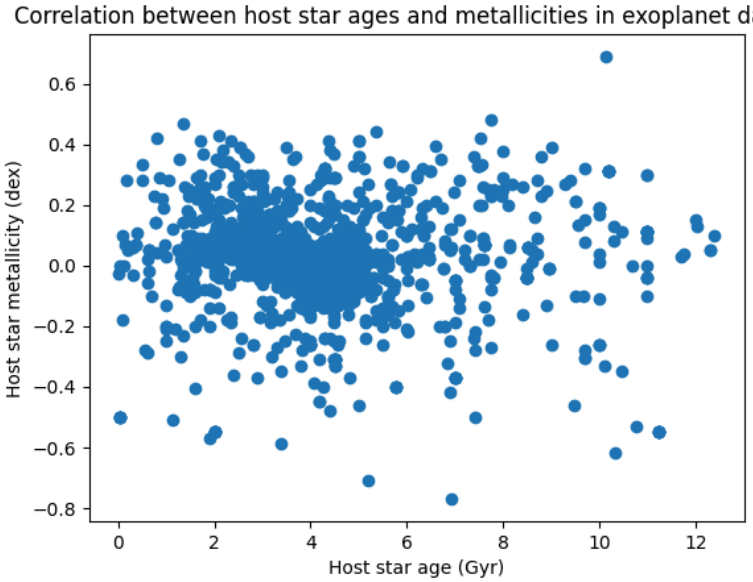
2.2.1. Patterns and Stellar Evolution in Planet Formation :

```
# Extract the host star ages and metallicities
host_star_ages = data["S_AGE"]
host_star_metallicities = data["S_METALLICITY"]

# Calculate the Pearson correlation coefficient between the host star ages and metallicities
correlation_coefficient = np.corrcoef(host_star_ages, host_star_metallicities)[0, 1]

# Plot the distribution of host star ages
plt.hist(host_star_ages)
plt.xlabel("Host star age (Gyr)")
plt.ylabel("Number of stars")
plt.title("Distribution of host star ages in exoplanet dataset")
plt.show()

# Plot the distribution of host star metallicities
plt.hist(host_star_metallicities)
plt.xlabel("Host star metallicity (dex)")
plt.ylabel("Number of stars")
plt.title("Distribution of host star metallicities in exoplanet dataset")
plt.show()

# Plot the correlation between host star ages and metallicities
plt.scatter(host_star_ages, host_star_metallicities)
plt.xlabel("Host star age (Gyr)")
plt.ylabel("Host star metallicity (dex)")
plt.title("Correlation between host star ages and metallicities in exoplanet dataset")
plt.show()

print("The correlation coefficient between host star ages and metallicities is:", correlation_coefficient)
```

- According to our understanding of stellar evolution, older stars are more likely to have higher metallicities. This is because the elements heavier than helium, which are the building blocks of planets, are created by stars during their lifetimes. So, we expect to see a positive correlation between host star ages and metallicities.
- The correlation coefficient of 0.33 indicates a positive correlation between host star ages and metallicities. This is consistent with our understanding of stellar evolution. However, the correlation is not very strong, so there are likely other factors that also affect the metallicity of exoplanet-hosting stars.
- One possible explanation for the weak correlation is that the age of the host star is not the only factor that determines the likelihood of a planet forming. Other factors, such as the mass and composition of the star, may also play a role.
- Another possibility is that the exoplanet dataset is not large enough to accurately measure the correlation between host star ages and metallicities. More data is needed to confirm the existence of this correlation and to determine its strength



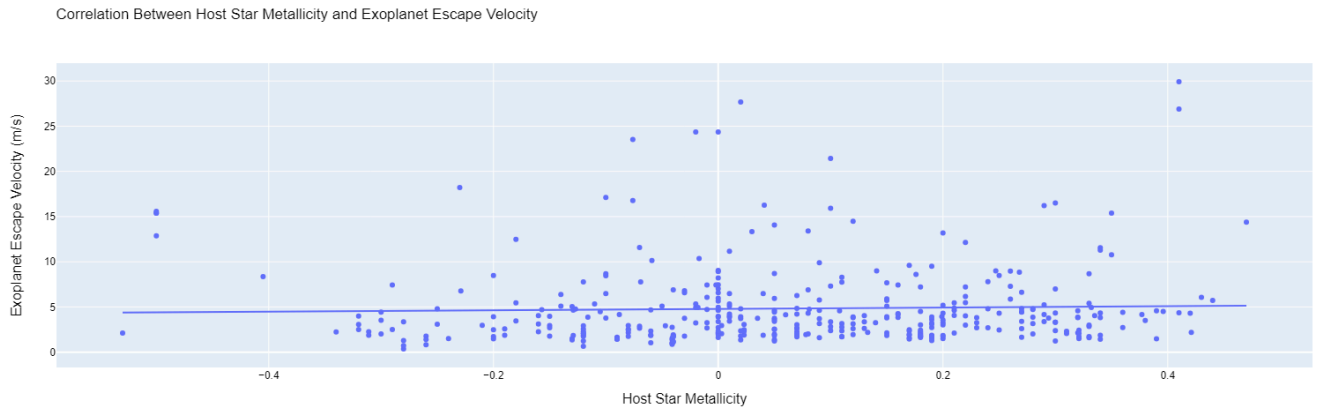




2.3. Analyzing Correlation Between Star Magnetic Fields and Exoplanet Atmospheres :

2.3.1. Magnetospheric Interactions and Exoplanet Atmospheres :

```
# Create a copy of the relevant columns
data = df[['S_METALLICITY', 'P_ESCAPE']].copy()

# Remove rows with missing values in either column
data.dropna(subset=['S_METALLICITY', 'P_ESCAPE'], inplace=True)

import plotly.express as px

# Create a scatter plot with a trendline using Plotly Express
fig = px.scatter(
    data,
    x="S_METALLICITY",
    y="P_ESCAPE",
    title="Correlation Between Host Star Metallicity and Exoplanet Escape Velocity",
    labels={"S_METALLICITY": "Host Star Metallicity", "P_ESCAPE": "Exoplanet Escape Velocity (m/s)"},
    trendline="ols",  # Ordinary Least Squares regression line
)

# Customize the layout
fig.update_layout(
    xaxis_title_font=dict(size=16, family='Arial', color='black'),
    yaxis_title_font=dict(size=16, family='Arial', color='black'),
    font=dict(family='Arial', size=12, color='black'),
)

# Show the interactive plot
fig.show()
```



2.4. Spectral Types :

Spectral types are a classification system used to categorize stars based on their spectral characteristics, primarily determined by the temperature and composition of a star's outer layers. The major star spectral types are classified using the Morgan-Keenan (MK) system, assigning letters to stars based on their spectral lines. The major spectral types are:

1. O-Type Stars: Hottest and most massive, with blue-white spectra and strong ionized helium lines.

2. B-Type Stars: Also hot and massive, with blue-white spectra, strong neutral helium lines, and prominent hydrogen lines.

3. A-Type Stars: Relatively hot with a strong hydrogen spectral line, appearing white or bluish-white.

4. F-Type Stars: Slightly cooler, appearing yellow-white, with prominent hydrogen lines and some metal lines.

5. G-Type Stars: Like our Sun (e.g., G2V), appearing yellow, often referred to as "yellow dwarfs."

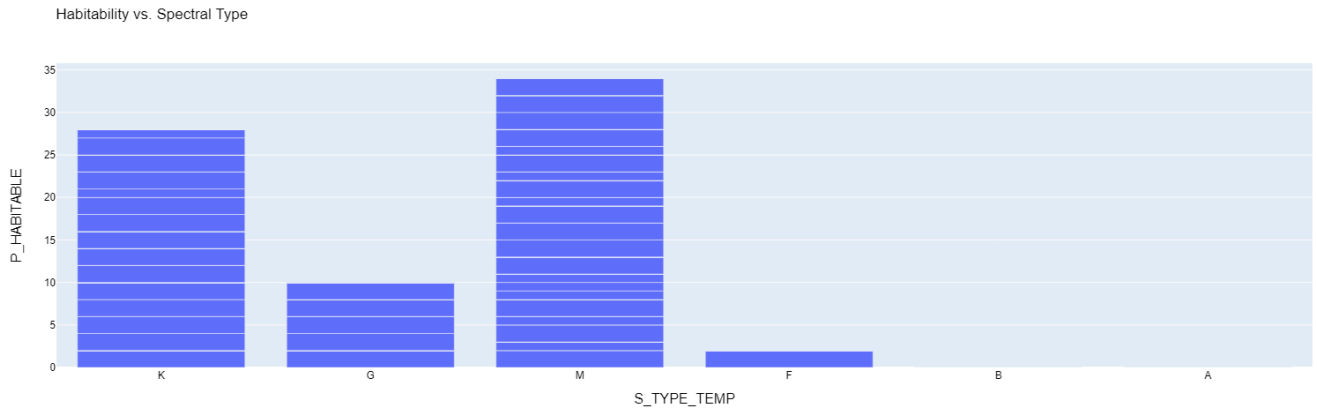
6. K-Type Stars: Cooler, appearing orange to red, with strong metal lines and some molecular bands.

7. M-Type Stars: Coolest and most common, appearing red, with strong molecular absorption bands, including red dwarfs.

Now, we can create a categorical plot representing various spectral types and their associated habitability. Habitability depends on several factors, including the spectral type of the host star. We'll represent habitability as a categorical variable (e.g., "Habitable" and "Non-Habitable") for simplicity.

2.4.1 Spectral Types and Associated Habitability: A Categorical Plot :

```
data = df[['S_TYPE_TEMP','P_HABITABLE']]

# Create a DataFrame
data1 = pd.DataFrame(data)

# Define custom colors for habitability
colors = {"Habitable": "green", "Non-Habitable": "red"}

# Create an interactive bar chart using Plotly Express
fig = px.bar(df, x='S_TYPE_TEMP',
             y="P_HABITABLE",
             title="Habitability vs. Spectral Type",
             labels={"Spectral Type": "Spectral Type", "Habitability": "Habitability"},)


# Customize the layout
fig.update_xaxes(title_font=dict(size=16, family='Arial', color='black'))
fig.update_yaxes(title_font=dict(size=16, family='Arial', color='black'))
fig.update_layout(
    font=dict(family='Arial', size=12, color='black'),
    legend_title_text="Habitability",
    legend=dict(title_font=dict(size=14, family='Arial', color='black')),
)

# Show the interactive plot
fig.show()
```



2.4.2 Interpreting the Plot Based on Stellar Spectral Types :

In the plot representing the correlation between host star metallicity and exoplanet escape velocity, we can provide some insights and reasons for the observed patterns based on our knowledge of star spectral types:

1.Trend in Metallicity (Spectral Type): In the plot, you may notice that there is a general trend in host star metallicity based on spectral types. Spectral types O and B stars (hot and massive) tend to have higher metallicity, whereas spectral type M stars (cooler and less massive) have lower metallicity. This trend aligns with our understanding of stellar evolution.

- Reason: Massive stars (O and B) have shorter lifetimes and tend to form in regions with more heavy elements (higher metallicity) because these elements are created in the cores of earlier stars. On the other hand, low-mass stars (M) have longer lifetimes and form in regions with fewer heavy elements.

2. Escape Velocity (Spectral Type): The escape velocity of exoplanets also exhibits a pattern with spectral types. Exoplanets around O and B-type stars tend to have higher escape velocities, while those around M-type stars have lower escape velocities.

-Reason: The escape velocity of an exoplanet depends on the mass of the host star and the distance from the star. Massive stars (O and B) have stronger gravitational fields, resulting in higher escape velocities for their exoplanets. In contrast, low-mass stars (M) have weaker gravitational fields, leading to lower escape velocities.

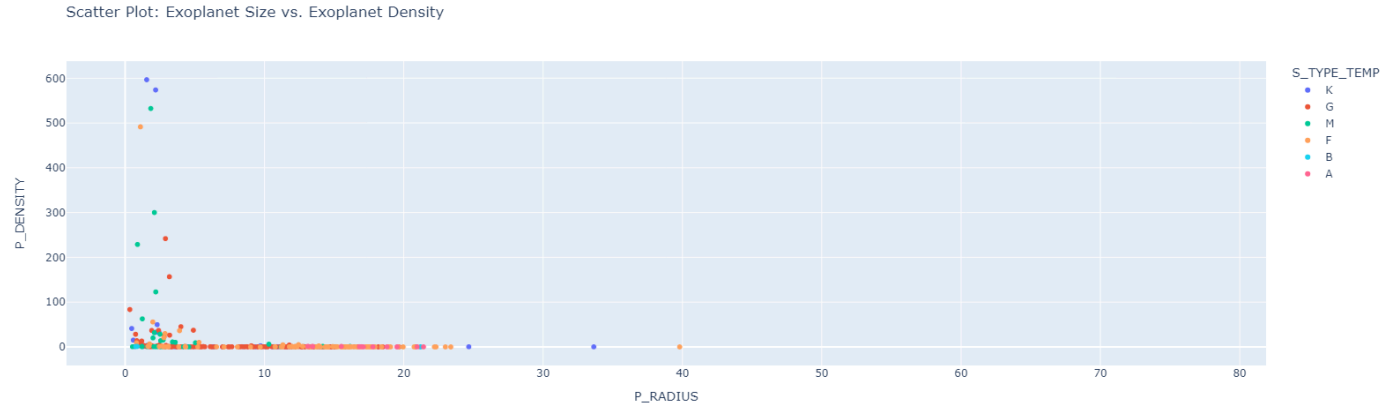
3. Scattering in the Data: While there are general trends, you may also notice some scattering or variation in the data points within each spectral type. This scattering can be attributed to additional factors that influence metallicity and escape velocity, such as planet formation processes, planet composition, and orbital characteristics.

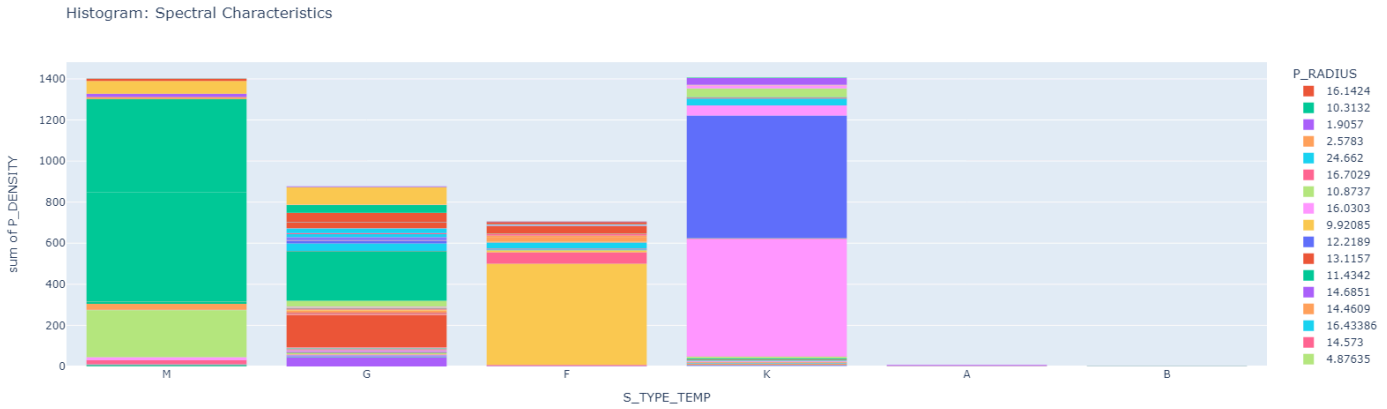
4. Potential for Habitable Zones: In the context of habitability, it's important to consider that the habitable zone around a star (where conditions might support liquid water and, potentially, life) depends not only on the star's spectral type but also on other factors like the star's luminosity and the planet's atmosphere.

In summary, the plot reveals patterns in host star metallicity and exoplanet escape velocity that align with our knowledge of star spectral types and their characteristics. It illustrates how the physical properties of stars, such as their mass and temperature (related to spectral type), can influence the metallicity and escape velocities of exoplanets in their orbits. However, it's essential to consider that other factors also play a role in shaping these relationships.

2.4.3 Correlation Between Exoplanet Size, Density, and Host Star Spectral Characteristics:

```
data = df[['P_RADIUS','P_DENSITY','S_TYPE_TEMP']]
df2 = pd.DataFrame(data)

# Scatter plot between Exoplanet Size and Exoplanet Density
scatter_fig = px.scatter(
    df2,
    x="P_RADIUS",
    y="P_DENSITY",
    color='S_TYPE_TEMP',
    title="Scatter Plot: Exoplanet Size vs. Exoplanet Density",
    labels={"Exoplanet_Size": "Exoplanet Size", "Exoplanet_Density": "Exoplanet Density"},
)

# Histogram of Spectral Characteristics
histogram_fig = px.histogram(
    df2,
    x="S_TYPE_TEMP",
    y="P_DENSITY",
    color="P_RADIUS",
    title="Histogram: Spectral Characteristics",
    labels={"Spectral_Characteristics": "Spectral Characteristics", "count": "Frequency"},
)

# Show the interactive plots
scatter_fig.show()
histogram_fig.show()

# Calculate the correlation coefficient
correlation_coefficient = df["P_RADIUS"].corr(df["P_DENSITY"])
print(f"Pearson's Correlation Coefficient: {correlation_coefficient}")
```

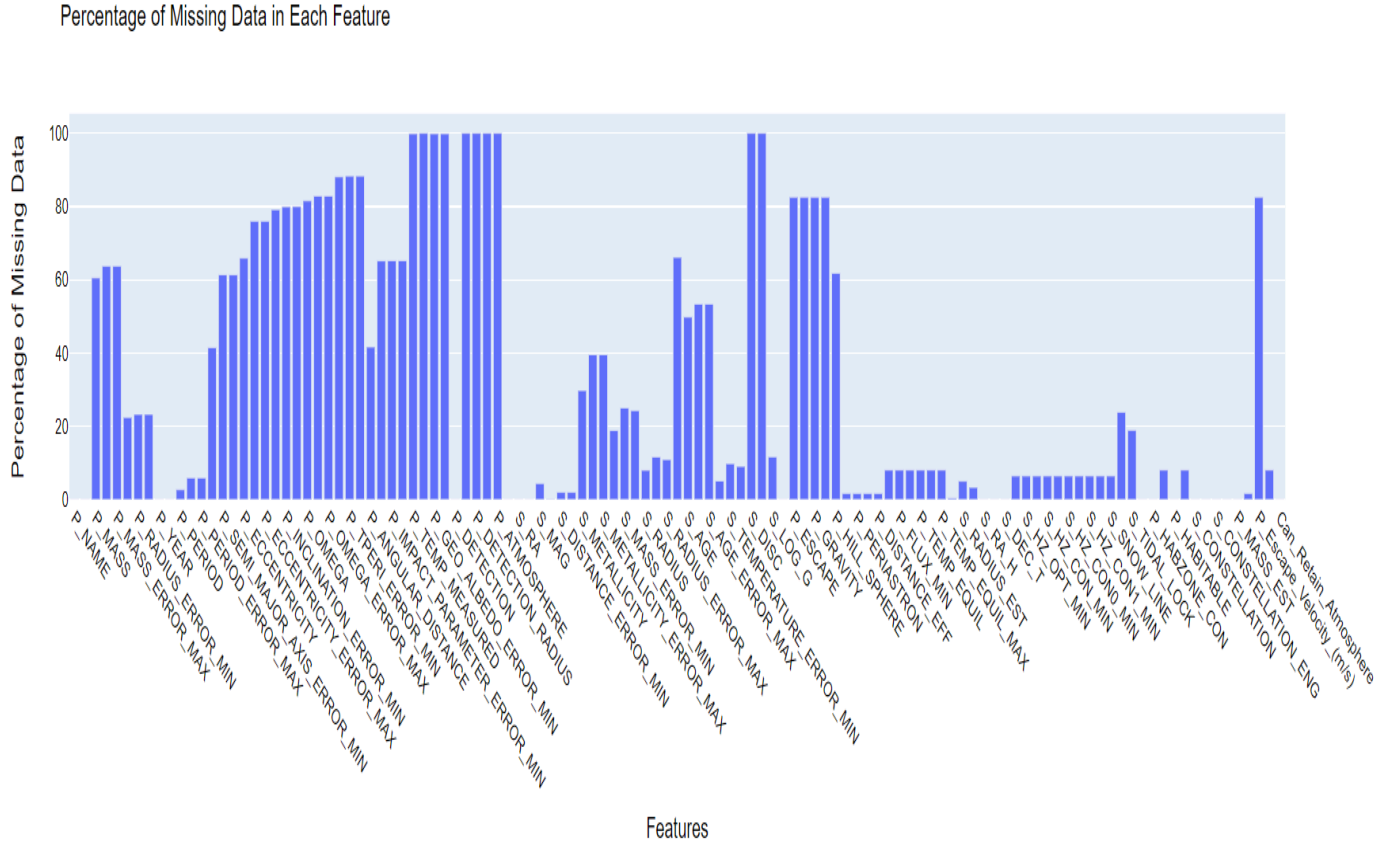




3. Feature Engineering:

3.1. Assessing Missing Data Percentage by Feature :

```
# Calculate the percentage of null values in each feature
null_percentage = (df.isnull().sum() / len(df)) * 100

# Create a DataFrame for plotting
null_percentage_df = pd.DataFrame({'Feature': null_percentage.index, 'Percentage Missing': null_percentage.values})

# Create an interactive bar chart using Plotly Express
fig = px.bar(
    null_percentage_df,
    x='Feature',
    y='Percentage Missing',
    title='Percentage of Missing Data in Each Feature',
    labels={'Feature': 'Features', 'Percentage Missing': 'Percentage of Missing Data'},
)

# Customize the layout
fig.update_xaxes(title_font=dict(size=16, family='Arial', color='black'))
fig.update_yaxes(title_font=dict(size=16, family='Arial', color='black'))
fig.update_layout(
    font=dict(family='Arial', size=12, color='black'),
    xaxis_tickangle=45,  # Rotate x-axis labels for better readability
)

# Show the interactive plot
fig.show()
```



3.2 Feature Reduction:

```
[ ]  # Drop columns with all NaN values
     data = df.dropna(axis=1, how='all')

     # data_cleaned = data.dropna(axis=0, how='all')

     # Print the shape of the cleaned DataFrame to check the changes
     print("Shape of cleaned DataFrame:", data.shape)
```

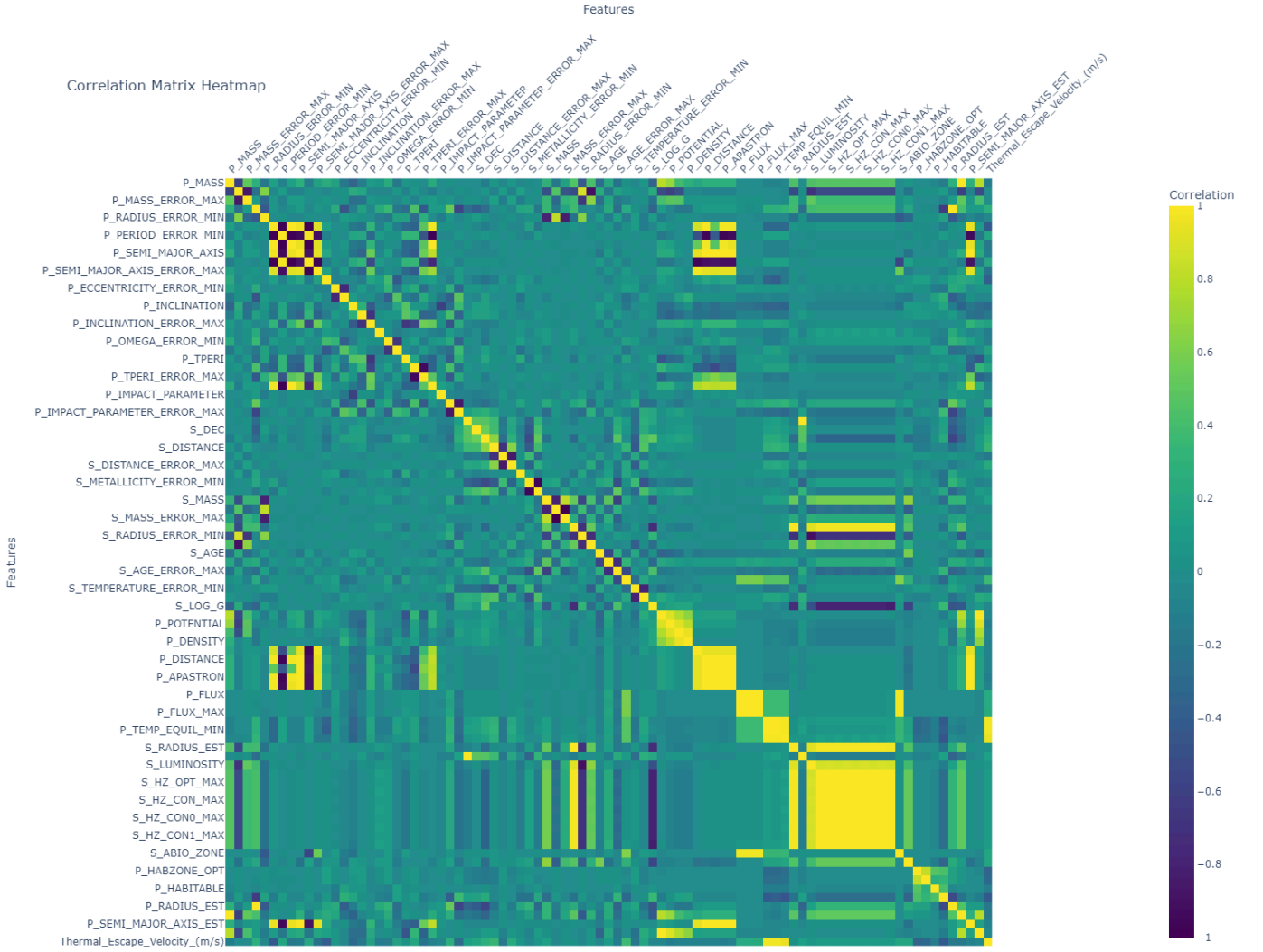
```
     Shape of cleaned DataFrame: (3367, 105)
```

```
[ ]  # Define the value to check for (e.g., 0)
     value_to_check = 0

     # Identify columns with only the specified value
     columns_to_remove = data.columns[(data == value_to_check).all()]

     # Drop the identified columns
     data_cleaned = data.drop(columns=columns_to_remove)

     # Print the shape of the cleaned DataFrame to check the changes
     print("Shape of cleaned DataFrame:", data_cleaned.shape)
```

```
     Shape of cleaned DataFrame: (3367, 104)
```

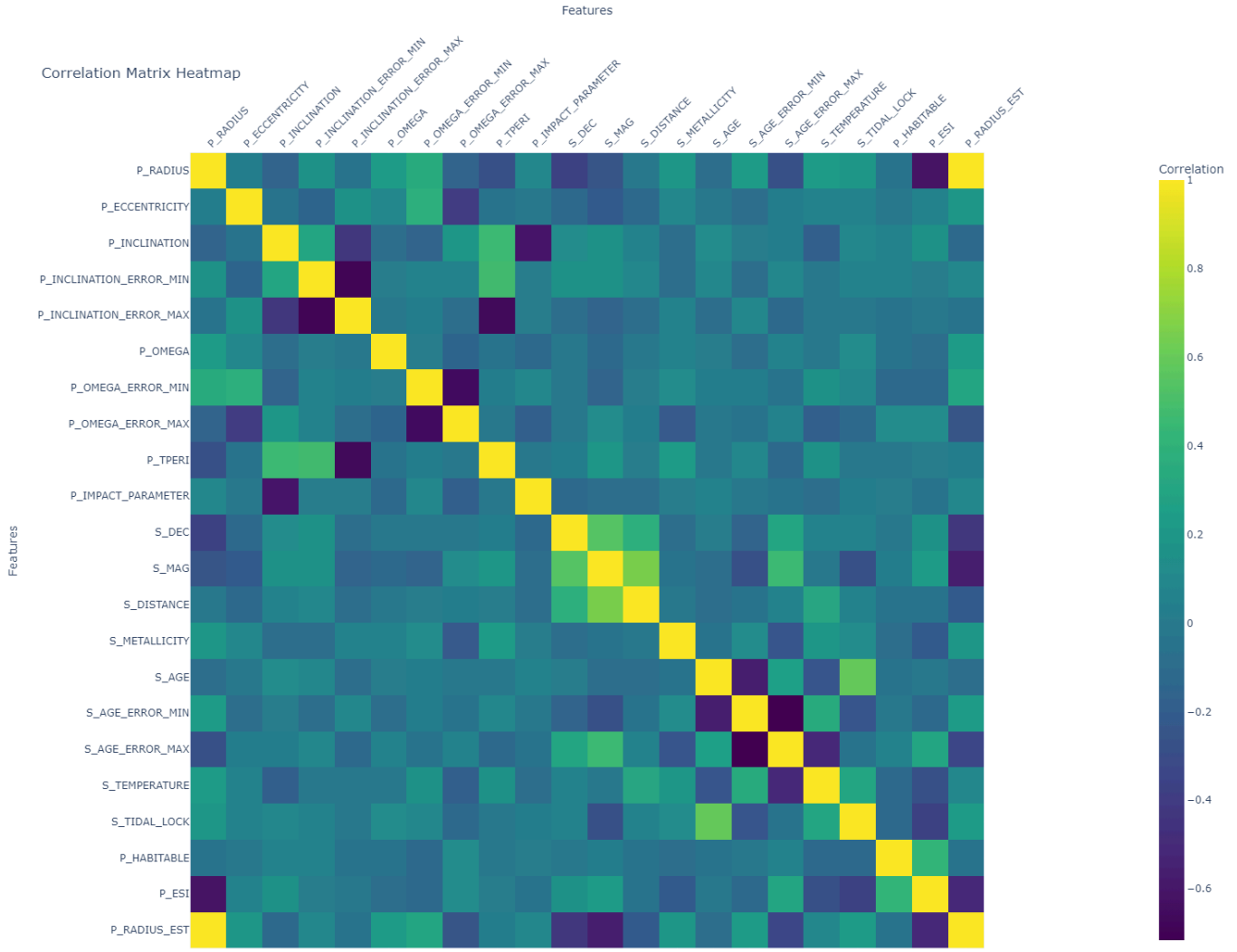
```
[ ]  data_cleaned.drop(['P_NAME','P_STATUS','P_RADIUS_ERROR_MAX', 'P_YEAR', 'P_UPDATED', 'S_CONSTELLATION_ENG', 'S_CONSTELLATION_ABR', 'S_CONSTELLATION', 'P_DETECTION', 'S_NAME', 'S_ALT_NAMES'
```

3.2.1 Detecting Redundant or Highly Correlated features :



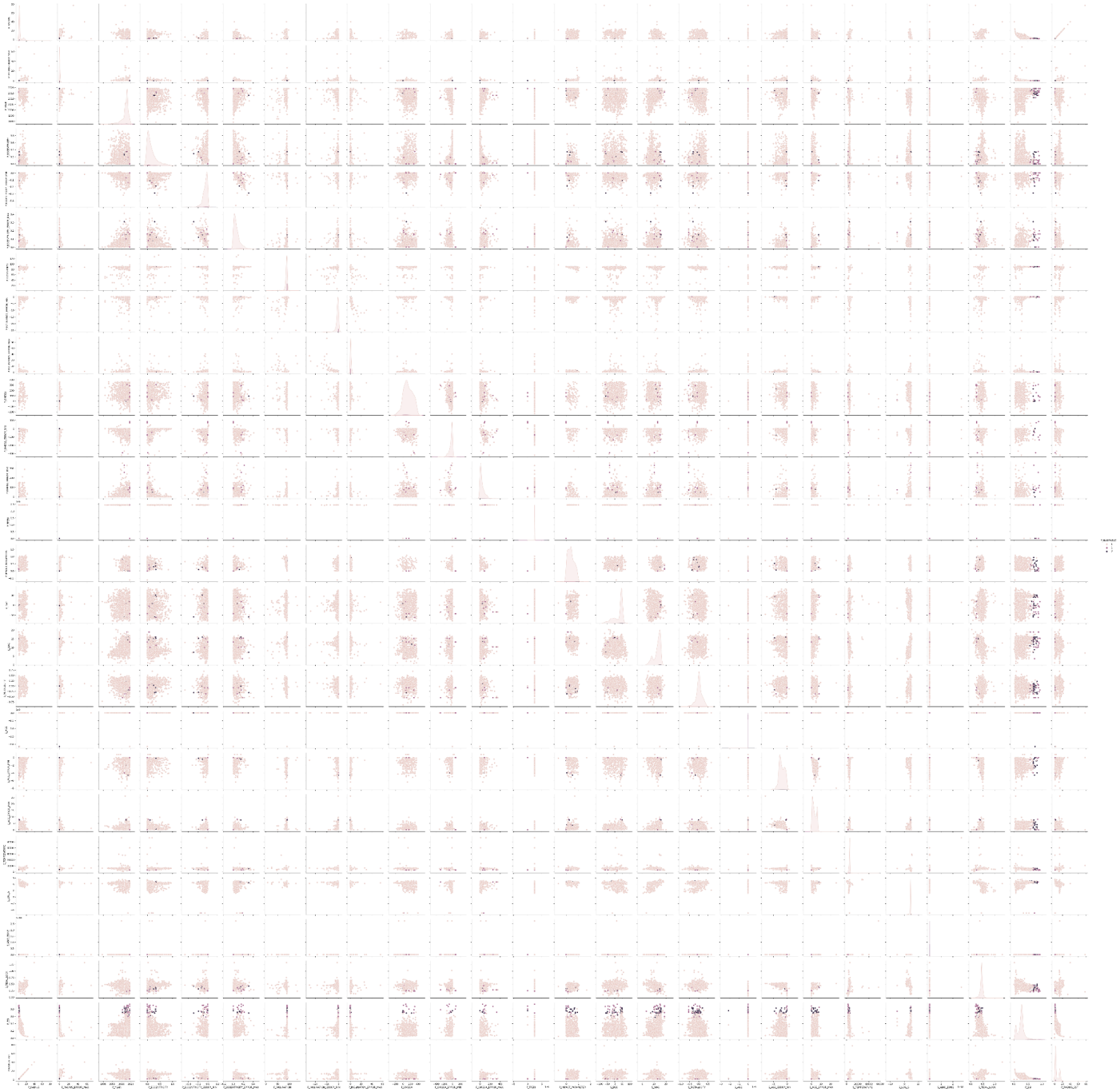
3.2.2 Selecting Feature Reduction Method:

```python
# Set a threshold for high correlation (adjust as needed)
correlation_threshold = 0.75

# Create a correlation matrix
correlation_matrix = df.corr().abs()

# Create a mask to identify highly correlated features
mask = (correlation_matrix >= correlation_threshold) & (correlation_matrix < 1.0)

# Identify columns to drop based on the mask
columns_to_drop = set()
for column in mask.columns:
    correlated_columns = mask.loc[mask[column]].index.tolist()
    if correlated_columns:
        columns_to_drop.add(column)  # Keep one of the correlated features

# Drop the identified columns from the DataFrame
df_filtered = df.drop(columns=columns_to_drop)

# Print the remaining features
remaining_features = df_filtered.columns.tolist()
print("Remaining Features:")
print(len(remaining_features))
```



This method is particularly useful when dealing with datasets where highly correlated features can lead to multicollinearity issues, which can negatively impact the performance and interpretability of certain machine learning models. By removing one feature from each highly correlated group, we aim to reduce redundancy and improve model stability. The choice of the correlation threshold (0.75 in this case) can be adjusted based on the specific dataset and problem requirements.

3.2.3. Exploring Feature Relationships Before and After Reduction :



3.3. Data Imputation Technique Selection :

```
# Identify numeric (continuous) and categorical columns
numeric_columns = data.select_dtypes(include=['number']).columns
categorical_columns = data.select_dtypes(exclude=['number']).columns

# Impute missing values in numeric columns with the mean (for continuous data)
data[numeric_columns] = data[numeric_columns].fillna(data[numeric_columns].mean())

# Impute missing values in categorical columns with the mode (for categorical data)
data[categorical_columns] = data[categorical_columns].fillna(data[categorical_columns].mode().iloc[0])

data.head()
```

In order to address the issue of missing values in our dataset, I've applied a suitable imputation technique. The objective is to fill in the gaps in our data effectively while considering the nature of the columns.

- First, I categorized the columns in our dataset into two main types: numeric (continuous) and categorical. This categorization is crucial because it determines how we handle the missing values in each column.
- For the numeric columns, which typically represent continuous data such as measurements or quantities, I chose to impute the missing values using the mean. This approach is appropriate for continuous data because it helps us maintain the central tendency of the data. By filling in missing values with the mean of each respective numeric column, we ensure that the statistical properties of the data, like the average, remain relatively unchanged.
- On the other hand, for categorical columns, which contain non-numeric data like categories or labels, I decided to impute missing values with the mode. The mode represents the most frequently occurring value in a categorical column. Imputing missing values with the mode is a suitable choice for categorical data because it helps preserve the overall distribution of categories within the column. By filling in missing values with the mode, we maintain the integrity of the categorical information.
- In summary, I employed a two-fold imputation strategy based on the type of data in each column: mean imputation for numeric (continuous) columns and mode imputation for categorical columns. This approach ensures that our dataset remains meaningful and representative after handling missing values, without introducing significant bias into the data.

4. Habitability classification:

4.1.1 Developing a Classifier for Exoplanet Habitability Classification :

```
[ ]  X = np.array(data_model.drop('P_HABITABLE', axis=1))
     y = np.array(data_model['P_HABITABLE'])
     y = y.astype(int)

[ ]  X_resampled, y_resampled = X,y
```

```
from sklearn.model_selection import train_test_split

X_train, X_, y_train, y_ = train_test_split(X_resampled, y_resampled, test_size=0.4, random_state=42)
X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.5, random_state=42)
del X_, y_

print(X_train.shape, X_cv.shape)
```

```
(2020, 36) (673, 36)
```

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import KLD, SparseCategoricalCrossentropy
from tensorflow.keras.optimizers import Adam


# Define the model
model = Sequential([
    Dense(units=64, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(units=40, activation='relu'),
    Dense(units=3, activation='softmax')  # Use 'softmax' activation for multi-class classification
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss=SparseCategoricalCrossentropy(from_logits=False),  # Use 'SparseCategoricalCrossentropy' for integer labels
              metrics=['accuracy'])

# Fit the model to your training data
model.fit(X_train_scaled, y_train, epochs=5)
```

```
Epoch 1/5
64/64 [==============================] - 1s 3ms/step - loss: 0.3135 - accuracy: 0.9337
Epoch 2/5
64/64 [==============================] - 0s 3ms/step - loss: 0.0558 - accuracy: 0.9866
Epoch 3/5
64/64 [==============================] - 0s 3ms/step - loss: 0.0269 - accuracy: 0.9911
Epoch 4/5
64/64 [==============================] - 0s 4ms/step - loss: 0.0138 - accuracy: 0.9975
Epoch 5/5
64/64 [==============================] - 0s 4ms/step - loss: 0.0079 - accuracy: 0.9985
<keras.src.callbacks.History at 0x7985080bc2b0>
```

```
# Evaluate the model on your test data
test_metrics = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_metrics[1])
```

```
22/22 [==============================] - 0s 4ms/step - loss: 0.0048 - accuracy: 0.9985
Test Accuracy: 0.998516321182251
```

4.1.2. K-Fold Cross Validation and Training Analysis :

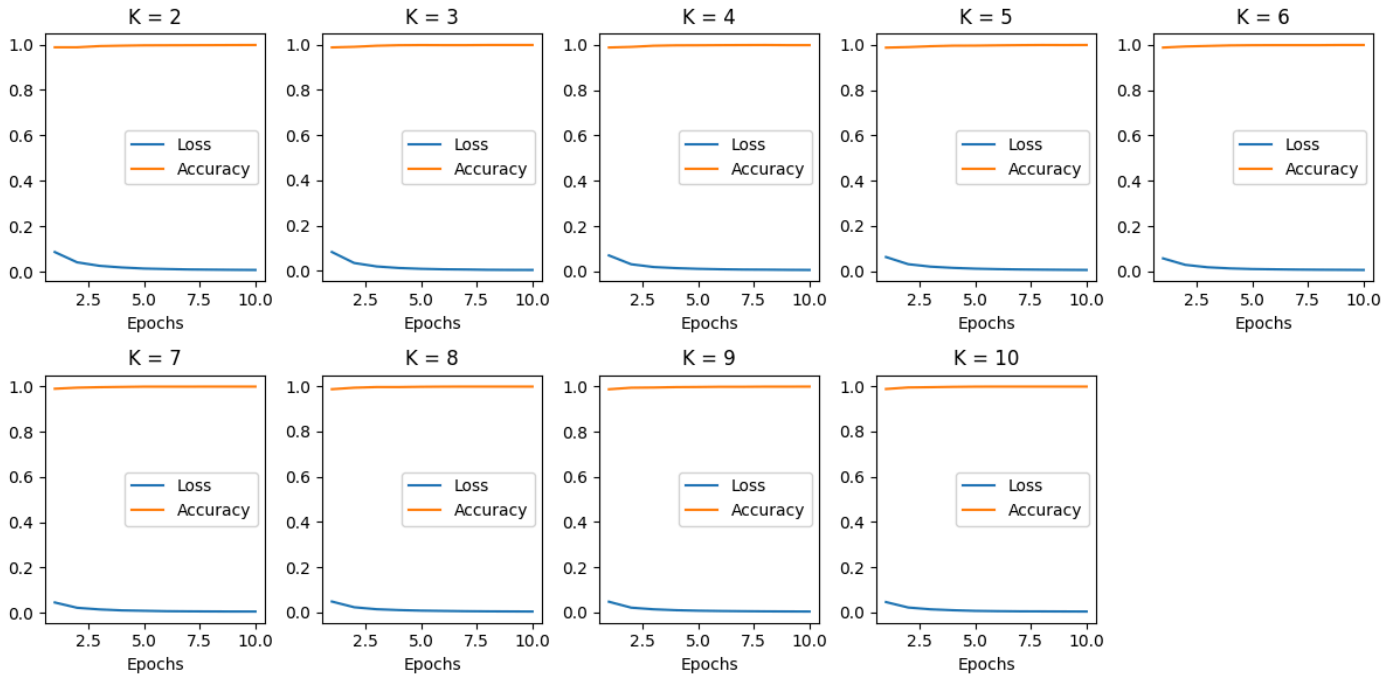
```
    # Split the data into training and validation sets for this fold
    X_train, X_val = X_resampled[train_index], X_resampled[val_index]
    y_train, y_val = y[train_index], y[val_index]

    # Scale the input features using StandardScaler
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)

    # Create and train the neural network model
    model = create_model()
    history = model.fit(X_train_scaled, y_train, epochs=10, validation_data=(X_val_scaled, y_val), verbose=0)

    # Store the loss and accuracy for this fold
    fold_loss.append(history.history['val_loss'])
    fold_accuracy.append(history.history['val_accuracy'])

    end_time = time.time()

    # Print information about the fold's performance
    print(f'For k: {k}, split no: {s_no}, loss: {history.history["val_loss"][-1]:.4f}, accuracy: {history.history["val_accuracy"][-1]:.4f}, time taken: {end_time - start_time:.4f}

# Compute average loss and accuracy over all folds for this K
avg_loss = np.mean(fold_loss, axis=0)
avg_accuracy = np.mean(fold_accuracy, axis=0)

loss_per_k.append(avg_loss)
accuracy_per_k.append(avg_accuracy)
```

Loading...

```
For k: 2, split no: 1, loss: 0.0041, accuracy: 0.9988, time taken: 3.9185
For k: 2, split no: 2, loss: 0.0126, accuracy: 0.9964, time taken: 5.7505
For k: 3, split no: 1, loss: 0.0042, accuracy: 0.9982, time taken: 4.8277
For k: 3, split no: 2, loss: 0.0019, accuracy: 0.9991, time taken: 4.0612
For k: 3, split no: 3, loss: 0.0072, accuracy: 0.9973, time taken: 4.3418
For k: 4, split no: 1, loss: 0.0051, accuracy: 0.9988, time taken: 4.3609
For k: 4, split no: 2, loss: 0.0016, accuracy: 1.0000, time taken: 5.6677
For k: 4, split no: 3, loss: 0.0161, accuracy: 0.9941, time taken: 5.6724
For k: 4, split no: 4, loss: 0.0050, accuracy: 0.9988, time taken: 3.9351
For k: 5, split no: 1, loss: 0.0095, accuracy: 0.9970, time taken: 5.6634
For k: 5, split no: 2, loss: 0.0022, accuracy: 1.0000, time taken: 4.4767
For k: 5, split no: 3, loss: 0.0045, accuracy: 0.9985, time taken: 6.3218
For k: 5, split no: 4, loss: 0.0021, accuracy: 1.0000, time taken: 5.6935
For k: 5, split no: 5, loss: 0.0053, accuracy: 1.0000, time taken: 5.6654
For k: 6, split no: 1, loss: 0.0114, accuracy: 0.9964, time taken: 5.7372
For k: 6, split no: 2, loss: 0.0010, accuracy: 1.0000, time taken: 5.6553
For k: 6, split no: 3, loss: 0.0013, accuracy: 1.0000, time taken: 4.5507
For k: 6, split no: 4, loss: 0.0191, accuracy: 0.9982, time taken: 4.1457
For k: 6, split no: 5, loss: 0.0031, accuracy: 1.0000, time taken: 5.6934
For k: 6, split no: 6, loss: 0.0058, accuracy: 0.9982, time taken: 4.0142
For k: 7, split no: 1, loss: 0.0091, accuracy: 0.9958, time taken: 5.6844
For k: 7, split no: 2, loss: 0.0014, accuracy: 1.0000, time taken: 4.0898
For k: 7, split no: 3, loss: 0.0008, accuracy: 1.0000, time taken: 5.6739
For k: 7, split no: 4, loss: 0.0007, accuracy: 1.0000, time taken: 4.7989
For k: 7, split no: 5, loss: 0.0245, accuracy: 0.9979, time taken: 4.0147
For k: 7, split no: 6, loss: 0.0030, accuracy: 1.0000, time taken: 4.3747
For k: 7, split no: 7, loss: 0.0043, accuracy: 0.9979, time taken: 5.9053
For k: 8, split no: 1, loss: 0.0039, accuracy: 0.9976, time taken: 4.0776
For k: 8, split no: 2, loss: 0.0012, accuracy: 1.0000, time taken: 5.6759
For k: 8, split no: 3, loss: 0.0006, accuracy: 1.0000, time taken: 5.6621
For k: 8, split no: 4, loss: 0.0009, accuracy: 1.0000, time taken: 4.6810
For k: 8, split no: 5, loss: 0.0115, accuracy: 0.9976, time taken: 4.0016
For k: 8, split no: 6, loss: 0.0048, accuracy: 0.9976, time taken: 5.6490
For k: 8, split no: 7, loss: 0.0012, accuracy: 1.0000, time taken: 5.9134
For k: 8, split no: 8, loss: 0.0021, accuracy: 1.0000, time taken: 4.8162
For k: 9, split no: 1, loss: 0.0101, accuracy: 0.9973, time taken: 5.7409
For k: 9, split no: 2, loss: 0.0002, accuracy: 1.0000, time taken: 5.7246
For k: 9, split no: 3, loss: 0.0007, accuracy: 1.0000, time taken: 4.7366
```

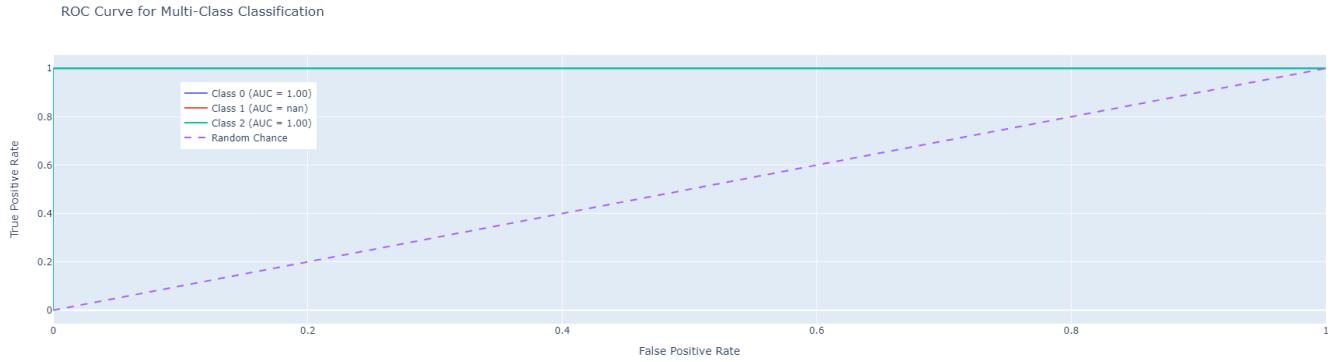
```
For k: 9, split no: 4, loss: 0.0019, accuracy: 1.0000, time taken: 3.9912
For k: 9, split no: 5, loss: 0.0009, accuracy: 1.0000, time taken: 5.6817
For k: 9, split no: 6, loss: 0.0156, accuracy: 0.9973, time taken: 4.2214
For k: 9, split no: 7, loss: 0.0030, accuracy: 1.0000, time taken: 5.6901
For k: 9, split no: 8, loss: 0.0019, accuracy: 1.0000, time taken: 5.6700
For k: 9, split no: 9, loss: 0.0044, accuracy: 1.0000, time taken: 5.6898
For k: 10, split no: 1, loss: 0.0146, accuracy: 0.9941, time taken: 4.7001
For k: 10, split no: 2, loss: 0.0003, accuracy: 1.0000, time taken: 5.6512
For k: 10, split no: 3, loss: 0.0008, accuracy: 1.0000, time taken: 4.3890
For k: 10, split no: 4, loss: 0.0010, accuracy: 1.0000, time taken: 5.9306
For k: 10, split no: 5, loss: 0.0007, accuracy: 1.0000, time taken: 5.6626
For k: 10, split no: 6, loss: 0.0142, accuracy: 0.9970, time taken: 5.9133
For k: 10, split no: 7, loss: 0.0007, accuracy: 1.0000, time taken: 3.8555
For k: 10, split no: 8, loss: 0.0021, accuracy: 1.0000, time taken: 4.6666
For k: 10, split no: 9, loss: 0.0062, accuracy: 0.9970, time taken: 3.9157
For k: 10, split no: 10, loss: 0.0020, accuracy: 1.0000, time taken: 5.6506
```

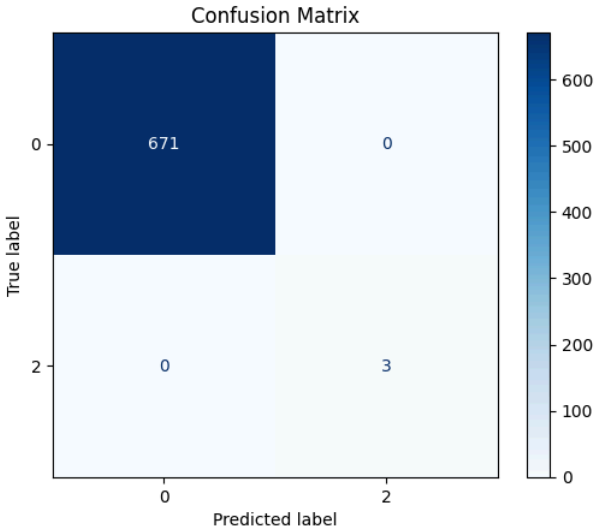
```
# Plot loss and accuracy versus epochs for each K
plt.figure(figsize=(12, 6))
for i, k in enumerate(k_values):
    plt.subplot(2, 5, i+1)
    num_epochs = len(loss_per_k[i])  # Get the actual number of epochs for this K
    plt.plot(range(1, num_epochs+1), loss_per_k[i], label='Loss')
    plt.plot(range(1, num_epochs+1), accuracy_per_k[i], label='Accuracy')
    plt.title(f'K = {k}')
    plt.xlabel('Epochs')
    plt.legend()
plt.tight_layout()
plt.show()
```



4.2. Performance Visualization: ROC Curve and Confusion Matrix :

```
import plotly.express as px
import plotly.graph_objs as go
from sklearn.metrics import roc_curve, auc, confusion_matrix, ConfusionMatrixDisplay

# Predict the class labels
y_pred = model.predict(X_test)

# Binarize the labels
n_classes = 3  # Number of classes (3 classes in your case)
y_test_bin = label_binarize(y_test, classes=range(n_classes))

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Create ROC curve traces
roc_curves = []
for i in range(n_classes):
    roc_curve_trace = go.Scatter(x=fpr[i], y=tpr[i], mode='lines',
                                 name=f'Class {i} (AUC = {roc_auc[i]:.2f})')
    roc_curves.append(roc_curve_trace)

# Create ROC curve for random chance (diagonal)
random_chance_trace = go.Scatter(x=[0, 1], y=[0, 1], mode='lines', line=dict(dash='dash'),
                                 name='Random Chance')

# Layout for ROC curve plot
roc_curve_layout = go.Layout(
    title='ROC Curve for Multi-Class Classification',
    xaxis=dict(title='False Positive Rate'),
    yaxis=dict(title='True Positive Rate'),
    legend=dict(x=0.1, y=0.9)
)

# Create ROC curve figure
roc_curve_fig = go.Figure(data=roc_curves + [random_chance_trace], layout=roc_curve_layout)
roc_curve_fig.show()

# Compute and plot confusion matrix using Matplotlib
conf_matrix = confusion_matrix(y_test, y_pred.argmax(axis=1))
ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=unique_labels).plot(cmap=plt.cm.Blues, values_format='.0f')
plt.title('Confusion Matrix')
plt.show()
```





4.3. Hyperparameter Optimization :

```python
from scipy.stats import uniform, randint
from sklearn.metrics import accuracy_score


# Define the hyperparameter search space
param_dist = {
    'learning_rate': uniform(0.0001, 0.1),  # Example range for learning rate
    'units_hidden_layer1': randint(16, 256),  # Example range for the number of neurons in the 1st hidden layer
    'units_hidden_layer2': randint(16, 256),  # Example range for the number of neurons in the 2nd hidden layer
}

# Number of random samples to try
num_samples = 50

best_accuracy = 0
best_hyperparameters = None

for _ in range(num_samples):
    # Randomly sample hyperparameters from the search space
    hyperparameters = {
        'learning_rate': np.random.uniform(0.0001, 0.1),
        'units_hidden_layer1': np.random.randint(16, 256),
        'units_hidden_layer2': np.random.randint(16, 256),
    }

    # Create a model with the sampled hyperparameters
    model = create_model(**hyperparameters)

    # Train the model on the training data
    history = model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0)

    # Evaluate the model on the test data
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, np.argmax(y_pred, axis=1))

    # Check if this set of hyperparameters resulted in a better accuracy
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_hyperparameters = hyperparameters

# Print the best hyperparameters and accuracy
print(f"Best Learning Rate: {best_hyperparameters['learning_rate']}")
print(f"Best Units in Hidden Layer 1: {best_hyperparameters['units_hidden_layer1']}")
print(f"Best Units in Hidden Layer 2: {best_hyperparameters['units_hidden_layer2']}")
print(f"Best Accuracy: {best_accuracy}")
```

```
Best Learning Rate: 0.06545731550189048
Best Units in Hidden Layer 1: 251
Best Units in Hidden Layer 2: 194
Best Accuracy: 0.9955489614243324
```

The premise behind this optimization method is to perform a randomized search over the hyperparameter space to avoid getting stuck in local optima and to explore a wide range of hyperparameter combinations. By conducting multiple trials and tracking the best-performing set of hyperparameters, this method aims to fine-tune the neural network model for optimal performance on the given dataset.

THANK YOU !

Team: Analytics Astronauts

- **Bhavesh Pabnani**
- **Yash Raj**
- **Parijat Sutradhar**