Project Documentation - Exoplanet Detection & Classification using Machine Learning

INTRODUCTION :

The quest to discover planets beyond our solar system, known as exoplanets, has captivated astronomers and scientists for decades. Exoplanets are planets that orbit stars outside our solar system, and their discovery helps us understand the vastness and diversity of planetary systems in the universe. One of the most effective methods to detect these distant worlds is through the transit method.

The transit method involves observing the dimming of a star's light caused by a planet passing, or transiting, in front of it. When an exoplanet transits its host star, it blocks a small fraction of the star's light, leading to a temporary decrease in brightness. This periodic dip in brightness can be detected and analyzed to infer the presence of an exoplanet. By studying the light curve — a graph of a star's brightness over time — scientists can determine various properties of the exoplanet, such as its size, orbital period, and distance from the host star.

The Kepler Space Telescope has been instrumental in discovering thousands of exoplanets using the transit method. The data collected by Kepler consists of light curves for over 150,000 stars, each potentially hosting one or more exoplanets.

Analyzing light curves to identify exoplanets involves several steps, including preprocessing the data, detecting potential transits, and extracting features such as transit depth, duration, and periodicity. Once these features are extracted, machine learning algorithms can be employed to classify the exoplanet candidates and confirm their existence.

In this project, we analyze light curve data from the Kepler Space Telescope to detect and characterize exoplanets. We employ various data preprocessing techniques, including flattening the light curve to remove trends and normalizing the data. We then use a threshold-based method to detect potential transits and extract features such as transit depth, duration, and periodicity.

To classify the exoplanet candidates, we train and evaluate several machine learning models, including Logistic Regression, Random Forest, and Gradient Boosting. We also perform hyperparameter tuning to optimize the models and improve their accuracy.

By the end of this project, we aim to develop a reliable method for detecting exoplanets and characterizing their properties using light curve data, contributing to the broader effort of exploring and understanding planetary systems beyond our own.

DETAILED STEPS FOR EXOPLANET DETECTION AND CLASSIFICATION PROJECT :

Step 1: Define the Objective

Step 2: Set Up Your Environment

Tools Used:

- **Python**: A versatile programming language commonly used for data science and astronomy.
- **Pip**: A package manager for Python used to install libraries.

Actions:

1. Ensure you have Python installed on your computer.
2. Install necessary Python libraries: `lightkurve`, `astropy`, `numpy`, `pandas`, `matplotlib`, `seaborn`, `scikit-learn` and `scipy`.

Step 3: Gather Data

Tools Used:

- **Lightkurve**: A Python package for downloading and analyzing astronomical light curve data from missions like Kepler and TESS.

Actions:

1. Search for light curve data using the target star's Kepler ID.
2. Download the light curve file containing the star's brightness measurements over time.

Step 4: Visualize Light Curve Data

Tools Used:

- **Lightkurve**: For loading light curve data.
- **Matplotlib**: A plotting library for creating visualizations.

Actions:

1. Use Lightkurve to search and download the light curve data for the target star.
2. Plot the light curve to visualize the star's brightness over time, looking for any dips that might indicate transits.

Step 5: Data Preprocessing

Tools Used:

- **Pandas**: A library for data manipulation and analysis.
- **Lightkurve**: For normalizing and flattening light curves.

Actions:

1. Remove any NaN (Not a Number) values from the light curve data to handle missing values.
2. Normalize the light curve to correct for variations in brightness that are not related to transits.
3. Flatten the light curve to remove long-term trends and highlight short-term events like transits.

Step 6: Feature Extraction

Tools Used:

- **SciPy**: A library for scientific computing, used here for signal processing.

Actions:

1. Detect potential transit events by identifying dips in the light curve.
2. Measure the depth, duration, and periodicity of each detected dip to extract features that describe each potential transit event.

Step 7: Split Data into Training and Testing Sets

Tools Used:

- **Scikit-learn**: A machine learning library for data splitting and model training.

Actions:

1. Label the data, creating labels for whether each event is a transit (exoplanet) or not.
2. Split the dataset into training and testing sets to evaluate model performance.

Step 8: Train a Machine Learning Model

Tools Used:

- **Scikit-learn**: For implementing basic machine learning models.

- **TensorFlow/Keras**: For more advanced machine learning models, especially deep learning.

Actions:

1. Choose a machine learning model, starting with simpler models like Logistic Regression and moving to more complex ones like Random Forest, Gradient Boosting, or Convolutional Neural Networks (CNNs).
2. Train the model on the training set, using the extracted features.
3. Validate the model on the testing set, adjusting hyperparameters to improve performance.

Step 9: Evaluate Model Performance

Tools Used:

- **Scikit-learn**: For evaluation metrics and performance visualization.

Actions:

1. Use metrics such as accuracy, precision, recall, F1-score to evaluate the model.
2. Create visualizations (e.g., confusion matrix) to better understand the model's performance.

Step 10: Hyperparameter Tuning

Tools Used:

- **Grid Search/Random Search**: Techniques for hyperparameter tuning.

Actions:

1. Continuously refine your model by tuning hyperparameters using Grid Search or Random Search.
2. Experiment with different feature extraction methods and preprocessing techniques.

STEP 1 : DEFINING THE OBJECTIVE :

The primary objective of this project is to classify exoplanet candidates from the Kepler mission using machine learning techniques. The project is divided into two main parts: data analysis and model training. Each part plays a crucial role in achieving accurate and reliable classification of exoplanet candidates, confirmed exoplanets, and false positives.

The data analysis part include detecting exoplanets using lightcurve data from missions like Kepler. The goal is to identify potential exoplanets by analyzing light curves and classify them as either exoplanetary transits or other phenomena (e.g., binary stars).

The model training part includes building an efficient classifier that classifies exoplanets into various categories (ie., Confirmed, Candidates, False Positives), using available observations.

THE TRANSIT METHOD TO DETECT EXOPLANETS :

A transit in the context of exoplanet detection is an astronomical event that occurs when a planet passes in front of its host star as viewed from Earth. This passage causes a temporary and periodic dimming of the star's light, which can be detected and analyzed to infer the presence of the planet.

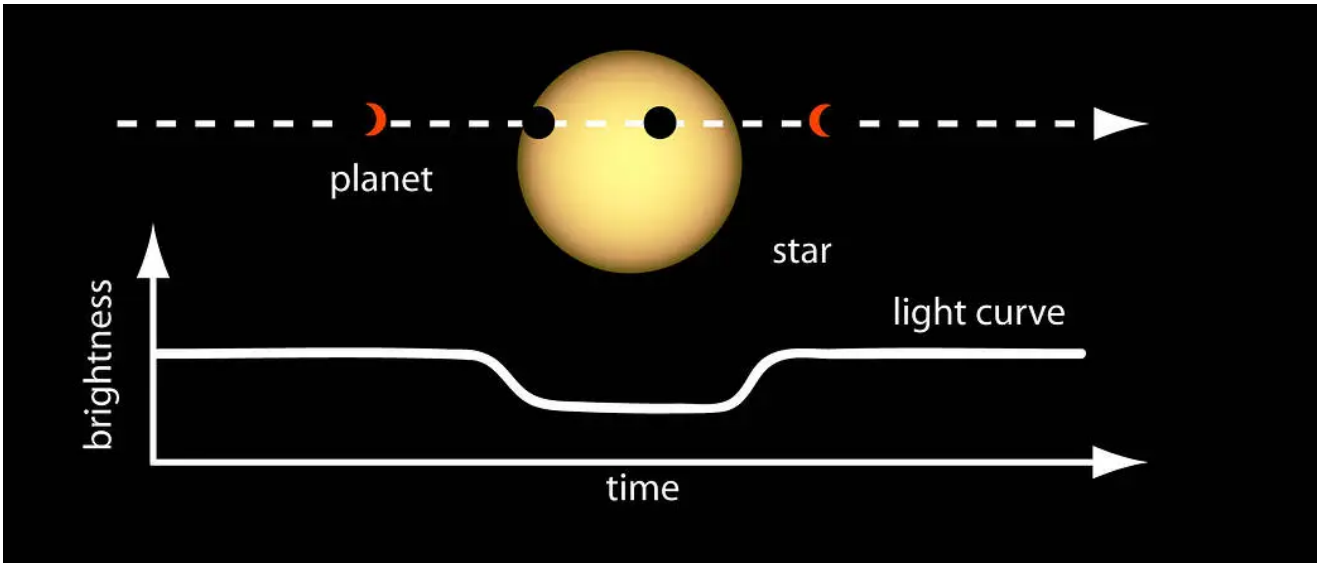


Image from https://www.nasa.gov/image-article/light-curve-of-planet-transiting-star/

Key Concepts Involved in a Transit:

1. **Light Curve**:
   - A light curve is a graph of a star's brightness as a function of time. When a planet transits its star, it blocks a portion of the star's light, causing a characteristic dip in the light curve.
2. **Transit Depth**:
   - The depth of the transit dip corresponds to the fraction of the star's light that is blocked by the planet. This can provide information about the planet's size relative to the star.
3. **Transit Duration**:
   - The duration of the transit is the time it takes for the planet to cross the star's disk. This can help estimate the planet's orbital speed and, when combined with other

data, its distance from the star.

4. **Periodicity**:
    - Transits are periodic events that recur at regular intervals, corresponding to the planet's orbital period around the star. Detecting multiple transits at consistent intervals can confirm the presence of a planet.

The Transit Method of Exoplanet Detection:

The transit method is one of the most successful techniques for detecting exoplanets. It involves continuous monitoring of a star's brightness to detect the characteristic periodic dips caused by transiting planets. This method has been employed by several space telescopes, including Kepler, TESS (Transiting Exoplanet Survey Satellite), and others.

Steps Involved in Detecting a Transit:

1. **Observation**:
    - Continuous monitoring of the star's brightness over an extended period.
2. **Light Curve Analysis**:
    - Plotting the star's brightness as a function of time to produce a light curve.
3. **Transit Identification**:
    - Identifying periodic dips in the light curve that indicate potential transits.
4. **Feature Extraction**:
    - Analyzing the depth, duration, and periodicity of the transit to infer the planet's properties.
5. **Confirmation**:
    - Performing follow-up observations and analyses to confirm the planetary nature of the detected transits and rule out false positives.

STEP 2: SETTING UP THE ENVIRONMENT :

```
import lightkurve as lk
from lightkurve import search_targetpixelfile
from lightkurve import TessTargetPixelFile

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.signal import find_peaks

from astropy.timeseries import BoxLeastSquares
from astropy.time import Time
```

```
from sklearn.model_selection import train_test_split , GridSearchCV,
StratifiedKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.impute import SimpleImputer

import seaborn as sns
```

Packages Used :

Lightkurve

```
import lightkurve as lk
```

- **Lightkurve**: A Python package for downloading, analyzing, and visualizing astronomical light curves, primarily from the Kepler and TESS missions.

Lightkurve Functions

```
from lightkurve import search_targetpixelfile
```

- **search_targetpixelfile**: A function to search for target pixel files for a given object, which contain raw pixel data used to produce light curves.

```
from lightkurve import TessTargetPixelFile
```

- **TessTargetPixelFile**: A class for handling TESS target pixel files, allowing for manipulation and extraction of light curves from TESS data.

NumPy

```
import numpy as np
```

- **NumPy**: A fundamental package for scientific computing in Python, providing support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions.

Pandas

```
import pandas as pd
```

- **Pandas**: A powerful data manipulation and analysis library for Python, providing data structures like DataFrames to handle structured data efficiently.

Matplotlib

```
import matplotlib.pyplot as plt
```

- **Matplotlib**: A comprehensive library for creating static, animated, and interactive visualizations in Python.

SciPy

```
from scipy.signal import find_peaks
```

- **SciPy**: A Python library used for scientific and technical computing. The `find_peaks` function from the `scipy.signal` module is used to detect peaks (local maxima) in data.

Astropy

```
from astropy.timeseries import BoxLeastSquares
```

- **Astropy**: A package intended for astronomy-related computations. The `BoxLeastSquares` class is used for detecting periodic signals in time-series data, particularly useful for identifying transits in light curves.

```
from astropy.time import Time
```

- **Time**: A class in Astropy for handling and converting time in various formats, crucial for time-series data analysis.

Scikit-Learn

```
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold
```

- **train_test_split**: A function to split data arrays into random train and test subsets.
- **GridSearchCV**: A tool for performing hyperparameter tuning by exhaustive search over specified parameter values.
- **StratifiedKFold**: A cross-validator that provides train/test indices to split data into train/test sets, preserving the percentage of samples for each class.

```
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

- **StandardScaler**: A transformer to standardize features by removing the mean and scaling to unit variance.
- **LabelEncoder**: A utility to convert categorical labels into numeric format.

```
from sklearn.linear_model import LogisticRegression
```

- **LogisticRegression**: A linear model for classification tasks that estimates probabilities using a logistic function.

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
```

- **RandomForestClassifier**: An ensemble learning method for classification that constructs multiple decision trees and outputs the mode of their predictions.
- **GradientBoostingClassifier**: An ensemble technique that builds models sequentially, with each model trying to correct errors made by the previous one.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix, ConfusionMatrixDisplay
```

- **accuracy_score**: A function to compute the accuracy of the model.
- **precision_score**: A function to compute the precision of the model.
- **recall_score**: A function to compute the recall of the model.
- **f1_score**: A function to compute the F1 score, which is the harmonic mean of precision and recall.
- **confusion_matrix**: A function to compute the confusion matrix to evaluate the accuracy of a classification.
- **ConfusionMatrixDisplay**: A utility to display the confusion matrix.

```
from sklearn.impute import SimpleImputer
```

- **SimpleImputer**: A utility to handle missing data by imputing missing values with strategies like mean, median, or most frequent.

Seaborn

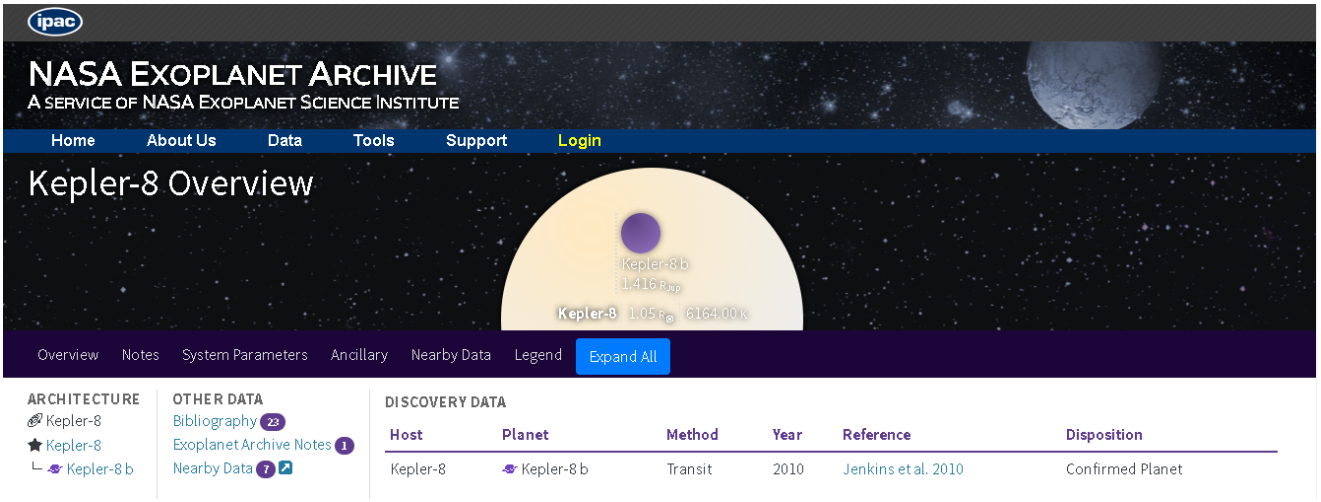
```
import seaborn as sns
```

- **Seaborn**: A Python visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.

STEP 3: GATHERING DATA :

1. **Download the Pixel File for a Given Star**:

We choose to analyze a star who's presence of an exoplanet has already been confirmed.
Star : Kepler-8
KeplerID : KIC 6922244
Planet(s) : Kepler-8 b



Source : NASA Exoplanet Archive - https://exoplanetarchive.ipac.caltech.edu/overview/Kepler-8%20b#planet_Kepler-8-b_collapsible

```
# A quarter means a quarter of a year
pixelFile = search_targetpixelfile('KIC 6922244', author="Kepler",
cadence="long", quarter=4).download()
# Show a single snapshot
pixelFile.plot(frame=42)
```

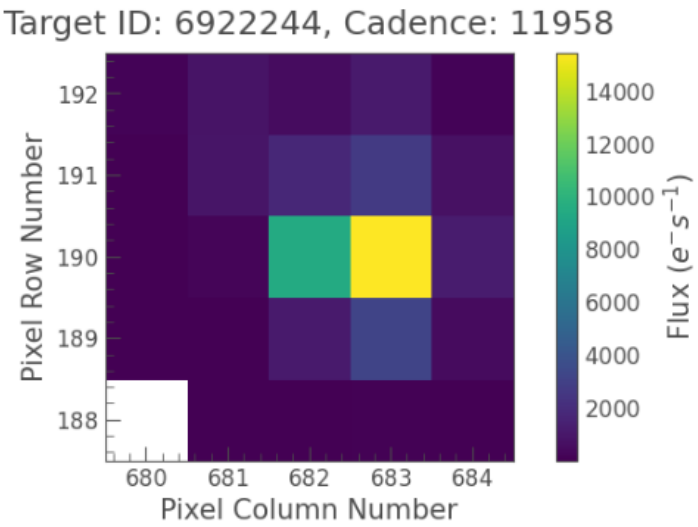
- **search_targetpixelfile**: This function searches for the target pixel file of a specified star. In this case, it searches for data related to the star with ID 'KIC 6922244' from the Kepler mission, during the 4th quarter, with a long cadence.
- What does `cadence` and `quarter` mean in the data?
  `Cadence`:
- **Description**: Cadence refers to the frequency at which observations are made. In the context of Kepler data, it usually means how often the brightness of a star was recorded.
- **Types**: Common cadences are long cadence (approximately 30 minutes) and short cadence (approximately 1 minute).
- **Usage**: Determines the resolution of the light curve data. Short cadence data provides finer details of the light curve.
  `Quarter`:
- **Description**: A quarter is a segment of the Kepler mission's observation period. The Kepler spacecraft collected data in approximately three-month periods, each referred to as a quarter.

- **Usage**: Different quarters represent different time periods of observation. Combining multiple quarters can provide a longer and more complete light curve for a star.

    - **download()**: Downloads the pixel file.
    - **plot(frame=42)**: Plots a specific frame from the pixel file, showing the flux measured by each pixel in a grid format.

    Output: The plot shows the flux values in a grid, where each cell represents a pixel on the telescope's CCD. The color represents the flux intensity, with yellow being the highest and purple the lowest.

```
<Axes: title={'center': 'Target ID: 6922244, Cadence: 11958'}, xlabel='Pixel Column Number', ylabel='Pixel Row Number'>
```



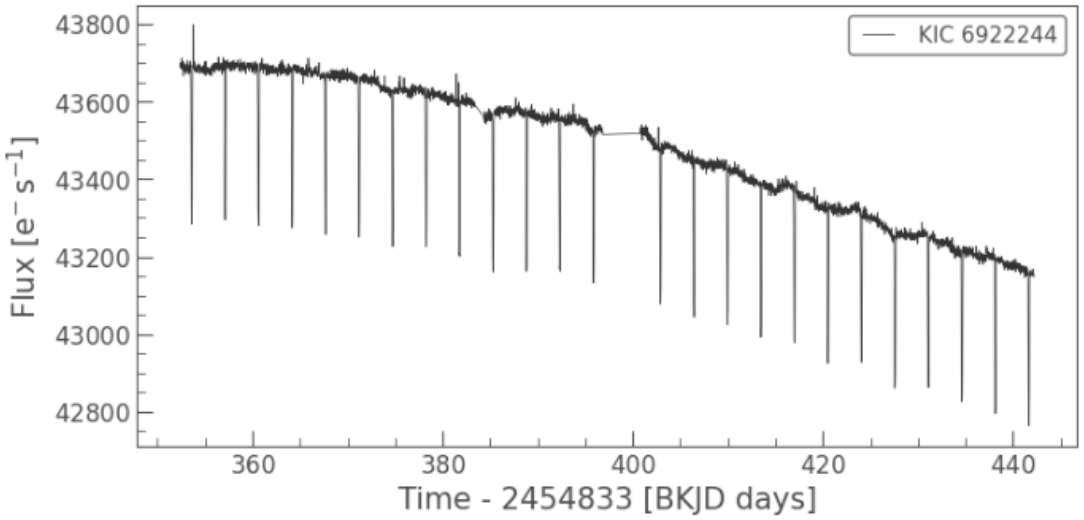
2. **Aperture Masks Make the Image Look Better for Analysis**:

```
pixel_to_lc =
pixelFile.to_lightcurve(aperture_mask=pixelFile.pipeline_mask)
pixel_to_lc.plot()

# Data about the lightcurve data
pixel_to_lc_df = pixel_to_lc.to_pandas()
print(pixel_to_lc_df.head())
print(pixel_to_lc_df.columns)
```

    - **to_lightcurve**: Converts the pixel file to a light curve, using the aperture mask provided by the pipeline. This mask helps in identifying which pixels should be included to measure the flux of the target star.
    - **plot()**: Plots the light curve, showing the flux (brightness) of the star over time.
    - **to_pandas()**: Converts the light curve data into a Pandas DataFrame for easier manipulation and analysis.
    - **print(pixel_to_lc_df.head())**: Displays the first few rows of the DataFrame.
    - **print(pixel_to_lc_df.columns)**: Displays the column names of the DataFrame.

Output:

- The plot shows the star's brightness over time. The x-axis is time, and the y-axis is the normalized flux.



- The DataFrame contains columns such as 'flux', 'flux_err', 'centroid_col', 'centroid_row', 'cadenceno', and 'quality', which provide detailed information about the flux measurements and their errors, the position of the star's centroid, the cadence number, and quality flags for each measurement.

```
                  flux  flux_err  centroid_col  centroid_row  cadenceno  \
time
352.376325  43689.148438  6.631562    682.680325    190.072614      11914
352.396758  43698.078125  6.631830    682.679939    190.072439      11915
352.437624  43694.105469  6.631788    682.679626    190.072676      11917
352.458058  43698.316406  6.631949    682.679788    190.072496      11918
352.478491  43687.648438  6.631505    682.679287    190.072465      11919

            quality
time
352.376325        0
352.396758     8192
352.437624       16
352.458058        0
352.478491        0
Index(['flux', 'flux_err', 'centroid_col', 'centroid_row', 'cadenceno',
       'quality'],
      dtype='object')
```

Interpretation of the Outputs:

1. **Pixel File Plot**:
   - This plot shows the flux captured by the Kepler spacecraft's CCD for the specified star. The bright central pixels indicate the star's position, while the surrounding pixels capture less light.
2. **Light Curve Plot**:
   - This plot visualizes the star's brightness over time. The periodic dips in brightness (if present) may indicate potential transits, where a planet passes in front of the star, causing a temporary decrease in observed brightness.

3. **DataFrame Head and Columns**:
    - The DataFrame provides a structured view of the light curve data. Each row represents a measurement taken at a specific time. The columns provide detailed information about each measurement:
        - **flux**: The measured brightness of the star.
        - **flux_err**: The error associated with the flux measurement.
        - **centroid_col** and **centroid_row**: The column and row of the star's centroid in the CCD grid.
        - **cadenceno**: The cadence number, which indicates the sequence of observations.
        - **quality**: Flags indicating the quality of each measurement, useful for filtering out poor-quality data.
4. **Download light curve data for a target star:**

```
# Specify the Kepler ID of the target star
target = 'KIC 6922244'  # Kepler ID

# Search for the light curve file of the specified target
search_result = lk.search_lightcurve(target, quarter=4, cadence='long')

# Print the search results to understand what data is available
print(search_result)

# Download all the light curve files from the search results
lc_files = search_result.download_all()

# Print the downloaded light curve files to understand their structure
print(lc_files)
```

Explanation:

1. **Specify the Kepler ID of the Target Star**:

```
target = 'KIC 6922244'  # Kepler ID
```

    - **target**: The Kepler ID of the target star is specified. Here, the target is 'KIC 6922244'.
2. **Search for the Light Curve File of the Specified Target**:

```
search_result = lk.search_lightcurve(target, quarter=4, cadence='long')
```

- **lk.search_lightcurve**: This function searches for light curve data of the specified target star.
  - **target**: The Kepler ID of the target star.
  - **quarter**: The observation quarter. In this case, it is set to the 4th quarter.
  - **cadence**: The observation cadence. 'long' indicates long cadence observations.

3. **Print the Search Results**:

```
print(search_result)
```

- **print(search_result)**: This command prints the search results, giving an overview of the available data files.

4. **Download All the Light Curve Files from the Search Results**:

```
lc_files = search_result.download_all()
```

- **download_all()**: This method downloads all the light curve files found in the search results.

5. **Print the Downloaded Light Curve Files**:

```
print(lc_files)
```

- **print(lc_files)**: This command prints the downloaded light curve files, providing information about their structure and content.

Output:

1. **Search Results**:
   The output of `print(search_result)` will show the list of available light curve files for the specified target star, indicating the observation quarter and cadence.

```
SearchResult containing 1 data products.

 #       mission      year author exptime  target_name  distance
                                    s                    arcsec
--- ----------------- ---- ------ ------- ------------- --------
  0 Kepler Quarter 04 2010 Kepler    1800 kplr006922244      0.0
```

2. **Downloaded Light Curve Files**:
   The output of `print(lc_files)` will show the details of the downloaded light curve files, indicating their structure and content.

```
LightCurveCollection of 1 objects:
    0: <KeplerLightCurve LABEL="KIC 6922244" QUARTER=4 AUTHOR=Kepler FLUX_ORIGIN=pdcsap_flux>
```

STEP 4: VISUALIZING LIGHT CURVE DATA & STEP 5: DATA PREPROCESSING :

```
# Plot all the downloaded light curve files
lc_files.plot()

# Combine the light curves from all the downloaded files
lc = lc_files.PDCSAP_FLUX.stitch()

# Plot the combined light curve to visualize the star's brightness over time
lc.plot()
plt.title(f'Light Curve for {target}')
plt.xlabel('Time [BKJD days]')
plt.ylabel('Normalized Flux')
plt.show()

# Remove NaN values
lc = lc.remove_nans()

# Normalize the light curve
lc = lc.normalize()

# Flatten the light curve to remove long-term trends
lc_flat = lc.flatten(window_length=901)
lc_flat.plot()
plt.title(f'Flattened Light Curve for {target}')
plt.xlabel('Time [BKJD days]')
plt.ylabel('Normalized Flux')
plt.show()
```

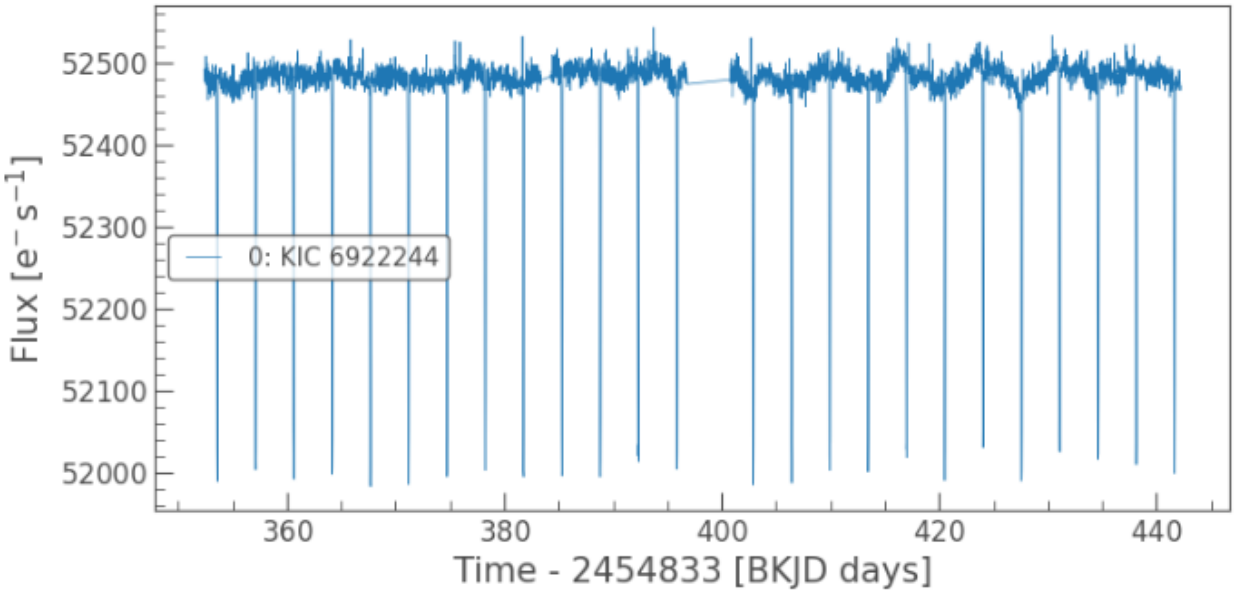
Explanation:

1. **Plot All the Downloaded Light Curve Files**:

   ```
   lc_files.plot()
   ```

   - **lc_files.plot()**: This function plots all the downloaded light curve files, providing a visual representation of the star's brightness over time across all quarters.

   Output:
   - A plot showing the star's brightness (flux) over time for each downloaded light curve file.



2. **Combine the Light Curves from All the Downloaded Files**:

```
lc = lc_files.PDCSAP_FLUX.stitch()
```

   - **lc_files.PDCSAP_FLUX.stitch()**: This function combines (stitches) all the light curve files' Pre-search Data Conditioning Simple Aperture Photometry (PDCSAP) flux values into a single light curve.

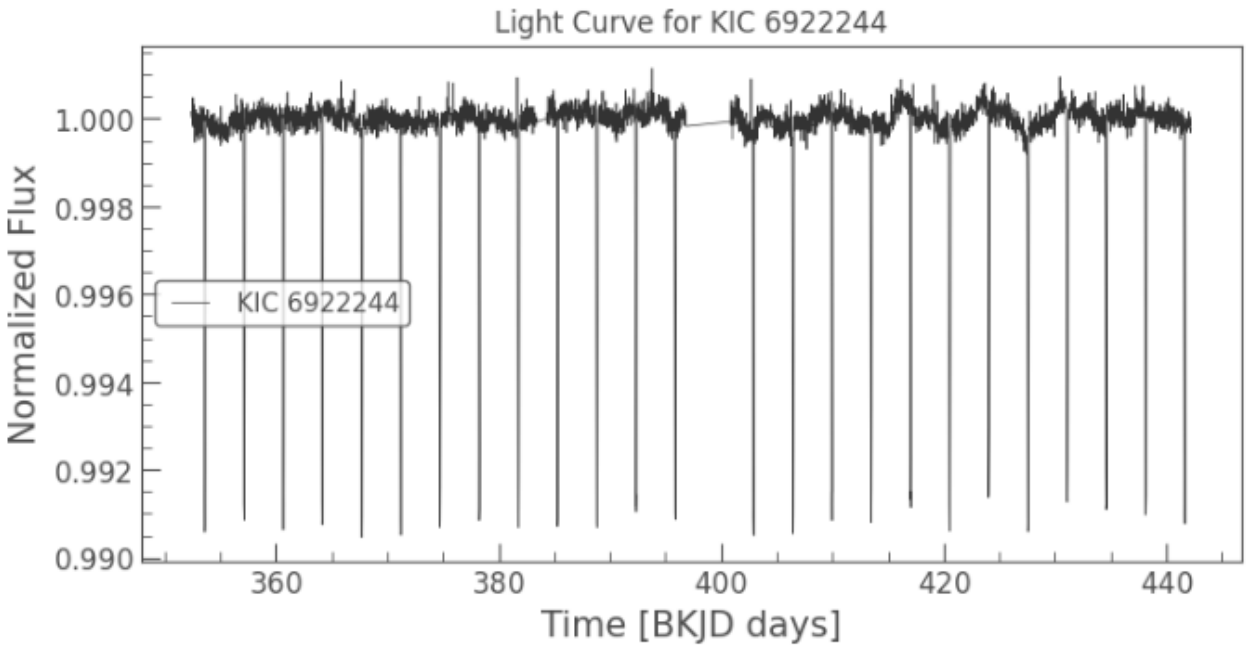
3. **Plot the Combined Light Curve**:

```
lc.plot()
plt.title(f'Light Curve for {target}')
plt.xlabel('Time [BKJD days]')
plt.ylabel('Normalized Flux')
plt.show()
```

   - **lc.plot()**: This function plots the combined light curve, showing the star's brightness over time.
   - **plt.title, plt.xlabel, plt.ylabel**: These functions add title and labels to the plot for better understanding.

Output:

- A plot showing the combined light curve, representing the star's brightness over time.



4. **Remove NaN Values**:

```
lc = lc.remove_nans()
```

   - **lc.remove_nans()**: This function removes any NaN (Not a Number) values from the light curve, ensuring clean data for further analysis.

5. **Normalize the Light Curve**:

```
lc = lc.normalize()
```

   - **lc.normalize()**: This function normalizes the light curve, scaling the flux values to have a mean of 1. This step helps in standardizing the data.

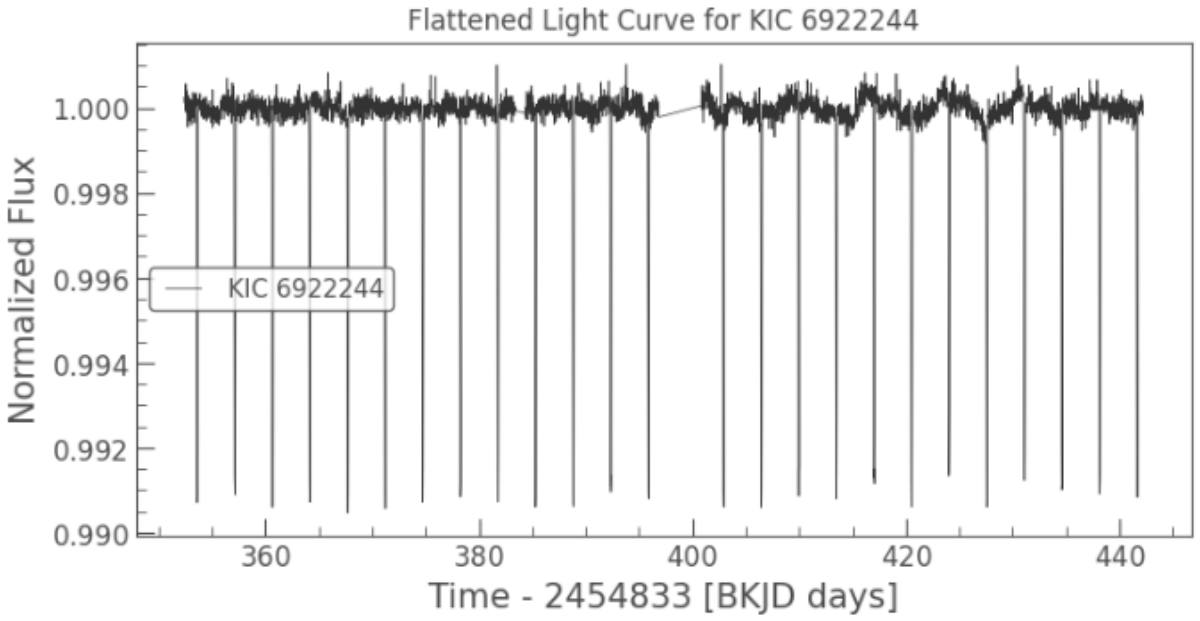
6. **Flatten the Light Curve**:

```
lc_flat = lc.flatten(window_length=901)
lc_flat.plot()
plt.title(f'Flattened Light Curve for {target}')
plt.xlabel('Time [BKJD days]')
plt.ylabel('Normalized Flux')
plt.show()
```

   - **lc.flatten(window_length=901)**: This function flattens the light curve by removing long-term trends using a window length of 901. Flattening helps in identifying periodic signals more easily.
   - **lc_flat.plot()**: This function plots the flattened light curve.
   - **plt.title, plt.xlabel, plt.ylabel**: These functions add title and labels to the plot for better understanding.

Output:

- A plot showing the flattened light curve, which highlights the periodic dips more clearly.



Understanding the Flattened Light Curve Plot:

Understanding the plot of the light curve scientifically involves interpreting the variations in the star's brightness over time.

Components of the Light Curve Plot :

1. **Time (X-axis)**:
   - The X-axis represents time in Barycentric Julian Date (BKJD) days. BKJD is a time standard used by the Kepler mission.
   - BKJD is the Julian Date minus 2,454,833 days. This standard allows astronomers to have a common reference point for time measurements.
2. **Normalized Flux (Y-axis)**:
   - The Y-axis represents the normalized flux. Flux is the amount of light received from the star.
   - Normalized flux is adjusted so that the average flux is 1. This makes it easier to see variations and dips in the light curve.
3. **Light Curve Data**:
   - The black points represent the brightness measurements of the star over time.
   - Variations in the light curve can indicate different astrophysical phenomena.

Key Features in the Light Curve :

1. **Periodic Dips**:
    - Look for periodic dips in the light curve. These dips could indicate transits, where a planet passes in front of the star, blocking some of its light.
    - The depth, duration, and regularity of these dips can provide information about the size and orbit of the exoplanet.
2. **Gaps in Data**:
    - Notice the gaps in the data, which appear as horizontal white spaces in the plot. These gaps can be due to the spacecraft entering safe mode, data transmission issues, or scheduled breaks in observation.
3. **Noise and Outliers**:
    - The scatter in the data points represents noise, which can come from various sources, including instrumental noise, cosmic rays, and stellar activity.
    - Some spikes or outliers might not be related to transits but could be due to noise or other stellar phenomena such as star spots or flares.

Scientific Interpretation :

1. **Transit Detection**:
    - Identify regular dips in the normalized flux. These dips are potential transits.
    - Measure the period (time between consecutive dips) to determine the orbital period of the exoplanet.
2. **Exoplanet Characterization**:
    - The depth of the dip can give you the planet's size relative to the star. A deeper dip indicates a larger planet.
    - The duration of the dip provides information about the orbital inclination and the planet's speed.
3. **Star Variability**:
    - Variations in the light curve aside from transits can indicate stellar activity such as rotation, pulsations, or flares.

Further Analysis :

To scientifically understand the light curve data more deeply, you can perform the following analyses:

1. **Data Cleaning**:
    - Remove or correct for outliers and noise to get a clearer signal.
2. **Periodicity Analysis**:
    - Use algorithms such as the Lomb-Scargle / Box Least Square periodogram to detect periodic signals in the light curve data.

3. **Transit Modeling**:
    - Fit a transit model to the light curve data to extract parameters such as the planet's radius, orbital period, and inclination.
4. **Stellar Parameters**:
    - Use additional data (e.g., star's temperature, mass, radius) to better understand the context of the light curve and refine your exoplanetary models.

Important note :

We see that other than the flux being normalized, there is no other difference between the plot of `lc_files` and `lc` that is obtained by stitching the lightcurves together using `PDCSAP_FLUX.stitch()` . This is because the `search_result` contains a single data product, ie, a single light curve corresponding to a long cadence (observation made for around 30mins).

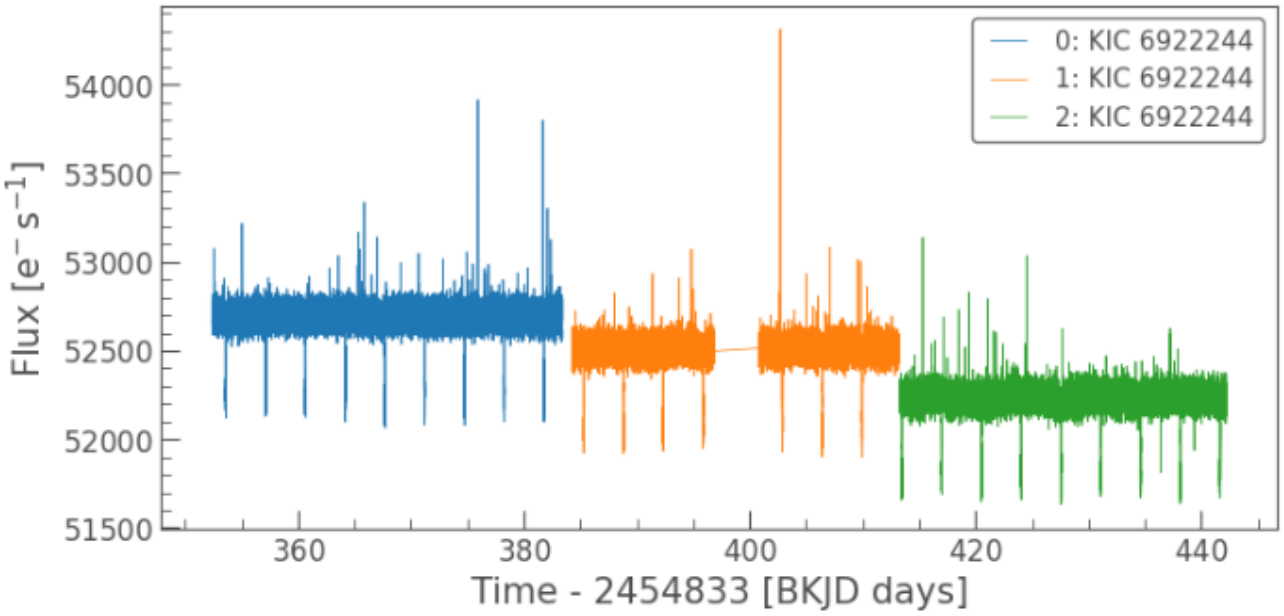
If we had filtered the short cadences, we would've got multiple shorter lightcurves, and stitching them together would show some significance.

The results obtained when the `cadence` was set to be short are shown below :

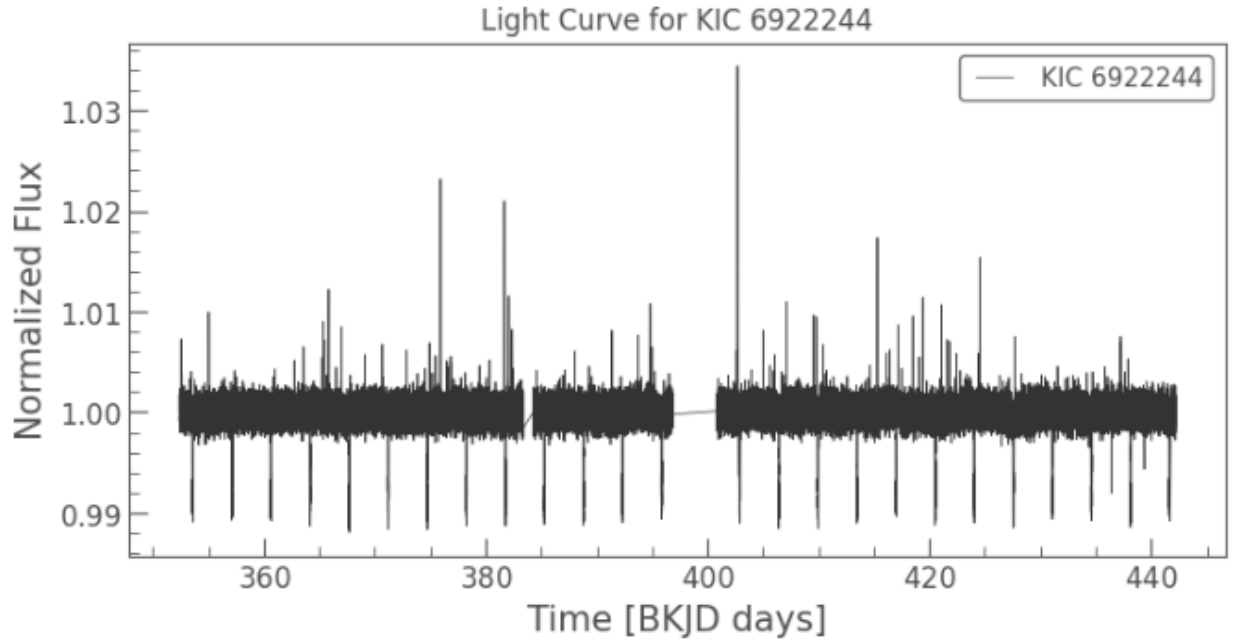
- `search_result` and the downloaded files :

```
SearchResult containing 3 data products.

 #       mission      year author exptime  target_name  distance
                                    s                    arcsec
--- ----------------- ---- ------ ------- ------------- --------
  0 Kepler Quarter 04 2010 Kepler      60 kplr006922244      0.0
  1 Kepler Quarter 04 2010 Kepler      60 kplr006922244      0.0
  2 Kepler Quarter 04 2010 Kepler      60 kplr006922244      0.0
LightCurveCollection of 3 objects:
    0: <KeplerLightCurve LABEL="KIC 6922244" QUARTER=4 AUTHOR=Kepler FLUX_ORIGIN=pdcsap_flux>
    1: <KeplerLightCurve LABEL="KIC 6922244" QUARTER=4 AUTHOR=Kepler FLUX_ORIGIN=pdcsap_flux>
    2: <KeplerLightCurve LABEL="KIC 6922244" QUARTER=4 AUTHOR=Kepler FLUX_ORIGIN=pdcsap_flux>
```

- 3 separate lightcurves corresponding to the downloaded data products (not yet stitched):



- lightcurve after stitching:



You can also view the available light curve data for a particular star from [https://archive.stsci.edu/kepler/data_search/search.php] , and choose any cadence and any quarters to perform the analysis.

Kepler ID	Investigation ID	Dataset Name	Quarter	RA (J2000)	Dec (J2000)	Target Type	Archive Class	Ref	Actual Start Time	Actual End Time	Release Date	R Mag	J Mag	KEP Mag	2MA
6922244	EX	KPLR006922244-2009131105131	0	18 45 09.149	+42 27 03.89	LC	CLC	10	2009-05-02 00:54:56	2009-05-11 17:51:31	2009-12-31 00:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009166043257	1	18 45 09.149	+42 27 03.89	LC	CLC	9	2009-05-13 00:15:49	2009-06-15 11:32:57	2009-12-31 00:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009201121230	2	18 45 09.149	+42 27 03.89	SC	CSC	6	2009-06-20 00:10:56	2009-07-20 19:12:30	2011-02-02 06:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009231120729	2	18 45 09.149	+42 27 03.89	SC	CSC	6	2009-07-20 19:42:54	2009-08-19 19:07:29	2011-02-02 06:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009259160929	2	18 45 09.149	+42 27 03.89	LC	CLC	6	2009-06-20 00:25:09	2009-09-16 23:09:29	2011-02-02 06:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009259162342	2	18 45 09.149	+42 27 03.89	SC	CSC	6	2009-08-20 20:38:32	2009-09-16 23:23:42	2011-02-02 06:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009291181958	3	18 45 09.149	+42 27 03.89	SC	CSC	5	2009-09-18 17:05:45	2009-10-19 01:19:58	2011-09-23 09:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009322144938	3	18 45 09.149	+42 27 03.89	SC	CSC	5	2009-10-19 21:56:47	2009-11-18 22:49:38	2011-09-23 09:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009350155506	3	18 45 09.149	+42 27 03.89	LC	CLC	5	2009-09-18 17:19:58	2009-12-16 23:55:06	2011-09-23 09:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2009350160919	3	18 45 09.149	+42 27 03.89	SC	CSC	5	2009-11-21 00:22:29	2009-12-17 00:09:19	2011-09-23 09:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010019161129	4	18 45 09.149	+42 27 03.89	SC	CSC	5	2009-12-19 20:49:43	2010-01-19 22:13:47	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010049094358	4	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-01-20 19:20:01	2010-02-18 17:43:58	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010078095331	4	18 45 09.149	+42 27 03.89	LC	CLC	5	2009-12-19 21:03:56	2010-03-19 16:53:31	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010078100744	4	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-02-18 19:13:13	2010-03-19 17:07:44	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010111051353	5	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-03-20 23:33:02	2010-04-21 12:13:53	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010140023957	5	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-04-22 18:39:10	2010-05-20 09:39:57	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010174085026	5	18 45 09.149	+42 27 03.89	LC	CLC	5	2010-03-20 23:47:15	2010-06-23 15:50:26	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010174090439	5	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-05-21 02:21:22	2010-06-23 16:04:39	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010203174610	6	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-06-24 22:29:55	2010-07-22 00:45:41	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010234115140	6	18 45 09.149	+42 27 03.89	SC	CSC	5	2010-07-22 20:53:04	2010-08-22 18:51:11	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010265121752	6	18 45 09.149	+42 27 03.89	LC	CLC	6	2010-06-24 22:44:09	2010-09-22 19:03:09	2012-01-07 12:00:00	13.511	12.576	13.563	184509144
6922244	EX	KPLR006922244-2010265121752	6	18 45 09.149	+42 27 03.89	SC	CSC	6	2010-08-23 16:26:48	2010-09-22 19:17:22	2012-01-07 12:00:00	13.511	12.576	13.563	184509144

STEP 6: FEATURE EXTRACTION :

The features that we are going to extract from potential transit signals are :

1. **Transit Depth**
2. **Transit Duration**
3. **Periodicity**

To facilitate the feature extraction process, we convert the light curve data into a Pandas DataFrame. This conversion provides a structured format for easy manipulation and analysis, enabling the application of various data processing techniques and feature extraction methods.

```
# Convert LightCurve to Pandas DataFrame
lc_df = lc_flat.to_pandas()

# Info about the DataFrame
print(lc_df.info())

# Display the first few rows of the DataFrame
print(lc_df.head())

# List all columns in the DataFrame
print(lc_df.columns)
```

Explanation:

1. **Convert LightCurve to Pandas DataFrame**:

```
lc_df = lc_flat.to_pandas()
```

- **lc_flat.to_pandas()**: This function converts the flattened light curve to a Pandas DataFrame. Pandas DataFrames are a powerful data structure that makes it easier to manipulate and analyze data.

2. **Get Information About the DataFrame**:

```
print(lc_df.info())
```

- **lc_df.info()**: This function prints a concise summary of the DataFrame, including the index dtype, column dtypes, non-null values, and memory usage.

```
<class 'pandas.core.frame.DataFrame'>
Index: 4108 entries, 352.3967580484896 to 442.20350409652747
Data columns (total 24 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   flux             4108 non-null   float64
 1   flux_err         4108 non-null   float64
 2   quality          4108 non-null   int32
 3   timecorr         4108 non-null   float32
 4   centroid_col     4108 non-null   float64
 5   centroid_row     4108 non-null   float64
 6   cadenceno        4108 non-null   int32
 7   sap_flux         4108 non-null   float32
 8   sap_flux_err     4108 non-null   float32
 9   sap_bkg          4108 non-null   float32
 10  sap_bkg_err      4108 non-null   float32
 11  pdcsap_flux      4108 non-null   float32
 12  pdcsap_flux_err  4108 non-null   float32
 13  sap_quality      4108 non-null   int32
 14  psf_centr1       4108 non-null   float64
 15  psf_centr1_err   4108 non-null   float32
 16  psf_centr2       4108 non-null   float64
 17  psf_centr2_err   4108 non-null   float32
 18  mom_centr1       4108 non-null   float64
 19  mom_centr1_err   4108 non-null   float32
 20  mom_centr2       4108 non-null   float64
 21  mom_centr2_err   4108 non-null   float32
 22  pos_corr1        4108 non-null   float32
 23  pos_corr2        4108 non-null   float32
dtypes: float32(13), float64(8), int32(3)
memory usage: 545.6 KB
```

3. **Display the First Few Rows of the DataFrame**:

```
print(lc_df.head())
```

- **lc_df.head()**: This function prints the first five rows of the DataFrame, providing a quick look at the data.

```
None
                flux  flux_err  quality  timecorr  centroid_col  centroid_row  \
time
352.396758  1.000244  0.000152     8192 -0.001411    682.690844    190.132704
352.437624  1.000116  0.000152       16 -0.001412    682.690381    190.133024
352.458058  1.000264  0.000152        0 -0.001412    682.690839    190.132567
352.478491  1.000006  0.000152        0 -0.001413    682.689486    190.133042
352.498924  0.999960  0.000152        0 -0.001413    682.690358    190.133177

            cadenceno      sap_flux  sap_flux_err      sap_bkg  ...  \
time                                                            ...
352.396758      11915  43698.070312      6.646242  1669.839966  ...
352.437624      11917  43694.105469      6.646162  1673.130615  ...
352.458058      11918  43698.316406      6.646313  1674.140625  ...
352.478491      11919  43687.648438      6.645916  1672.381470  ...
352.498924      11920  43686.480469      6.645871  1673.437500  ...

            psf_centr1  psf_centr1_err  psf_centr2  psf_centr2_err  \
time
352.396758  682.609001        0.000078  190.367766        0.000118
352.437624  682.608625        0.000078  190.368479        0.000118
352.458058  682.608724        0.000078  190.367920        0.000118
352.478491  682.608290        0.000078  190.368136        0.000118
352.498924  682.608523        0.000078  190.368750        0.000118

            mom_centr1  mom_centr1_err  mom_centr2  mom_centr2_err  pos_corr1  \
time
352.396758  682.690844        0.000186  190.132704        0.000183   0.098700
352.437624  682.690381        0.000186  190.133024        0.000183   0.098307
352.458058  682.690839        0.000186  190.132567        0.000183   0.098410
352.478491  682.689486        0.000186  190.133042        0.000183   0.098039
352.498924  682.690358        0.000186  190.133177        0.000183   0.098103

              pos_corr2
time
352.396758  -0.172299
352.437624  -0.171802
352.458058  -0.172030
352.478491  -0.171838
352.498924  -0.171517
```

4. **List All Columns in the DataFrame**:

```
print(lc_df.columns)
```

- **lc_df.columns**: This attribute prints the names of all columns in the DataFrame.

```
[5 rows x 24 columns]
Index(['flux', 'flux_err', 'quality', 'timecorr', 'centroid_col',
       'centroid_row', 'cadenceno', 'sap_flux', 'sap_flux_err', 'sap_bkg',
       'sap_bkg_err', 'pdcsap_flux', 'pdcsap_flux_err', 'sap_quality',
       'psf_centr1', 'psf_centr1_err', 'psf_centr2', 'psf_centr2_err',
       'mom_centr1', 'mom_centr1_err', 'mom_centr2', 'mom_centr2_err',
       'pos_corr1', 'pos_corr2'],
      dtype='object')
```

Threshold Method and Box Least Squares (BLS) Method :

Threshold Method:

The threshold method is a straightforward approach to detect transits in light curve data by identifying times when the flux drops below a predefined threshold.

```
# Convert the light curve to a NumPy array
time = lc_flat.time.value
flux = lc_flat.flux.value

# Detect potential transits using a simple threshold
threshold =  0.99184291  # Adjust this threshold as needed
transit_mask = flux < threshold

# Find local minima within the transit mask
peaks, _ = find_peaks(-flux, height=-threshold)

# Check if any transits were detected
if np.sum(transit_mask) == 0:
    print("No transits detected with the given threshold.")

else:
    # Extract the features: depth, duration, and periodicity

    # Initialize lists to hold the results for each transit
    transit_durations = []
    ingress_starts = []
    egress_ends = []
    # Identify contiguous segments of the transit mask
    transit_indices = np.where(transit_mask)[0]    #indices where transit_mask
is True
    split_points = np.where(np.diff(transit_indices) > 1)[0] + 1
    transit_segments = np.split(transit_indices, split_points)
    # Calculate ingress start and egress end for each transit
    for segment in transit_segments:
        ingress_start = time[segment[0]]
        egress_end = time[segment[-1]]
        transit_duration = egress_end - ingress_start
        ingress_starts.append(ingress_start)
        egress_ends.append(egress_end)
        transit_durations.append(transit_duration)

    transit_depth = 1-flux[peaks].min()
    transit_periodicity = np.diff(time[peaks]).mean()

    features = {
        'depth': transit_depth,
        'duration': transit_duration,
        'periodicity': transit_periodicity
    }

    print(features)
    ```

1. **Convert Light Curve to NumPy Arrays**:
```
```

   - The light curve's time and flux values are extracted into NumPy arrays for easier manipulation.

2. **Threshold Setting**:
   - A threshold value of 0.99184291 is set. This value should be adjusted based on the light curve data to effectively detect transits, meaning it should cover just the bottom most transit points.
   - But in general, a threshold value is set, typically based on the mean flux minus a multiple of the standard deviation. For example, if the mean flux is μ and the standard deviation is σ, a threshold might be μ−3σ .

3. **Transit Mask**:
   - A boolean mask is created where the flux values are below the threshold, indicating potential transit events.

4. **Finding Local Minima**:
   - - The `find_peaks` function from the `scipy.signal` module is used to find local minima within the flux data that are lower than the threshold. These minima are likely to be the center points of the transit events.
   - We find this because the transit mask found above filters even those flux that are very nearby to the flux that is actually associated with the transit (at the dips). Thus finding the local minima's of each dip shall be beneficial than the transit mask itself to get accurate values of the features.

5. **Check for Transits**:
   - If no transits are detected (no flux values below the threshold), a message is printed - "No transits detected with the given threshold.".
   - If transits are detected, features such as depth, duration, and periodicity are calculated:
     - **Depth**: The minimum flux value during the transit.

     - **Duration**: The time at which the minimum flux occurs.
         - **Measuring the Transit Duration**:
            - **Ingress Start (t1)**: Determine the time at which the light curve first crosses below the threshold. This is the start of the transit ingress.
            - **Ingress End (t2)**: Find the time at which the light curve reaches the minimum brightness level (the bottom of the transit).
            - **Egress Start (t3)**: Identify the time at which the light curve starts rising from the minimum brightness level.
            - **Egress End (t4)**: Determine the time at which the light curve crosses back above the threshold, marking the end of the transit egress.
            - **Calculate Transit Duration**: The total transit duration can be calculated as the difference between the times when the light curve enters and exits the transit threshold.
                        Transit Duration=t4−t1
         - **Identifying Contiguous Segments**: The indices where the `transit_mask` is true are   identified. These indices are split into contiguous segments where the difference between consecutive indices is more than 1. Each segment corresponds to a single transit event.

The `transit_segment` holds the following segments corresponding to each transits :
[array([56, 57, 58]), array([226, 227, 228, 229]), array([397, 398, 399]), array([568, 569, 570, 571]), array([738, 739, 740, 741]), array([909, 910, 911, 912]), array([1077, 1078, 1079, 1080]), array([1249, 1250, 1251]), array([1419, 1420, 1421]), array([1546, 1547, 1548, 1549]), array([1716, 1717, 1718]), array([1886, 1887, 1888, 1889]), array([2056, 2057, 2058, 2059]), array([2204, 2205, 2206, 2207]), array([2374, 2375, 2376, 2377]), array([2546, 2547, 2548, 2549]), array([2713, 2714, 2715, 2716]), array([2883, 2884, 2885]), array([3054, 3055, 3056, 3057]), array([3226, 3227]), array([3396, 3397, 3398, 3399]), array([3567, 3568, 3569]), array([3739, 3740, 3741]), array([3909, 3910, 3911, 3912]), array([4080, 4081, 4082, 4083])]
- **Calculating Ingress and Egress Times**: For each contiguous segment, the start (`ingress_start`) and end (`egress_end`) times are calculated as the first and last times within the segment. The transit duration is then calculated as the difference between these times.

- **Periodicity**: The average time difference between consecutive transits.

Box Least Squares (BLS) Method:

The BLS method is a more advanced technique that searches for periodic transits by fitting a box-shaped model to the light curve data. Here's how it is implemented in the provided code:

```python
# Use Lightkurve's built-in transit detection (BLS Method)
# Assumed period in days, adjust based on your knowledge or data
periods = np.linspace(1, 5, 10000)  # Example: searching between 1 and 5 days with 10000 steps
bls = lc_flat.to_periodogram(method='bls', period=periods, frequency_factor=500)

# Plot the BLS periodogram to inspect potential transit signals
bls.plot()
plt.title('BLS power spectrum')
plt.show()

# Extract the best-fit period and related attributes
best_power = bls.max_power
transit_time = bls.transit_time_at_max_power

print(f"Best power: {best_power}")
print(f"Transit time at max power: {transit_time}")

# Calculate the depth, duration, and periodicity of detected transits
bls_transit_depth = bls.depth_at_max_power
bls_transit_duration = bls.duration_at_max_power.to_value('d')
bls_transit_periodicity = bls.period_at_max_power.to_value('d')
```

```
bls_features = {
    'depth': bls_transit_depth,
    'duration': bls_transit_duration,
    'periodicity': bls_transit_periodicity
}

print(bls_features)
```

1. **Define Period Grid**:
   - A range of trial periods is defined using `np.linspace`. Here, the search is between 1 and 5 days with 10,000 steps.
2. **Calculate BLS Periodogram**:
   - Using Lightkurve's `to_periodogram` method, the BLS periodogram is computed. This method scans the light curve for periodic signals that match the box-shaped transit model.
   - **Frequency Factor**: it refers to the resolution and range of the periods tested. It involves the number of trial periods (or frequencies) tested over a specified range, affecting the precision and computational effort of the search.
3. **Plot BLS Periodogram**:
   - The periodogram is plotted to visualize the power spectrum, which highlights the most significant periodic signals.
4. **Extract Best-Fit Period and Attributes**:
   - The period with the maximum power is identified as the best-fit period. This period is likely the orbital period of the potential exoplanet.
   - The transit time at maximum power, the depth of the transit, and the duration of the transit are extracted.

Explanation:

- **Best-Fit Period**: `best_period = bls.period_at_max_power` gives the period at which the BLS algorithm found the maximum power, indicating the most likely period of the transiting planet.
- **Best Power**: `best_power = bls.max_power` provides the power of the signal at the best-fit period.
- **Transit Time**: `transit_time = bls.transit_time_at_max_power` gives the time of the first transit at the best-fit period.
- **Depth, Duration, and Periodicity**: These are extracted directly from the `BoxLeastSquaresPeriodogram` object.
- `to_value('d')` converts the quantities to a scalar value in days.

5. **Calculate Features**:
   - **Depth**: The depth of the transit at the best-fit period.

- **Duration**: The duration of the transit at the best-fit period.
- **Periodicity**: The period corresponding to the maximum power in the periodogram.

Summary:

Both methods aim to identify transits in the light curve data, but they do so in different ways:

- The **Threshold Method** is simpler and faster, relying on a predefined threshold to detect dips in flux. It is useful for a quick initial scan but may miss subtle transits.
- The **BLS Method** is more sophisticated, fitting a box-shaped model to the data and searching for periodic signals. It is more sensitive and accurate, making it suitable for detailed analysis and confirmation of potential exoplanet transits.

Output Obtained From Threshold and BLS Method :

Result of Threshold method:

```
{'depth': MaskedNDArray(0.00953973), 'duration': 0.06130308889987646, 'periodicity': 3.3873035987492655}
```

{'depth': MaskedNDArray(0.00953973), 'duration': 0.06130308889987646, 'periodicity': 3.3873035987492655}

Result of BLS method:

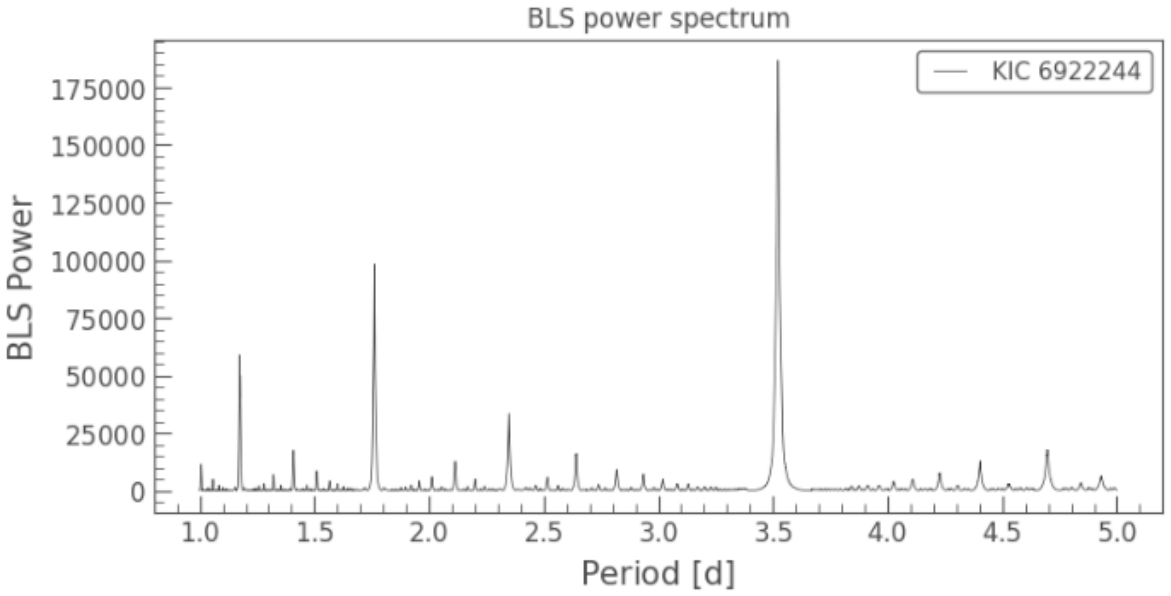
```
Best power: 186578.47219238934
Transit time at max power: 353.60175804848956
{'depth': <Quantity 0.00845154>, 'duration': 0.1, 'periodicity': 3.522652265226523}
```

Best power: 186578.47219238934
Transit time at max power: 353.60175804848956
{'depth': <Quantity 0.00845154>, 'duration': 0.1, 'periodicity': 3.522652265226523}

	CONFIRMED VALUES	VALUES OBTAINED FROM THRESHOLD METHOD	VALUES OBTAINED FROM BLS METHOD
TRANSIT DEPTH (%)	0.937	0.953	0.845
TRANSIT DURATION (in hours)	3.31	1.44	2.4
PERIODICITY (in days)	3.522	3.387	3.522

We see that both the methods give closer values for each features. But BLS method is more reliable!

BLS Plot :



- **X-axis (Period [d])**: This represents the trial periods (in days) tested by the BLS algorithm.
- **Y-axis (BLS Power)**: This shows the signal power for each trial period. Peaks in this graph indicate periods where potential transit signals were found.
- **Peaks**: Each significant peak represents a candidate period where the BLS algorithm detected a potential transit. The highest peak corresponds to the best period with the strongest transit signal.

Observations:

- **High Peak at 3.5 days**: This indicates a strong potential transit signal at a period of 3.5 days. This is the best-fit period found by the BLS method.
- **Smaller Peaks**: Other smaller peaks might indicate weaker potential signals or harmonics of the main period.

This plot is a visual representation of the BLS power spectrum, and the highest peak is typically interpreted as the most likely period for the detected exoplanet.

Choosing best `threshold` and `periods` :

- **Threshold Method**:
  - **Threshold Choice**: The threshold is typically chosen based on the noise level of the data. A common practice is to set the threshold slightly above the noise level to detect significant dips.
- **BLS Method**:

- **Periods Choice**: The array of periods should span the expected range of orbital periods for the exoplanets you are interested in.
- **Steps in Period Array**: The "steps" refer to the number of periods within the specified range. For example, `np.linspace(1, 5, 10000)` means testing 10000 periods between 1 and 5 days. Each number in this array represents a trial period at which the BLS algorithm will test for transits. In the BLS method, the period range is specified to search for periodic transits. In the provided code, periods are checked between 1 and 5 days, making it suitable for detecting exoplanets with short orbital periods. This range is particularly effective for identifying planets that orbit their host stars quickly, such as those close to their stars. To detect exoplanets with longer orbital periods, which are farther from their stars and take more time to complete one orbit, we can increase the range of periods to check. Adjusting the period range allows for the detection of a wider variety of exoplanets, including those with longer orbits.

Explanation of Phase Folding and Threshold Method Visualization:

Phase Folding the Light Curve (BLS Method) :

```
# Phase-fold the light curve to verify that the period and transit time
corresponds to the transit signal
# This puts the frequency spikes on top of each other if we got the
periodicity right in bls method

folded_lc =
lc_flat.fold(period=bls_transit_periodicity,epoch_time=transit_time).scatter()
folded_lc.set_xlim(-2,2)
folded_lc.plot()

# The depth can tell you about the size, etc
```

Code Explanation:

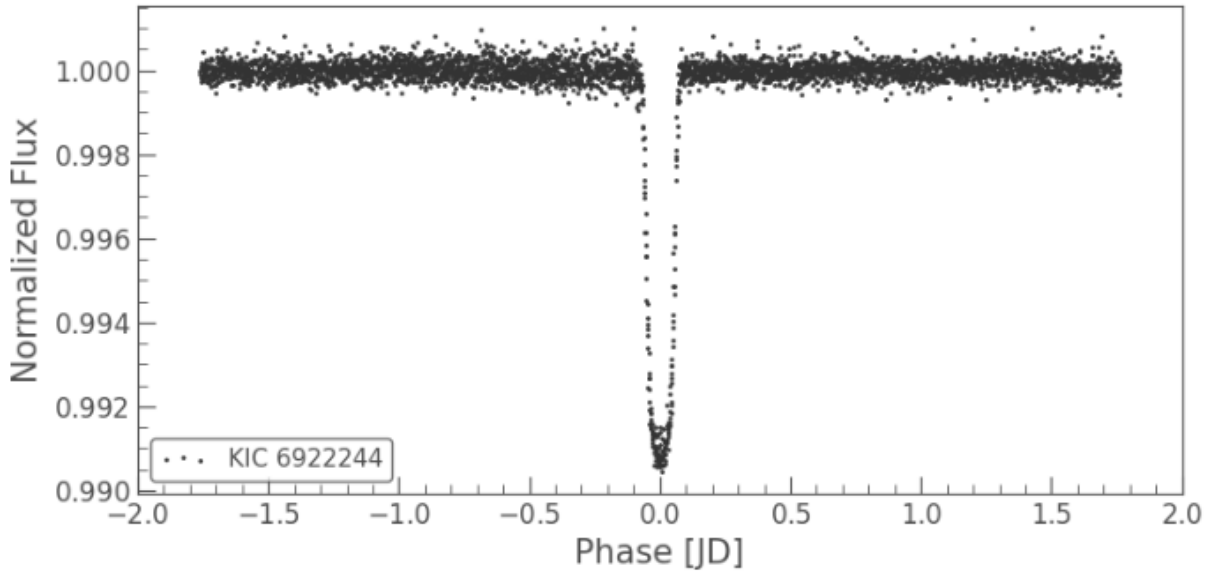
1. **Phase-fold the light curve**:
   - The purpose of phase folding is to align the data points such that all occurrences of the periodic signal (e.g., transits) are superimposed. This helps in verifying the periodicity and characteristics of the detected transit signal.
   - `lc_flat.fold(period=bls_transit_periodicity, epoch_time=transit_time)` takes the flattened light curve and folds it using the period and transit time detected by the BLS method.
   - `.scatter()` creates a scatter plot of the folded light curve.
   - `folded_lc.set_xlim(-2, 2)` sets the x-axis limits for better visualization.

- **NOTE :** The same can be done with the result obtained from Threshold method for verification.

2. **Plot the folded light curve**:
    - The folded light curve plot shows how the flux varies within one period of the detected signal.
    - The depth of the dip in the flux (transit) can indicate the size of the planet relative to its host star.

Output Explanation:

- The folded light curve plot visualizes the transit signal. The significant dip at phase 0 represents the detected transit, confirming the periodicity and providing a clear view of the transit's depth and duration.



Inference:

- This plot allows us to see multiple transits superimposed, verifying that the periodicity and transit signal detected by the BLS method are consistent. The depth of the transit can provide insights into the size of the planet relative to the star.

Threshold Method Visualization :

```
# Plot the normalized light curve
plt.figure(figsize=(10, 5))
plt.plot(time, flux, label='Normalized Flux', color='black')
plt.scatter(time[peaks], flux[peaks], color='red', s=10, label='Transit
Points')

# Marking the transit points with red dots
plt.xlabel('Time [BKJD]')
plt.ylabel('Normalized Flux')
plt.title(f'Normalized Light Curve for {target}')
```

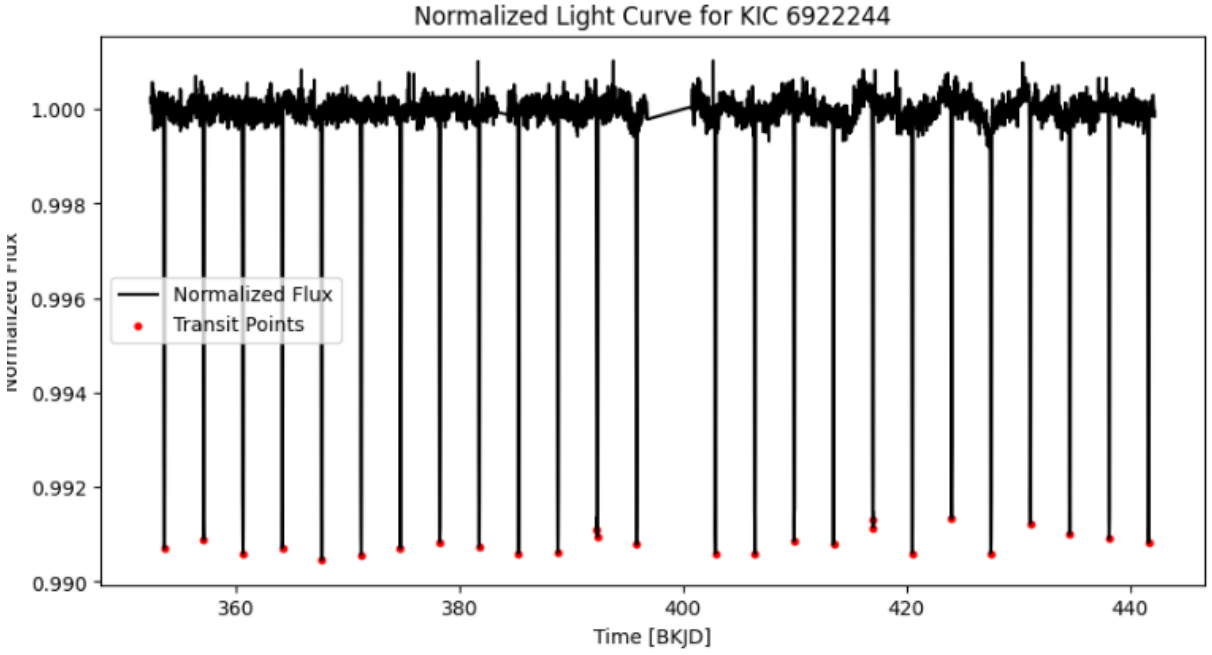
```
plt.legend()
plt.show()
```

Code Explanation:

1. **Plot the normalized light curve**:
   - `plt.figure(figsize=(10, 5))` creates a new figure with a specified size.
   - `plt.plot(time, flux, label='Normalized Flux', color='black')` plots the normalized light curve.
   - `plt.scatter(time[peaks], flux[peaks], color='red', s=10, label='Transit Points')` marks the detected transit points (those filtered by the threshold method) with red dots.
   - Labels and title are added for clarity.

Output Explanation:

- The normalized light curve plot shows the star's brightness over time, with detected transit points marked in red.
- The red dots indicate the positions where the flux dips below the specified threshold, suggesting potential transits.



Inference:

- This visualization helps in identifying the points where potential transits occur based on the threshold method.
- The regularity of the red dots suggests a consistent periodic signal, supporting the hypothesis of a transiting exoplanet.

Summary

- **Phase Folding**: Aligns the data points of the light curve to reveal the periodicity and characteristics of detected transits, confirming the results from the BLS method.
- **Threshold Method Visualization**: Marks the points on the light curve where potential transits occur, helping to visually confirm the presence of a periodic transit signal.

Both methods are crucial for validating and understanding the detected transits, providing insights into the size, duration, and periodicity of potential exoplanets.

ADDITIONAL STUDY - EXTENDING THE ANALYSIS TO LIGHTCURVES OF MULTI-PLANETARY SYSTEMS :

Multi-Planetary Systems

Multi-planetary systems are star systems that host more than one planet. These systems are of great interest to astronomers because they provide valuable information about planet formation and dynamics. Studying these systems can help us understand the arrangement and interaction of planets in a shared gravitational environment.

Light Curves of Multi-Planetary Systems

The light curve of a star with multiple planets will show multiple sets of periodic dips in brightness, each corresponding to a different planet. These dips occur when a planet passes (or transits) in front of its host star from our perspective, temporarily blocking some of the star's light.

- **Single-Planet Transit**: A single planet creates a periodic dip in the light curve.
- **Multi-Planet Transits**: Multiple planets create multiple sets of dips, which can sometimes overlap or be close to each other depending on the orbital periods of the planets.

Identifying these multiple transits in a light curve involves analyzing the data to find the periodic dips and then characterizing each detected signal to determine the properties of the planets.

Methods for Extracting Planet Features

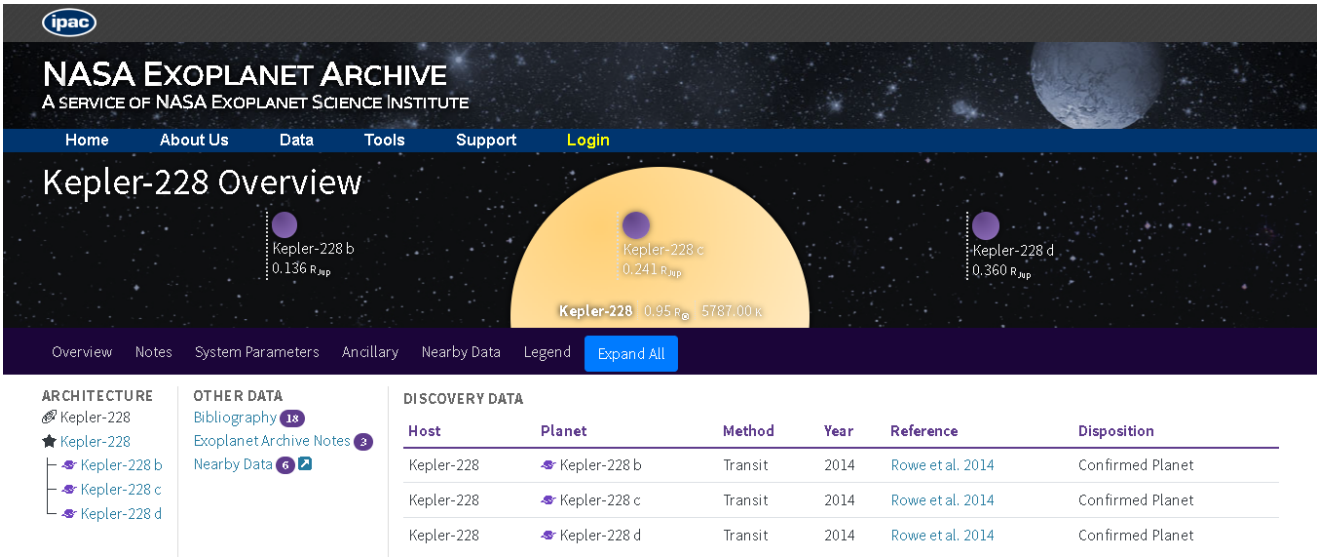
To extract the features of all the planets in a multi-planetary system, we can use two main methods:

1. **Box-Least Squares (BLS) Method**: Perform the BLS method iteratively for various periods to detect planets with long or short orbits.

2. **Masking Method**: Iteratively mask the light curve based on the planets already found to identify additional planets.

NOTE : The threshold method used earlier to detect a single exoplanet is not suitable to analyze multi-planetary systems. Further study has to be made to adopt a method like the Threshold method for multi-planetary systems.

To analyze a multi-planetary system, we use the below mentioned star whose presence of multiple planets is already confirmed



Star : Kepler-228
KeplerID : KIC 10872983
Planet(s) : Kepler-228 b, Kepler-228 c, Kepler-228 d

To detect multiple exoplanets just by iteratively performing BLS method for various periods, we have to know the confirmed orbital periods of the planets prior to the analysis. We find the orbital periods of Kepler-228 b, Kepler-228 c, Kepler-228 d are 2.5, 4.1 and 11 days respectfully. Therefore we can test the presence of these planets and obtain their features using the following periods :

- `periods = np.linspace(2, 3, 10000)`, for Kepler-228 b
- `periods = np.linspace(4, 6, 10000)`, for Kepler-228 c
- `periods = np.linspace(10, 13, 10000)`, for Kepler-228 d

Although, we shall use the **Masking method** to detect and extract features for multiple exoplanets belonging to the same star system , which is explained in detail below.

Masking Method to analyze Multi-planetary systems :

Masking is a technique used in transit detection to iteratively identify and analyze multiple transits in a light curve, especially when there are multiple exoplanets causing different transit

signals. Once a transit is detected, its data points are "masked" or removed, allowing the detection algorithm to find additional transits that may have been obscured by the first.

Steps for Masking Technique:

1. **Detect the First Transit**:
   - Use a transit detection algorithm (e.g., BLS) to find the first set of transit parameters.
2. **Mask the Detected Transit**:
   - Remove or mask the data points corresponding to the detected transit from the light curve.
3. **Repeat Detection on the Masked Light Curve**:
   - Run the BLS algorithm again on the masked light curve to find additional transits.
4. **Iterate Until No More Transits Are Detected**:
   - Continue the process until no more significant transits are found.

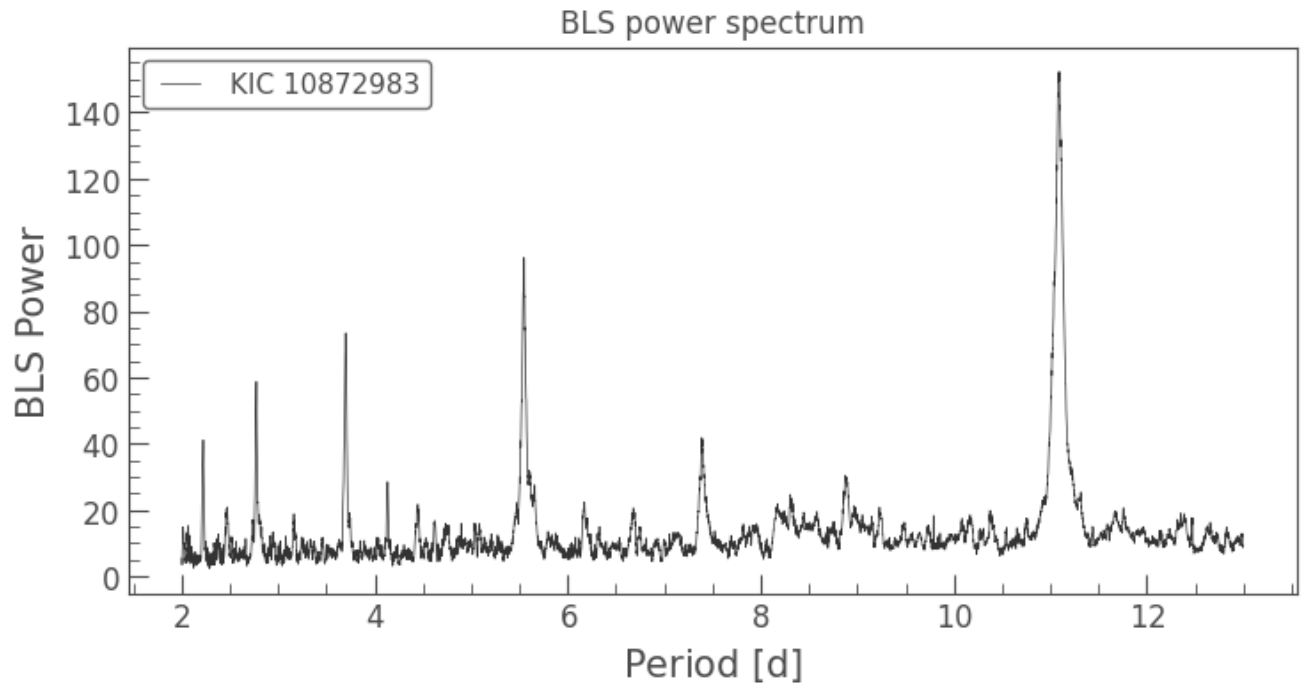
Below is the code for performing these tasks using the masking method, starting with the initial BLS method to find the first planet's features and then iteratively masking the light curve to find subsequent planets.

Analyzing the presence of First Planet :

- **Initial BLS Search**: The Box Least Squares (BLS) method is used to identify the first planet's period, transit time, depth, and duration. The same code used for single planetary system earlier can be used here .
- Analyzing the existing data , we find that all the Kepler-228's planets have their orbital periods <13 days. Therefore we perform the bls method with `periods = np.linspace(2, 13, 10000)`

Results :

BLS Spectrum :



The BLS Power Spectrum suggests that strong potential transit signal is observed at a period of 11 days. The feature values obtained are given below :

```
Best power: 152.23001266237472
Transit time at max power: 737.0242899629526
{'depth': <Quantity 0.0013101>, 'duration': 0.2, 'periodicity': 11.091309130913091}
```

This corresponds to the planet **Kepler-228 d** , as shown below :

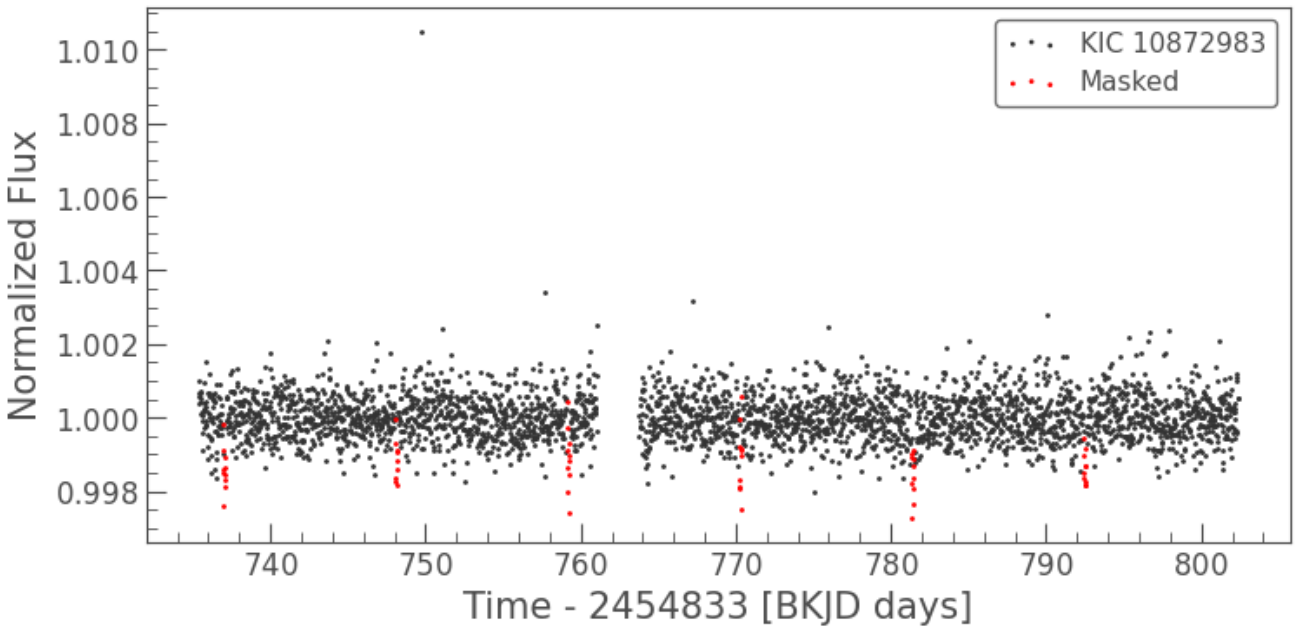
	CONFIRMED VALUES	VALUES OBTAINED FROM BLS METHOD
TRANSIT DEPTH (%)	0.1764	0.131
TRANSIT DURATION (in hours)	4.5	4.8
PERIODICITY (in days)	11.094	11.091

Analyzing the presence of Second Planet :

- **Masking the First Planet**: The light curve is masked to exclude the transits of the first planet, allowing for the detection of additional planets without interference.

```
# Create a cadence mask using the BLS parameters
planet_mask_1 = bls.get_transit_mask(period=bls_transit_periodicity,
                                     transit_time=transit_time,
                                     duration=bls_transit_duration)
masked_lc_1 = lc_flat[~planet_mask_1]
```

```
ax = masked_lc_1.scatter();
lc_flat[planet_mask_1].scatter(ax=ax, c='r', label='Masked');
```



Plot Explanation :

Black Dots (Original Data):
- These represent the observed flux of the star over time. The black dots show the light curve data before any masking is applied.

Red Dots (Masked Data):
- These represent the data points that correspond to the transits of the first detected planet. By masking, these points are identified and marked separately to indicate where the transit of the first planet occurs.

Interpretation :

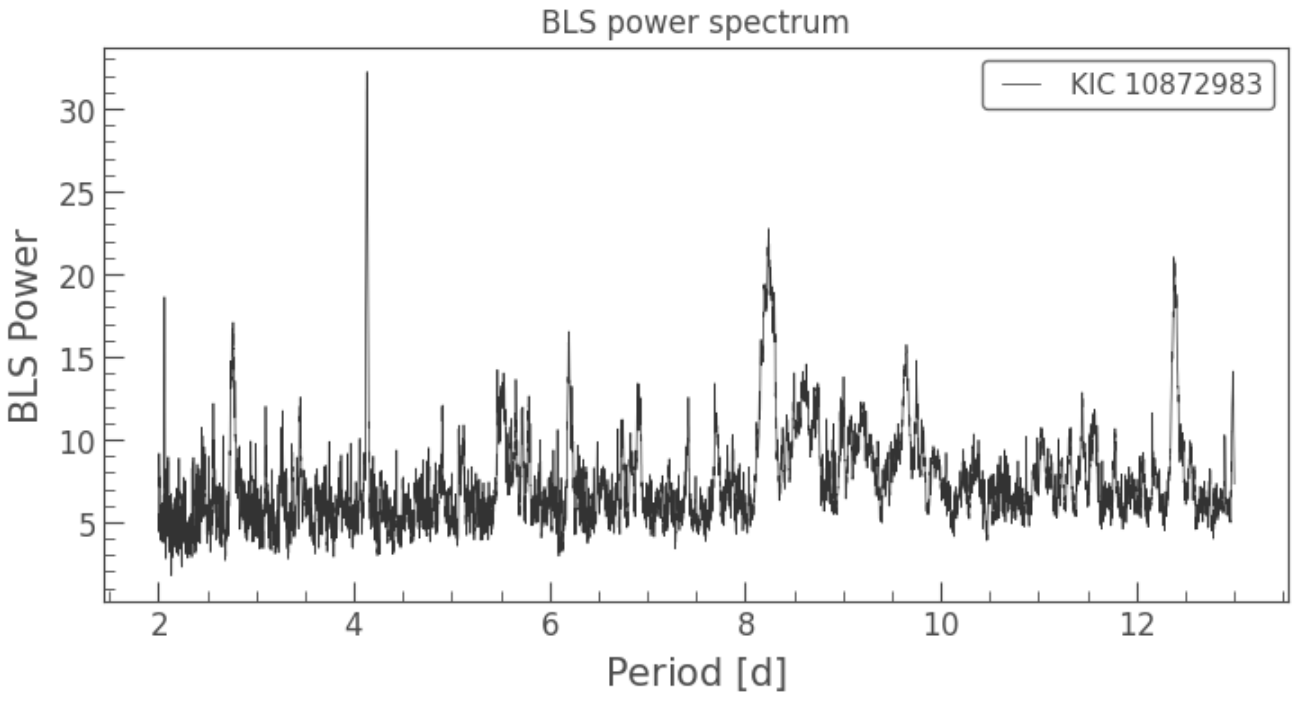
- **Detected Transits**: The red dots indicate the positions of the transits of the first detected planet. Each red dot represents a point in the light curve where the planet is transiting the star, causing a dip in brightness.
- **Remaining Light Curve**: The black dots, after removing the red dots (masked transits), show the remaining light curve. This remaining curve can now be analyzed further to detect additional planets without interference from the already detected planet's transits.
- **Second BLS Search**: BLS is performed on the masked light curve to detect the second planet. The process is similar to the first search but applied to the masked light curve.

```
period = np.linspace(2, 13, 10000)
bls_1 = masked_lc_1.to_periodogram('bls', period=period, frequency_factor=500)
bls_1.plot()
```

```
plt.title('BLS power spectrum')
plt.show()

# Extract the best-fit period and related attributes
best_power_1 = bls_1.max_power
transit_time_1 = bls_1.transit_time_at_max_power

print(f"Best power: {best_power_1}")
print(f"Transit time at max power: {transit_time_1}")

# Calculate the depth, duration, and periodicity of detected transits
bls_transit_depth_1 = bls_1.depth_at_max_power
bls_transit_duration_1 = bls_1.duration_at_max_power.to_value('d')
bls_transit_periodicity_1 = bls_1.period_at_max_power.to_value('d')

bls_features_1 = {
    'depth': bls_transit_depth_1,
    'duration': bls_transit_duration_1,
    'periodicity': bls_transit_periodicity_1
}

print(bls_features_1)
```

Results :

BLS Spectrum :



This BLS Power Spectrum suggests that strong potential transit signal is observed at a period of 4.1 days. The feature values obtained are given below :

```
Best power: 32.22801615805955
Transit time at max power: 739.4042899629526
{'depth': <Quantity 0.00042899>, 'duration': 0.15, 'periodicity': 4.133113311331133}
```

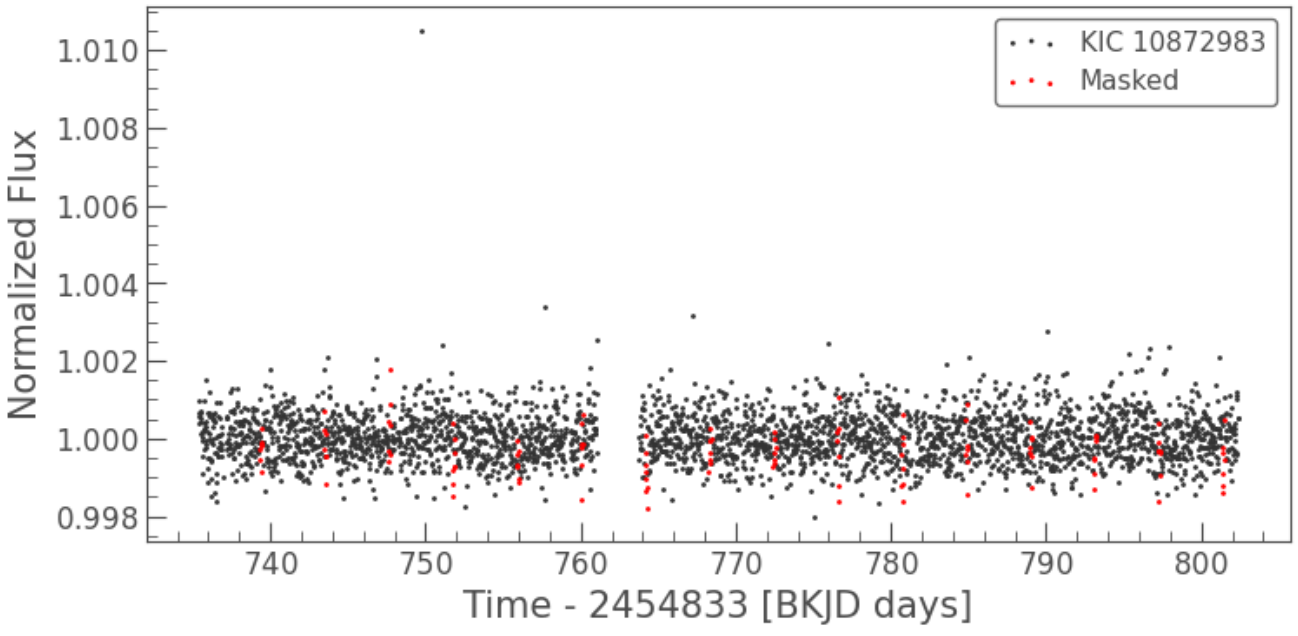
This corresponds to the planet **Kepler-228 c** , as shown below :

	CONFIRMED VALUES	VALUES OBTAINED FROM BLS METHOD
TRANSIT DEPTH (%)	0.0793	0.0428
TRANSIT DURATION (in hours)	3.17	3.6
PERIODICITY (in days)	4.134	4.133

Analyzing the presence of Third Planet :

- **Masking the Second Planet**: The light curve is further masked to exclude transits of both the first and second planets.

```
planet_mask_2 = bls_1.get_transit_mask(period=bls_transit_periodicity_1,
                                       transit_time=transit_time_1,
                                       duration=bls_transit_duration_1)

masked_lc_2 = masked_lc_1[~planet_mask_2]
ax = masked_lc_2.scatter();
masked_lc_1[planet_mask_2].scatter(ax=ax, c='r', label='Masked');
```

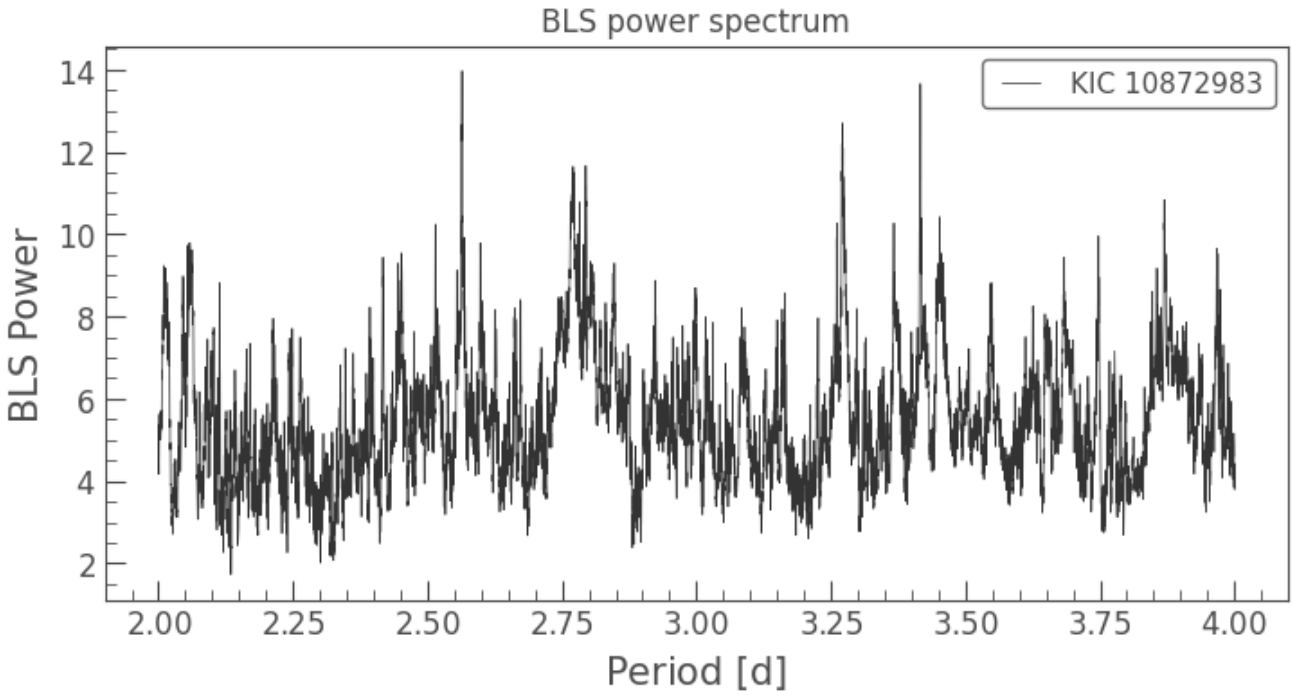


- **Third BLS Search**: BLS is performed on the double-masked light curve to detect the third planet.

```
period = np.linspace(2, 4, 10000)
bls_2 = masked_lc_2.to_periodogram('bls', period=period, frequency_factor=500)
bls_2.plot()
plt.title('BLS power spectrum')
plt.show()

# Extract the best-fit period and related attributes
best_power_2 = bls_2.max_power
transit_time_2 = bls_2.transit_time_at_max_power

print(f"Best power: {best_power_2}")
print(f"Transit time at max power: {transit_time_2}")

# Calculate the depth, duration, and periodicity of detected transits
bls_transit_depth_2 = bls_2.depth_at_max_power
bls_transit_duration_2 = bls_2.duration_at_max_power.to_value('d')
bls_transit_periodicity_2 = bls_2.period_at_max_power.to_value('d')

bls_features_2 = {
    'depth': bls_transit_depth_2,
    'duration': bls_transit_duration_2,
    'periodicity': bls_transit_periodicity_2
}

print(bls_features_2)
```

Results :

BLS Spectrum :



This BLS Power Spectrum suggests that strong potential transit signal is observed at a period of 2.5 days. The feature values obtained are given below :

```
Best power: 13.963991681414125
Transit time at max power: 736.5242899629526
{'depth': <Quantity 0.00040436>, 'duration': 0.05, 'periodicity': 2.564256425642564}
```

This corresponds to the planet **Kepler-228 b** , as shown below :

	CONFIRMED VALUES	VALUES OBTAINED FROM BLS METHOD
TRANSIT DEPTH (%	-	0.0404
TRANSIT DURATION (in hours)	2.3	1.2
PERIODICITY (in days)	2.566	2.564

Should we proceed looking for another planet ?

The answer depends on the BLS power observed corresponding to various periods in the BLS Power plot. Although sometimes, the lightcurve, and subsequently the BLS power, might be affected due to Instrumental artifacts, Stellar Activity, issues with data processing etc. The analysis can be proceeded to detect more planets if there are more significant peaks in the BLS power plot.

Reasons for deviation of BLS values from the confirmed values as masking is increased : ;

Detecting multiple exoplanets in a single light curve is inherently more complex than detecting a single planet. Several factors can contribute to the deviation in the derived features for the second and third planets. Here are some potential reasons for the deviation and strategies to tackle them:

Reasons for Deviation

1. **Overlapping Transits**:
   - The light curve might contain overlapping transits, especially if the planets have similar periods or transit durations. This overlap can distort the signal and make it harder to accurately determine the parameters of individual transits.
2. **Transit Timing Variations (TTVs)**:
   - In multi-planetary systems, gravitational interactions between planets can cause variations in the timing of transits. These variations can lead to inaccuracies when fitting transits if not properly accounted for.
3. **Signal-to-Noise Ratio (SNR)**:
   - The signal of additional planets may be weaker compared to the first planet, leading to a lower SNR. This makes it harder to detect and accurately characterize subsequent planets.
4. **Residual Noise After Masking**:
   - Masking transits can leave residual noise or distortions in the light curve, which can affect the detection of subsequent planets.
5. **Incorrect Masking**:
   - If the mask does not perfectly align with the actual transits, it can either leave parts of the transits unmasked or mask non-transit data, leading to errors in subsequent analysis.

Strategies to Improve Detection

1. **Refining the Masking Process**:
   - Ensure the masking process accurately identifies and removes the full duration of the transit, including ingress and egress phases. You may need to manually inspect and adjust the masks.
2. **Multiple Iterations and Fine-Tuning**:
   - Perform multiple iterations of the BLS method with different initial conditions and parameters. Fine-tuning the period range and resolution can help in better detecting weaker signals.

3. **Advanced Detrending**:
    - Apply advanced detrending techniques to remove residual trends and noise from the light curve. This can improve the accuracy of detecting subsequent transits.
4. **Combining BLS with Other Methods**:
    - Use complementary methods like transit timing variations (TTV) analysis or machine learning techniques to cross-validate and refine the detected parameters.
5. **Stacking Transits**:
    - Stack multiple detected transits to improve the SNR. This can help in better visualizing and fitting the transit model, especially for weaker signals.

TRANSITION TO MACHINE LEARNING MODEL TRAINING AND TESTING :

After successfully completing the analysis of lightcurve data to detect potential exoplanets, we now transition to the machine learning phase. In this phase, we will utilize a dataset provided by NASA's Exoplanet Archive, which contains information on confirmed, candidate, and false positive exoplanets from the Kepler mission. The objective is to train a machine learning model to predict the status of exoplanets based on this data.

Objective

The goal is to develop and evaluate machine learning models that can classify exoplanets into categories such as confirmed, candidate, and false positive. By leveraging the dataset from NASA, we aim to create a reliable predictive model that can assist in identifying exoplanetary candidates with higher accuracy.

Initial Setup: Loading and Exploring the Dataset

First, we load the dataset and examine its structure to understand the available features and their types. This step helps us identify any necessary preprocessing steps such as handling missing values, dropping irrelevant columns, and converting categorical data to numerical formats.

Code to Load and Explore the Dataset:

```
import pandas as pd

# Load the dataset
data = pd.read_csv('/content/cumulative.csv', skiprows=55)

# Display basic information about the dataset
```

```
data.info()

# Display the first few rows of the dataset
data.head()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9564 entries, 0 to 9563
Data columns (total 49 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   kepid               9564 non-null   int64
 1   kepoi_name          9564 non-null   object
 2   kepler_name         2745 non-null   object
 3   koi_disposition     9564 non-null   object
 4   koi_pdisposition    9564 non-null   object
 5   koi_score           8054 non-null   float64
 6   koi_fpflag_nt       9564 non-null   int64
 7   koi_fpflag_ss       9564 non-null   int64
 8   koi_fpflag_co       9564 non-null   int64
 9   koi_fpflag_ec       9564 non-null   int64
 10  koi_period          9564 non-null   float64
 11  koi_period_err1     9110 non-null   float64
 12  koi_period_err2     9110 non-null   float64
 13  koi_time0bk         9564 non-null   float64
 14  koi_time0bk_err1    9110 non-null   float64
 15  koi_time0bk_err2    9110 non-null   float64
 16  koi_impact          9201 non-null   float64
 17  koi_impact_err1     9110 non-null   float64
 18  koi_impact_err2     9110 non-null   float64
 19  koi_duration        9564 non-null   float64
 20  koi_duration_err1   9110 non-null   float64
 21  koi_duration_err2   9110 non-null   float64
 22  koi_depth           9201 non-null   float64
 23  koi_depth_err1      9110 non-null   float64
 24  koi_depth_err2      9110 non-null   float64
 25  koi_prad            9201 non-null   float64
 26  koi_prad_err1       9201 non-null   float64
 27  koi_prad_err2       9201 non-null   float64
 28  koi_teq             9201 non-null   float64
 29  koi_teq_err1        0 non-null      float64
 30  koi_teq_err2        0 non-null      float64
 31  koi_insol           9243 non-null   float64
 32  koi_insol_err1      9243 non-null   float64
 33  koi_insol_err2      9243 non-null   float64
 34  koi_model_snr       9201 non-null   float64
 35  koi_tce_plnt_num    9218 non-null   float64
 36  koi_tce_delivname   9218 non-null   object
 37  koi_steff           9201 non-null   float64
 38  koi_steff_err1      9096 non-null   float64
 39  koi_steff_err2      9081 non-null   float64
 40  koi_slogg           9201 non-null   float64
 41  koi_slogg_err1      9096 non-null   float64
 42  koi_slogg_err2      9096 non-null   float64
 43  koi_srad            9201 non-null   float64
 44  koi_srad_err1       9096 non-null   float64
 45  koi_srad_err2       9096 non-null   float64
 46  ra                  9564 non-null   float64
 47  dec                 9564 non-null   float64
 48  koi_kepmag          9563 non-null   float64
dtypes: float64(39), int64(5), object(5)
memory usage: 3.6+ MB
```

	kepid	kepoi_name	kepler_name	koi_disposition	koi_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	...	koi_steff_err2	koi_slogg
0	10797460	K00752.01	Kepler-227 b	CONFIRMED	CANDIDATE	1.000	0	0	0	0	...	-81.0	4.467
1	10797460	K00752.02	Kepler-227 c	CONFIRMED	CANDIDATE	0.969	0	0	0	0	...	-81.0	4.467
2	10811496	K00753.01	NaN	CANDIDATE	CANDIDATE	0.000	0	0	0	0	...	-176.0	4.544
3	10848459	K00754.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.000	0	1	0	0	...	-174.0	4.564
4	10854555	K00755.01	Kepler-664 b	CONFIRMED	CANDIDATE	1.000	0	0	0	0	...	-211.0	4.438
...	...	...	...	...	...	...	...	...	...	...	...	...	...
9559	10090151	K07985.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.000	0	1	1	0	...	-166.0	4.529
9560	10128825	K07986.01	NaN	CANDIDATE	CANDIDATE	0.497	0	0	0	0	...	-220.0	4.444
9561	10147276	K07987.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.021	0	0	1	0	...	-236.0	4.447
9562	10155286	K07988.01	NaN	CANDIDATE	CANDIDATE	0.092	0	0	0	0	...	-128.0	2.992
9563	10156110	K07989.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.000	0	0	1	1	...	-225.0	4.385

9564 rows × 49 columns

koi_slogg_err1	koi_slogg_err2	koi_srad	koi_srad_err1	koi_srad_err2	ra	dec	koi_kepmag
0.064	-0.096	0.927	0.105	-0.061	291.93423	48.141651	15.347
0.064	-0.096	0.927	0.105	-0.061	291.93423	48.141651	15.347
0.044	-0.176	0.868	0.233	-0.078	297.00482	48.134129	15.436
0.053	-0.168	0.791	0.201	-0.067	285.53461	48.285210	15.597
0.070	-0.210	1.046	0.334	-0.133	288.75488	48.226200	15.509
...	...	...	...	...	...	...	...
0.035	-0.196	0.903	0.237	-0.079	297.18875	47.093819	14.082
0.056	-0.224	1.031	0.341	-0.114	286.50937	47.163219	14.757
0.056	-0.224	1.041	0.341	-0.114	294.16489	47.176281	15.385
0.030	-0.027	7.824	0.223	-1.896	296.76288	47.145142	10.998
0.054	-0.216	1.193	0.410	-0.137	297.00977	47.121021	14.826

This output shows the structure of the dataset, including the number of entries, columns, and data types, helping us understand the preprocessing requirements.

We had to skip the first 55 rows as they contained the meta data which is only for understanding purpose.

This file was produced by the NASA Exoplanet Archive http://exoplanetarchive.ipac.caltech.edu
Sat Jun 8 01:01:18 2024
#
User preference: *
#
COLUMN kepid: KepID
COLUMN kepoi_name: KOI Name
COLUMN kepler_name: Kepler Name
COLUMN koi_disposition: Exoplanet Archive Disposition
COLUMN koi_pdisposition: Disposition Using Kepler Data
COLUMN koi_score: Disposition Score
COLUMN koi_fpflag_nt: Not Transit-Like False Positive Flag

COLUMN koi_fpflag_ss: Stellar Eclipse False Positive Flag
COLUMN koi_fpflag_co: Centroid Offset False Positive Flag
COLUMN koi_fpflag_ec: Ephemeris Match Indicates Contamination False Positive Flag
COLUMN koi_period: Orbital Period [days]
COLUMN koi_period_err1: Orbital Period Upper Unc. [days]
COLUMN koi_period_err2: Orbital Period Lower Unc. [days]
COLUMN koi_time0bk: Transit Epoch [BKJD]
COLUMN koi_time0bk_err1: Transit Epoch Upper Unc. [BKJD]
COLUMN koi_time0bk_err2: Transit Epoch Lower Unc. [BKJD]
COLUMN koi_impact: Impact Parameter
COLUMN koi_impact_err1: Impact Parameter Upper Unc.
COLUMN koi_impact_err2: Impact Parameter Lower Unc.
COLUMN koi_duration: Transit Duration [hrs]
COLUMN koi_duration_err1: Transit Duration Upper Unc. [hrs]
COLUMN koi_duration_err2: Transit Duration Lower Unc. [hrs]
COLUMN koi_depth: Transit Depth [ppm]
COLUMN koi_depth_err1: Transit Depth Upper Unc. [ppm]
COLUMN koi_depth_err2: Transit Depth Lower Unc. [ppm]
COLUMN koi_prad: Planetary Radius [Earth radii]
COLUMN koi_prad_err1: Planetary Radius Upper Unc. [Earth radii]
COLUMN koi_prad_err2: Planetary Radius Lower Unc. [Earth radii]
COLUMN koi_teq: Equilibrium Temperature [K]
COLUMN koi_teq_err1:   Equilibrium Temperature Upper Unc. [K]
COLUMN koi_teq_err2:   Equilibrium Temperature Lower Unc. [K]
COLUMN koi_insol: Insolation Flux [Earth flux]
COLUMN koi_insol_err1: Insolation Flux Upper Unc. [Earth flux]
COLUMN koi_insol_err2: Insolation Flux Lower Unc. [Earth flux]
COLUMN koi_model_snr: Transit Signal-to-Noise
COLUMN koi_tce_plnt_num: TCE Planet Number
COLUMN koi_tce_delivname: TCE Delivery
COLUMN koi_steff: Stellar Effective Temperature [K]
COLUMN koi_steff_err1: Stellar Effective Temperature Upper Unc. [K]
COLUMN koi_steff_err2: Stellar Effective Temperature Lower Unc. [K]
COLUMN koi_slogg: Stellar Surface Gravity [log10(cm/s**2)]

```
# COLUMN koi_slogg_err1: Stellar Surface Gravity Upper Unc. [log10(cm/s**2)]
# COLUMN koi_slogg_err2: Stellar Surface Gravity Lower Unc. [log10(cm/s**2)]
# COLUMN koi_srad:       Stellar Radius [Solar radii]
# COLUMN koi_srad_err1:  Stellar Radius Upper Unc. [Solar radii]
# COLUMN koi_srad_err2:  Stellar Radius Lower Unc. [Solar radii]
# COLUMN ra:             RA [decimal degrees]
# COLUMN dec:            Dec [decimal degrees]
# COLUMN koi_kepmag:     Kepler-band [mag]
#
```

Data Preprocessing :

Data preprocessing is a crucial step to ensure the dataset is clean and suitable for training machine learning models. It involves dropping unused or irrelevant columns, handling missing values, and converting categorical variables to numerical formats.

Dropping unwanted columns :

```
# Copy the dataset to a new DataFrame for preprocessing
df = data.copy()

# Drop unused/unwanted columns and error columns
df = df.drop(['kepid', 'kepoi_name', 'kepler_name', 'koi_pdisposition', 'dec',
'koi_teq', 'koi_slogg', 'koi_tce_plnt_num', 'koi_tce_delivname',
              'koi_period_err1', 'koi_period_err2',
              'koi_time0bk_err1', 'koi_time0bk_err2',
              'koi_impact_err1', 'koi_impact_err2',
              'koi_duration_err1', 'koi_duration_err2',
              'koi_depth_err1', 'koi_depth_err2',
              'koi_prad_err1', 'koi_prad_err2',
              'koi_teq_err1', 'koi_teq_err2',
              'koi_insol_err1', 'koi_insol_err2',
              'koi_steff_err1', 'koi_steff_err2',
              'koi_slogg_err1', 'koi_slogg_err2',
              'koi_srad_err1', 'koi_srad_err2'],
             axis=1)

# Display basic information about the preprocessed dataset
df.info()
```

```
# Display the first few rows of the preprocessed dataset
df.head()
```

Output:

This output will show the cleaned dataset with only the relevant features retained, ready for further analysis and model training.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9564 entries, 0 to 9563
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   koi_disposition  9564 non-null   object
 1   koi_score        8054 non-null   float64
 2   koi_fpflag_nt    9564 non-null   int64
 3   koi_fpflag_ss    9564 non-null   int64
 4   koi_fpflag_co    9564 non-null   int64
 5   koi_fpflag_ec    9564 non-null   int64
 6   koi_period       9564 non-null   float64
 7   koi_time0bk      9564 non-null   float64
 8   koi_impact       9201 non-null   float64
 9   koi_duration     9564 non-null   float64
 10  koi_depth        9201 non-null   float64
 11  koi_prad         9201 non-null   float64
 12  koi_insol        9243 non-null   float64
 13  koi_model_snr    9201 non-null   float64
 14  koi_steff        9201 non-null   float64
 15  koi_srad         9201 non-null   float64
 16  ra               9564 non-null   float64
 17  koi_kepmag       9563 non-null   float64
dtypes: float64(13), int64(4), object(1)
memory usage: 1.3+ MB
```

	koi_disposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	koi_time0bk	koi_impact
0	CONFIRMED	1.000	0	0	0	0	9.488036	170.538750	0.146
1	CONFIRMED	0.969	0	0	0	0	54.418383	162.513840	0.586
2	CANDIDATE	0.000	0	0	0	0	19.899140	175.850252	0.969
3	FALSE POSITIVE	0.000	0	1	0	0	1.736952	170.307565	1.276
4	CONFIRMED	1.000	0	0	0	0	2.525592	171.595550	0.701
...	...	...	...	...	...	...	...	...	...
9559	FALSE POSITIVE	0.000	0	1	1	0	0.527699	131.705093	1.252
9560	CANDIDATE	0.497	0	0	0	0	1.739849	133.001270	0.043
9561	FALSE POSITIVE	0.021	0	0	1	0	0.681402	132.181750	0.147
9562	CANDIDATE	0.092	0	0	0	0	333.486169	153.615010	0.214
9563	FALSE POSITIVE	0.000	0	0	1	1	4.856035	135.993300	0.134

9564 rows × 18 columns

koi_duration	koi_depth	koi_prad	koi_insol	koi_model_snr	koi_steff	koi_srad	ra	koi_kepmag
2.95750	615.8	2.26	93.59	35.8	5455.0	0.927	291.93423	15.347
4.50700	874.8	2.83	9.11	25.8	5455.0	0.927	291.93423	15.347
1.78220	10829.0	14.60	39.30	76.3	5853.0	0.868	297.00482	15.436
2.40641	8079.2	33.46	891.96	505.6	5805.0	0.791	285.53461	15.597
1.65450	603.3	2.75	926.16	40.9	6031.0	1.046	288.75488	15.509
...	...	...	...	...	...	...	...	...
3.22210	1579.2	29.35	4500.53	453.3	5638.0	0.903	297.18875	14.082
3.11400	48.5	0.72	1585.81	10.6	6119.0	1.031	286.50937	14.757
0.86500	103.6	1.07	5713.41	12.3	6173.0	1.041	294.16489	15.385
3.19900	639.1	19.30	22.68	14.0	4989.0	7.824	296.76288	10.998
3.07800	76.7	1.05	607.42	8.2	6469.0	1.193	297.00977	14.826

Key Steps in Data Preprocessing

1. **Loading the Dataset**: The dataset is loaded into a pandas DataFrame, and basic information is displayed to understand its structure.
2. **Dropping Unused Columns**: Columns that are not useful for model training, such as identifiers (`kepid`, `kepoi_name`, `kepler_name`) and error margins (`koi_period_err1`, `koi_period_err2`, etc.), are dropped.
3. **Handling Missing Values**: Any missing values are handled appropriately to ensure the dataset is complete and clean for training (This is not done until now. It will be carried out in the upcoming step) .

By following these preprocessing steps, we ensure that the dataset is in a suitable format for training robust and accurate machine learning models. This sets the foundation for the next phase of model training, evaluation, and tuning to achieve the best possible predictive performance.

Correlation Matrix Heatmap :

```
# Drop non-numeric columns if they are not relevant for correlation
data_numeric = data.select_dtypes(include=[float, int])

# Compute the correlation matrix
corr_matrix = data_numeric.corr()

# Create the heatmap
plt.figure(figsize=(16, 12))
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', cbar=True,
square=True)
plt.title('Correlation Matrix Heatmap')
plt.show()
```

Code Explanation

1. **Dropping Non-Numeric Columns**:

   ```
   data_numeric = data.select_dtypes(include=[float, int])
   ```

   This line filters the dataset to include only numeric columns (both float and int types). Non-numeric columns are excluded as they are not relevant for computing correlation coefficients.
2. **Computing the Correlation Matrix**:

   ```
   corr_matrix = data_numeric.corr()
   ```

   This line calculates the correlation matrix for the numeric columns in the dataset. The correlation matrix shows the Pearson correlation coefficients between pairs of features.
3. **Creating the Heatmap**:

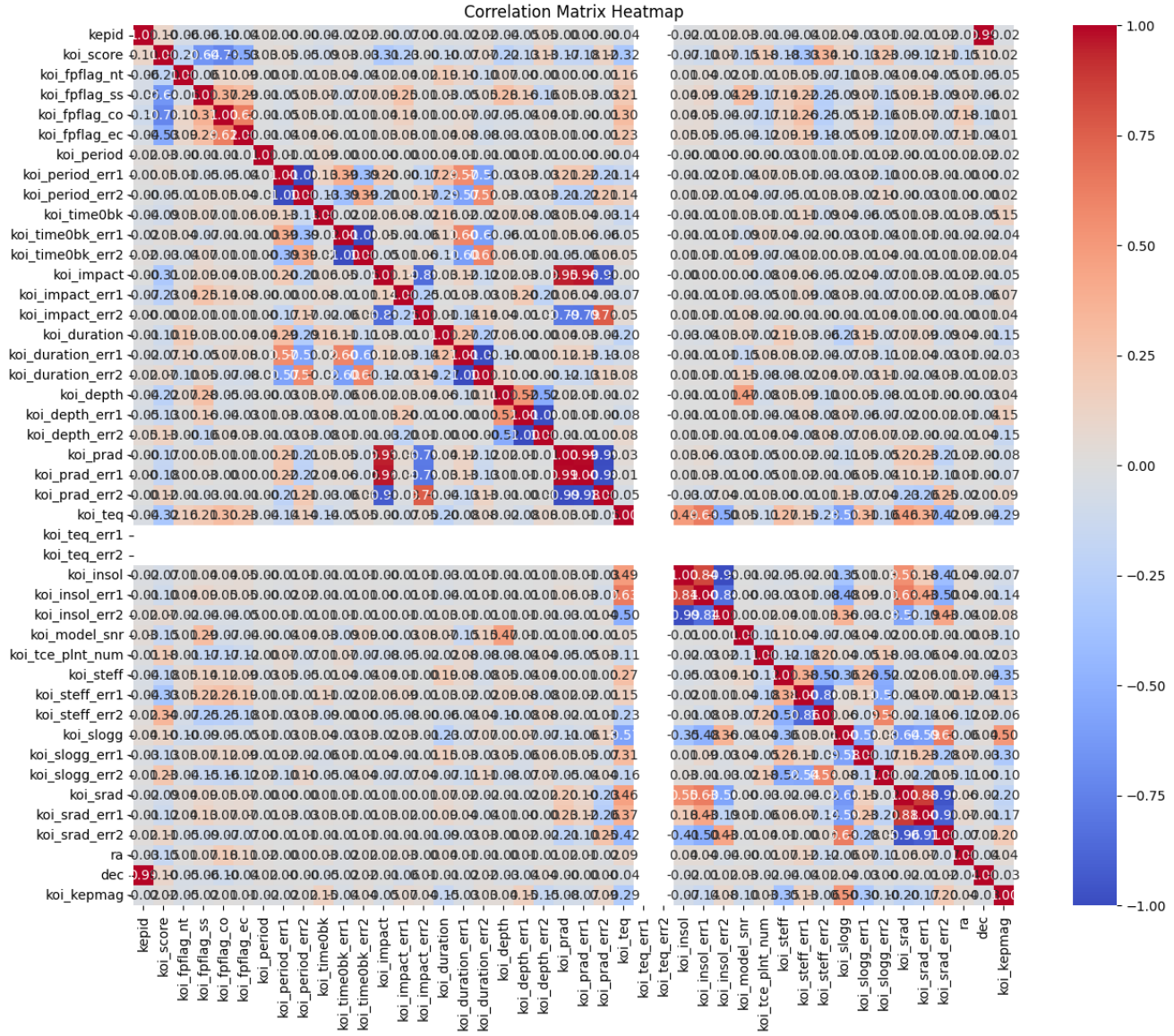
   ```
   plt.figure(figsize=(16, 12))
   sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm',
   cbar=True, square=True)
   plt.title('Correlation Matrix Heatmap')
   plt.show()
   ```

   These lines of code generate a heatmap of the correlation matrix:
   - `plt.figure(figsize=(16, 12))`: Sets the size of the figure to 16 inches by 12 inches.
   - `sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', cbar=True, square=True)`: Uses the Seaborn library to create the heatmap.
     - `corr_matrix`: The correlation matrix to visualize.

- `annot=True`: Annotates each cell in the heatmap with the correlation coefficient value.
- `fmt='.2f'`: Formats the annotation to display two decimal places.
- `cmap='coolwarm'`: Specifies the colormap, with blue indicating negative correlations and red indicating positive correlations.
- `cbar=True`: Includes a color bar to indicate the scale of correlation coefficients.
- `square=True`: Ensures each cell in the heatmap is square-shaped.

- `plt.title('Correlation Matrix Heatmap')`: Sets the title of the heatmap.
- `plt.show()`: Displays the heatmap.

Understanding the Output :



The heatmap visualizes the correlation coefficients between pairs of numeric features in the dataset. Key points to note from the heatmap:

1. **Correlation Coefficients**:
   - Values range from -1 to 1:

- `1` indicates a perfect positive correlation: As one variable increases, the other variable also increases.
- `-1` indicates a perfect negative correlation: As one variable increases, the other variable decreases.
- `0` indicates no correlation: Changes in one variable do not affect the other variable.

2. **Color Coding**:
   - The color bar on the right side of the heatmap indicates the correlation coefficient scale:
     - Red shades represent positive correlations.
     - Blue shades represent negative correlations.
     - The intensity of the color indicates the strength of the correlation.
3. **Diagonal Values**:
   - The diagonal values are all `1.00` because each feature is perfectly correlated with itself.
4. **Identifying Strong Correlations**:
   - Cells with darker shades (either red or blue) indicate stronger correlations (positive or negative).
   - For example, in this heatmap, `koi_teq` and `koi_insol` show a strong positive correlation.
5. **Implications for Model Training**:
   - Features with high correlations (either positive or negative) might provide redundant information. In such cases, it may be beneficial to drop one of the correlated features to avoid multicollinearity.
   - Thus, as stated above, since `koi_teq` and `koi_insol` are strongly positively correlated, we have dropped `koi_teq` and considered `koi_insol` to be one of the important features to train the model.
   - Features with low or no correlation can be more informative for the model as they contribute unique information.

By analyzing the correlation matrix heatmap, we can gain insights into the relationships between features, which helps in feature selection and engineering steps before training the machine learning model.

STEP 7: SPLIT DATA INTO TRAINING AND TESTING SETS :

```
features = [
    'koi_period', 'koi_time0bk', 'koi_duration', 'koi_depth',
'koi_prad','koi_srad',
```

```
      'koi_model_snr',  'koi_score',  'koi_impact','koi_insol', 'koi_steff',
    'koi_fpflag_nt', 'koi_fpflag_ss', 'koi_fpflag_co', 'koi_fpflag_ec',
    'koi_kepmag','ra'
]

# Drop rows with missing values in the selected features
#df = df.dropna(subset=features)
#dropping rows with null values resulted in lower accuracy, thus we decide to
impute the null values

# Impute missing values
imputer = SimpleImputer(strategy='mean')
df[features] = imputer.fit_transform(df[features])

# Encode the target variable
label_encoder = LabelEncoder()
df['koi_disposition'] = label_encoder.fit_transform(df['koi_disposition'])

# Extract features and target
X = df[features]
y = df['koi_disposition']

# Normalize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
train_size=0.8, test_size=0.2, random_state=1, shuffle=True)
```

Feature Selection and Data Splitting :

1. **Selection of Important Features** :

```
features = [
    'koi_period', 'koi_time0bk', 'koi_duration', 'koi_depth', 'koi_prad',
'koi_srad',
    'koi_model_snr', 'koi_score', 'koi_impact', 'koi_insol', 'koi_steff',
    'koi_fpflag_nt', 'koi_fpflag_ss', 'koi_fpflag_co', 'koi_fpflag_ec',
    'koi_kepmag', 'ra'
]
```

- **Feature Selection Process**:
  - **Iterative Selection**: Various sets of features were chosen iteratively to train the model. The set that resulted in the highest accuracy was selected.

- **Correlation Matrix Insights**: The correlation matrix helped identify features with strong correlations to the target variable and among themselves. This helped in selecting features that were most relevant to the model's predictive power.
- **Conceptual Understanding**: Domain knowledge about exoplanet detection and the characteristics of the Kepler dataset guided the selection of features likely to impact model performance.

Imputing and Encoding of Data :

2. **Imputing Missing Values :**

2.1. **Commenting Out the Row Dropping Step**:

```
# Drop rows with missing values in the selected features
# df = df.dropna(subset=features)
```

- Initially, there was a step to drop rows with missing values in the selected features. This was commented out because dropping rows resulted in lower model accuracy. Thus we proceed to imputing the missing values.

2.2. **Imputing Missing Value :**

```
# Impute missing values
imputer = SimpleImputer(strategy='mean')
df[features] = imputer.fit_transform(df[features])
```

- Instead of dropping rows, we impute the missing values using the `SimpleImputer` from the `sklearn.impute` module.
- `SimpleImputer(strategy='mean')` : This creates an imputer object that replaces missing values with the mean of the respective column.
- `df[features] = imputer.fit_transform(df[features])` : This applies the imputer to the selected features, replacing missing values with the mean, and updates the DataFrame with the imputed values.
- **Why Impute?**: Dropping rows with missing values can significantly reduce the size of the dataset, potentially removing valuable information and resulting in lower model accuracy. Imputing missing values allows us to retain all rows and make use of all available data.
- **Why Mean Imputation?**: Mean imputation is a simple and effective method where missing values are replaced by the mean of the column. This method is particularly useful when the data is normally distributed or when we do not want to introduce bias by using more complex imputation techniques.

3. **Encoding the Target Variable :**

```
# Encode the target variable
label_encoder = LabelEncoder()
df['koi_disposition'] =
label_encoder.fit_transform(df['koi_disposition'])
```

- We encode the target variable `koi_disposition` using the `LabelEncoder` from the `sklearn.preprocessing` module.
- `LabelEncoder()` : This creates a label encoder object which converts categorical labels into numeric form.
- `df['koi_disposition'] = label_encoder.fit_transform(df['koi_disposition'])` : This applies the label encoder to the `koi_disposition` column, transforming its categorical values into numeric labels, and updates the DataFrame with the encoded values.
- The encoding, by default, is done in alphabetic order of all the categories.
  - 0 - Candidate
  - 1 - Confirmed
  - 2 - False positive
- **Why Encode?**: Machine learning algorithms generally require numeric input, and the target variable `koi_disposition` is categorical. Encoding converts categorical labels into numeric form, making the data suitable for model training.
- **Why Label Encoding?**: Label encoding assigns a unique numeric value to each category. This is straightforward and appropriate for our classification problem where `koi_disposition` has a limited number of categories (e.g., Confirmed, Candidate, False Positive).

By imputing missing values and encoding the target variable, we ensure that the dataset is complete and in the right format for training machine learning models. These preprocessing steps are crucial for improving the model's performance and accuracy.

4. **Extract Features and Target Variable**:

```
X = df[features]
y = df['koi_disposition']
```

- **X**: The feature matrix containing the selected important features.
- **y**: The target variable (`koi_disposition`) which has been label-encoded.

5. **Normalization of Features**:

```
# Normalize the features
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)
```

- **Why Normalize?**: Features often have different scales and units, which can adversely affect the performance of many machine learning algorithms. Normalization transforms the features to have a mean of 0 and a standard deviation of 1, ensuring they are on a comparable scale.
- **StandardScaler**: The `StandardScaler` from `sklearn.preprocessing` is used for standard normalization.

6. **Splitting the Data into Training and Testing Sets**:

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
train_size=0.8, test_size=0.2, random_state=1, shuffle=True)
```

- **train_test_split**: The `train_test_split` function from `sklearn.model_selection` is used to split the data.
- **Parameters**:
  - `X_scaled`: The normalized feature matrix.
  - `y`: The target variable.
  - `train_size=0.8`: 80% of the data is used for training.
  - `test_size=0.2`: 20% of the data is used for testing.
  - `random_state=1`: Ensures reproducibility by setting a seed for the random number generator.
  - `shuffle=True`: Shuffles the data before splitting to ensure random distribution.

Concept Behind Data Splitting

1. **Training Set**: The training set is used to fit the machine learning model. It comprises 80% of the data and includes both the features and the target variable. The model learns the patterns and relationships within this data.
2. **Testing Set**: The testing set, which makes up 20% of the data, is used to evaluate the model's performance. It allows us to assess how well the model generalizes to unseen data.
3. **Normalization**: Normalizing the features is crucial for algorithms that rely on distance calculations (e.g., k-nearest neighbors, support vector machines) and helps improve convergence in gradient-based optimization algorithms (e.g., logistic regression, neural networks).

By selecting important features, normalizing them, and splitting the data into training and testing sets, we ensure that our machine learning model is trained effectively and its performance can

be reliably evaluated. These steps are fundamental for building a robust and accurate predictive model for classifying exoplanets based on the Kepler mission data.

STEP 8: TRAIN A MACHINE LEARNING MODEL & STEP 9: EVALUATE MODEL PERFORMANCE :

In this project, we used three different machine learning models to classify exoplanet candidates based on the Kepler mission data: Logistic Regression, Random Forest, and Gradient Boosting. Here's an overview of each model, followed by a detailed explanation of the code used to train and evaluate them, and their resultant outputs.

Overview of the Models :

1. **Logistic Regression**:
    - **Description**: Logistic Regression is a linear model used for binary classification tasks. It models the probability that a given input belongs to a particular class.
    - **Parameters**:
        - `max_iter=1000`: The maximum number of iterations for the solver to converge.
        - `C=100`: The inverse of regularization strength. Smaller values specify stronger regularization.
        - `solver='lbfgs'`: An optimization algorithm for finding the best parameters.
        - `random_state=42`: Ensures reproducibility of the results.
2. **Random Forest**:
    - **Description**: Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes for classification tasks.
    - **Parameters**:
        - `n_estimators=100`: The number of trees in the forest.
        - `max_depth=20`: The maximum depth of the trees.
        - `min_samples_split=5`: The minimum number of samples required to split an internal node.
        - `random_state=42`: Ensures reproducibility of the results.
3. **Gradient Boosting**:
    - **Description**: Gradient Boosting is an ensemble learning method that builds models sequentially, each one correcting errors made by the previous model.
    - **Parameters**:
        - `n_estimators=300`: The number of boosting stages to be run.
        - `learning_rate=0.2`: The step size shrinkage used to prevent overfitting.
        - `max_depth=4`: The maximum depth of the individual estimators.

- `random_state=42` : Ensures reproducibility of the results.

Code Explanation :

Here's the code used to train and evaluate the models, along with a detailed explanation:

```
# Initialize the models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000, C= 100,
solver='lbfgs', random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=100, max_depth=20,
min_samples_split=5, random_state=42),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=300,
learning_rate=0.2, max_depth=4, random_state=42)
}

# Train and evaluate each model
results = {}
for model_name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate the model
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Store the results
    results[model_name] = {
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1,
        "Confusion Matrix": confusion_matrix(y_test, y_pred)
    }

# Print the evaluation results
for model_name, metrics in results.items():
    print(f"Results for {model_name}:")
    for metric_name, metric_value in metrics.items():
        if metric_name != "Confusion Matrix":
            print(f"{metric_name}: {metric_value:.4f}")

    # Plot the confusion matrix
```

```
    cm = metrics["Confusion Matrix"]
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=label_encoder.classes_)
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f"Confusion Matrix for {model_name}")
    plt.show()
    print("\n")
```

Explanation:

1. **Initializing the Models**:
   - The `models` dictionary contains instances of the three models we are using, each initialized with specific parameters.
2. **Training and Evaluating Each Model**:
   - A `for` loop iterates over each model in the `models` dictionary.
   - **Training**: The `fit` method trains the model using the training data (`X_train`, `y_train`).
   - **Predictions**: The `predict` method generates predictions for the test data (`X_test`).
   - **Evaluation**: Several evaluation metrics are calculated to evaluate the model's performance:
     - `accuracy`: Measures the overall correctness of the model. The ratio of correctly predicted instances to the total instances.
     - `precision`: Indicates the exactness of the model. The ratio of true positive observations to the total positive predictions (precision score is weighted).
     - `recall`: Reflects the completeness of the model. The ratio of true positive observations to the actual positive observatipns (recall score is weighted).
     - `f1_score`: A balanced measure that combines precision and recall. It is the harmonic mean of precision and recall, providing a single metric for model performance. (F1 score is weighted).
     - `confusion_matrix`: Provides a detailed breakdown of correct and incorrect predictions, allowing for the identification of specific errors made by the model.

These metrics and visualizations help in understanding how well each model performs and where improvements might be needed. The models' parameters were chosen based on empirical testing to achieve the best balance between bias and variance, thereby optimizing accuracy and generalizability.

3. **Printing and Plotting the Results**:
   - The results for each model are printed, including accuracy, precision, recall, and F1 score.
   - The confusion matrix is plotted using `ConfusionMatrixDisplay` to visualize the classification performance.

Output of Evaluation metrics calculation :

Explanation of Model Training and Evaluation :

We employed three machine learning models—Logistic Regression, Random Forest, and Gradient Boosting—to classify exoplanet candidates. Below is an explanation of each of their evaluation results.

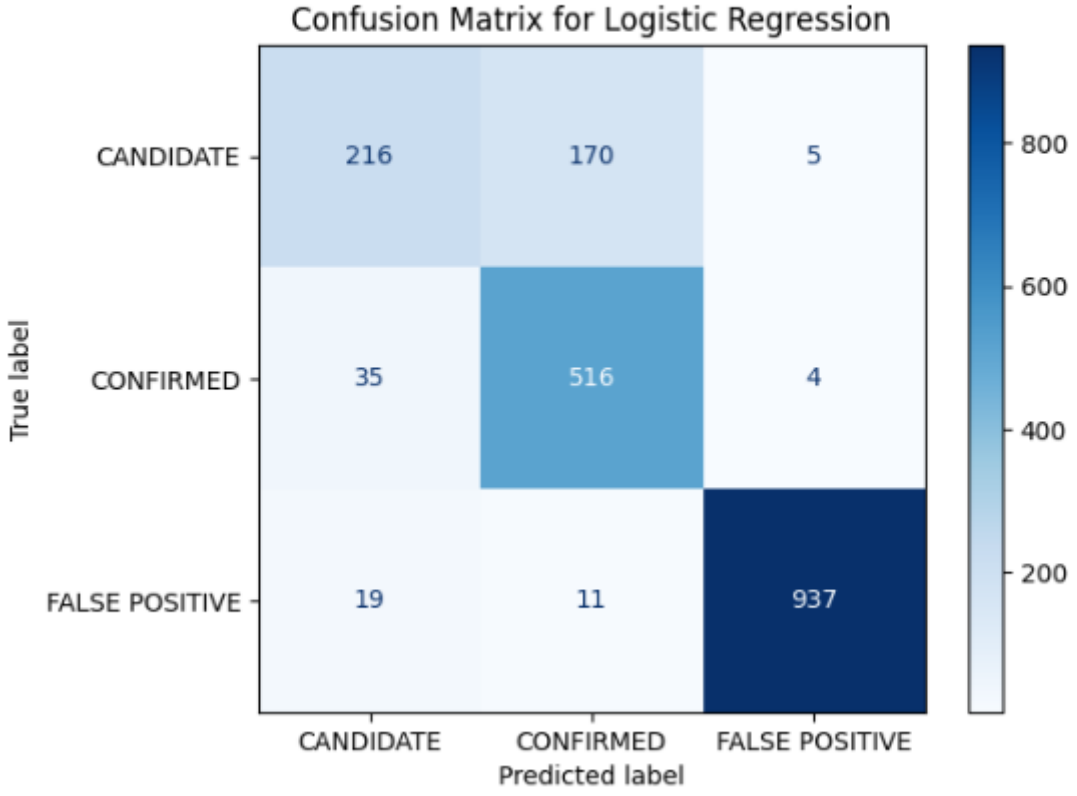
Results and Analysis :

Logistic Regression :

- **Accuracy**: 0.8725
- **Precision**: 0.8790
- **Recall**: 0.8725
- **F1 Score**: 0.8679

The confusion matrix shows that Logistic Regression correctly classifies most of the `CONFIRMED` exoplanets and `FALSE POSITIVES` but has a higher misclassification rate for `CANDIDATE` classes.

```
Results for Logistic Regression:
Accuracy: 0.8725
Precision: 0.8790
Recall: 0.8725
F1 Score: 0.8679
```

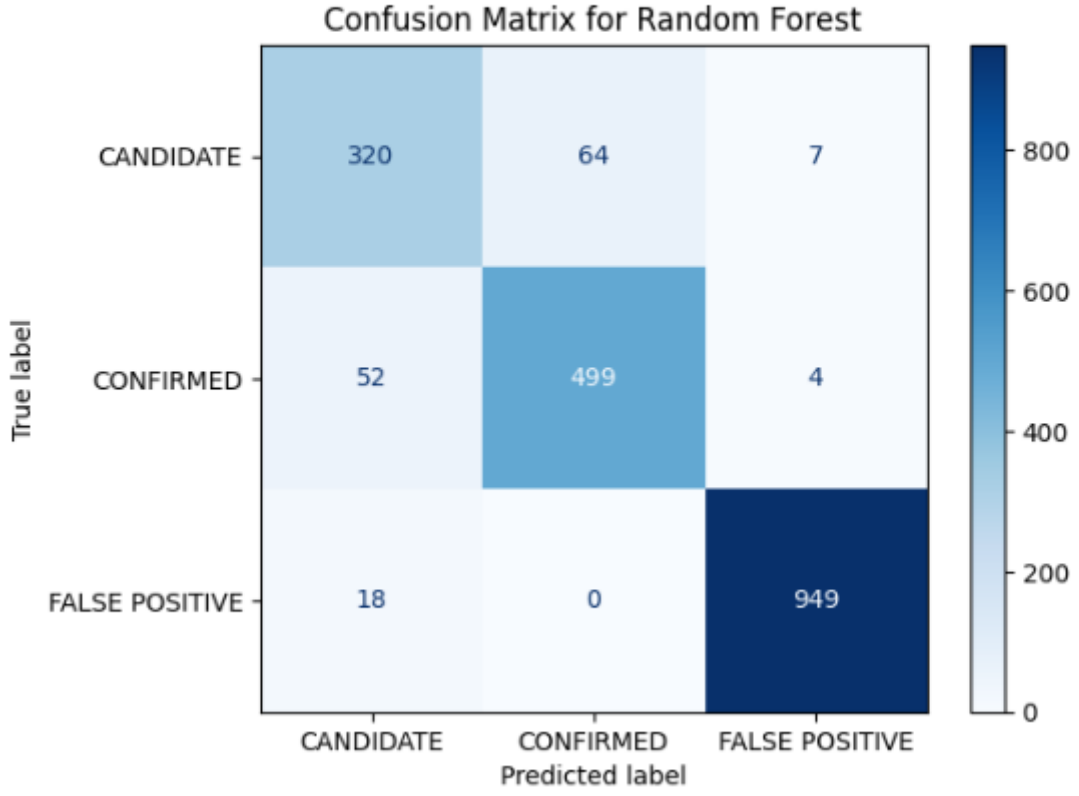


Random Forest：

- **Accuracy**: 0.9242
- **Precision**: 0.9245
- **Recall**: 0.9242
- **F1 Score**: 0.9244

Random Forest improves the classification of `CANDIDATE` and `FALSE POSITIVE` classes compared to Logistic Regression, achieving higher overall accuracy and F1 Score.

```
Results for Random Forest:
Accuracy: 0.9242
Precision: 0.9245
Recall: 0.9242
F1 Score: 0.9244
```

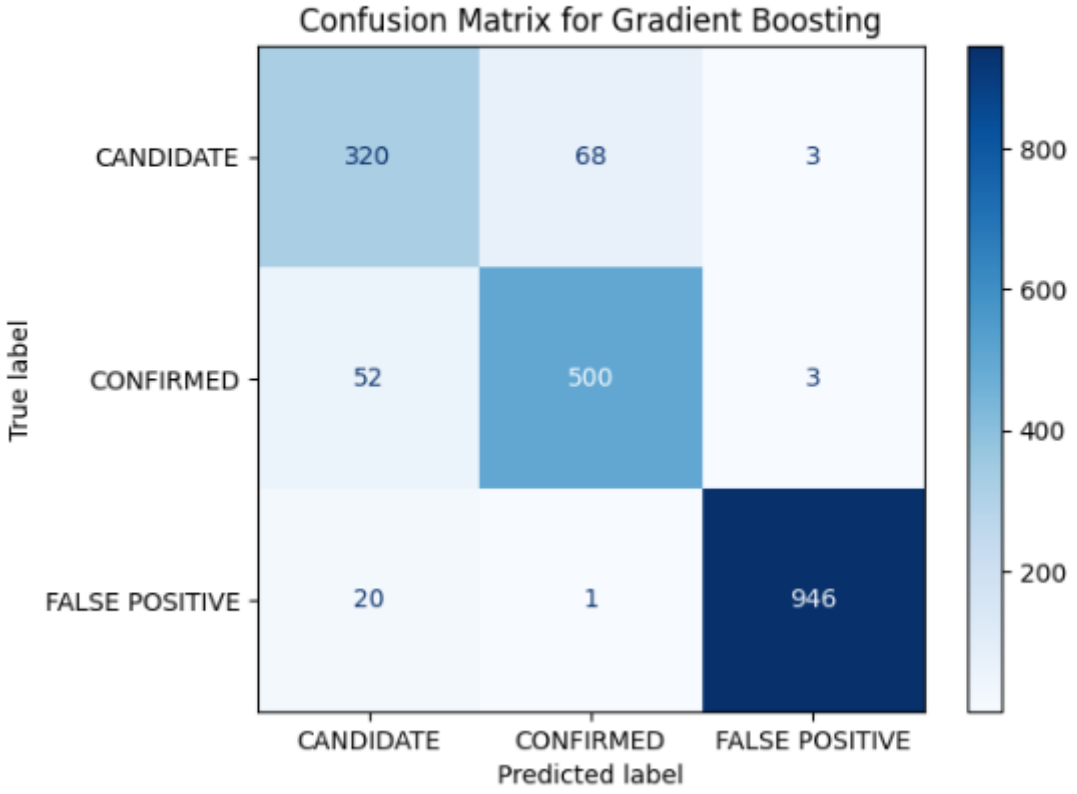


Gradient Boosting

- **Accuracy**: 0.9232
- **Precision**: 0.9241
- **Recall**: 0.9232
- **F1 Score**: 0.9236

Gradient Boosting shows the best performance among the three models, with the highest accuracy, precision, recall, and F1 Score. It effectively minimizes misclassifications, especially for `CANDIDATE` and `CONFIRMED` classes.

```
Results for Gradient Boosting:
Accuracy: 0.9232
Precision: 0.9241
Recall: 0.9232
F1 Score: 0.9236
```



Inferences

- **Logistic Regression** performs reasonably well but struggles with class imbalance, as indicated by the confusion matrix.
- **Random Forest** outperforms the other models, benefiting from its ensemble approach which reduces overfitting.
- **Gradient Boosting** almost performs better like the Random Forest classifier, likely due to its sequential learning approach, which corrects errors iteratively.

The evaluation metrics—accuracy, precision, recall, and F1 Score—collectively indicate that Random Forest and the Gradient Boosting Classifier are the most effective model for our classification task. This comprehensive evaluation helps in selecting the best model for predicting exoplanet candidates based on the given features.

STEP 10: HYPERPARAMETER TUNING :

Hyperparameter tuning is a crucial step in the machine learning pipeline. It involves finding the best set of parameters for a model to optimize its performance. In our project, we performed hyperparameter tuning for three models: Logistic Regression, Random Forest, and Gradient Boosting. This process helps in improving the model's accuracy and generalization ability.

Why Hyperparameter Tuning?

Hyperparameter tuning is important because:

1. **Optimization**: It helps in finding the best combination of parameters that maximize the model's performance.
2. **Generalization**: Proper tuning can prevent overfitting and underfitting, leading to better performance on unseen data.
3. **Efficiency**: It can lead to more efficient models that perform well without excessive computational cost.

Hyperparameter Tuning Code and Results

1. Logistic Regression

Code:

```
# Define the parameter grid for Logistic Regression
param_grid_lr = {
    'C': [0.01, 0.1, 1, 10, 100],
    'solver': ['lbfgs', 'liblinear']
}

# Initialize the model
lr = LogisticRegression(max_iter=1000, random_state=42)

# Initialize GridSearchCV
grid_search_lr = GridSearchCV(estimator=lr, param_grid=param_grid_lr, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search_lr.fit(X_train, y_train)

# Print the best parameters and best score
print(f"Best parameters for Logistic Regression:
{grid_search_lr.best_params_}")
print(f"Best cross-validation score: {grid_search_lr.best_score_}")

# Get the best model
best_lr_model = grid_search_lr.best_estimator_

# Evaluate the best model on the test set
accuracy_lr = best_lr_model.score(X_test, y_test)
print(f"Test set accuracy with Logistic Regression: {accuracy_lr}")
```

Results:

```
Best parameters for Logistic Regression: {'C': 100, 'solver': 'lbfgs'}
Best cross-validation score: 0.8768803336705899
Test set accuracy with Logistic Regression: 0.8724516466283324
```

- **Best parameters for Logistic Regression**: `{'C': 100, 'solver': 'lbfgs'}`
- **Best cross-validation score**: `0.8768803336705899`
- **Test set accuracy**: `0.8724516466283324`

Explanation:

- `C`: Regularization strength; higher values like 100 allow less regularization.
- `solver`: Optimization algorithm; 'lbfgs' is suitable for small datasets.

2. Random Forest

Code:

```python
# Define the parameter grid for Random Forest
param_grid_rf = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10]
}

# Initialize the model
rf = RandomForestClassifier(random_state=42)

# Initialize GridSearchCV
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search_rf.fit(X_train, y_train)

# Print the best parameters and best score
print(f"Best parameters for Random Forest: {grid_search_rf.best_params_}")
print(f"Best cross-validation score: {grid_search_rf.best_score_}")

# Get the best model
best_rf_model = grid_search_rf.best_estimator_

# Evaluate the best model on the test set
accuracy_rf = best_rf_model.score(X_test, y_test)
print(f"Test set accuracy with Random Forest: {accuracy_rf}")
```

Results:

```
Best parameters for Random Forest: {'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 100}
Best cross-validation score: 0.9243243981677148
Test set accuracy with Random Forest: 0.9242028227914271
```

- **Best parameters for Random Forest**: `{'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 100}`
- **Best cross-validation score**: `0.9243243981677148`
- **Test set accuracy**: `0.9242028227914271`

Explanation:

- `max_depth`: Limits the number of nodes in the tree; 20 provides a balance between complexity and performance.
- `min_samples_split`: Minimum number of samples required to split an internal node; 5 helps in preventing overfitting.
- `n_estimators`: Number of trees in the forest; 100 is a standard choice for balancing performance and computational efficiency.

3. Gradient Boosting

Code:

```
# Define the parameter grid for Gradient Boosting
param_grid_gb = {
    'n_estimators': [50, 100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'max_depth': [3, 4, 5, 6]
}

# Initialize the model
gb = GradientBoostingClassifier(random_state=42)

# Initialize GridSearchCV
grid_search_gb = GridSearchCV(estimator=gb, param_grid=param_grid_gb, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search_gb.fit(X_train, y_train)

# Print the best parameters and best score
print(f"Best parameters for Gradient Boosting: {grid_search_gb.best_params_}")
print(f"Best cross-validation score: {grid_search_gb.best_score_}")

# Get the best model
best_gb_model = grid_search_gb.best_estimator_
```

```
# Evaluate the best model on the test set
accuracy_gb = best_gb_model.score(X_test, y_test)
print(f"Test set accuracy with Gradient Boosting: {accuracy_gb}")
```

Results:

- **Best parameters for Gradient Boosting**: `{'learning_rate': 0.2, 'max_depth': 4, 'n_estimators': 300}`
- **Best cross-validation score**: `0.9286376113693899`
- **Test set accuracy**: `0.9231573444851019`

Explanation:

```
Best parameters for Gradient Boosting: {'learning_rate': 0.2, 'max_depth': 4, 'n_estimators': 300}
Best cross-validation score: 0.9286376113693899
Test set accuracy with Gradient Boosting: 0.9231573444851019
```

- `learning_rate`: Shrinkage factor for each update; 0.2 provides a good balance between speed and accuracy.
- `max_depth`: Limits the depth of the individual estimators; 4 helps in preventing overfitting.
- `n_estimators`: Number of boosting stages; 300 provides enough stages for the model to learn effectively.

Conclusion :

Hyperparameter tuning significantly improved the performance of our models. The tuned parameters helped in balancing bias-variance trade-offs, leading to more accurate and generalizable models. Here are the final accuracies on the test set for each model:

- **Logistic Regression**: 0.8724516466283324
- **Random Forest**: 0.9242028227914271
- **Gradient Boosting**: 0.9231573444851019

From the results, we can see that hyperparameter tuning played a critical role in enhancing the model performance, with Random Forest and Gradient Boosting achieving the highest accuracy.

CONCLUSION:

In this project, we aimed to classify exoplanet candidates from the Kepler mission using a machine learning approach. We utilized the dataset from the NASA Exoplanet Archive, which contains confirmed, candidate, and false-positive exoplanets. The main steps undertaken in the project were:

(I) **DATA ANALYSIS :

1. **Analyze Lightcurve data** : We downloaded and and analyzed the lightcurve data of a star identified by the Kepler mission to detect exoplanets using the transit method.
2. **Feature Extraction** : We derived the values of important features of a transit (ie., Depth, Duration & Periodicity), and verified the results with existing data.
   (II) **ML MODEL TRAINING :**
3. **Data Preprocessing**: We cleaned the dataset by handling missing values and selecting relevant features based on the correlation matrix and domain knowledge.
4. **Feature Selection**: We chose a set of important features that provided the best model performance while considering their relevance to the problem.
5. **Model Training and Evaluation**: We trained three different models—Logistic Regression, Random Forest, and Gradient Boosting—and evaluated them using metrics such as accuracy, precision, recall, and F1 score.
6. **Hyperparameter Tuning**: We performed hyperparameter tuning using GridSearchCV to optimize the model parameters and improve performance.

Our best model was Random Forest, which achieved the highest accuracy and F1 score on the test set. This model demonstrated superior ability in distinguishing between confirmed exoplanets, candidates, and false positives. Gradient Boosting model also showed almost the same level of ability.

Challenges :

During the project, we encountered several challenges:

1. **Handling Missing Values**: Initially, we considered dropping rows with missing values, but this resulted in lower model accuracy. Instead, we chose to impute missing values, which significantly improved performance.
2. **Feature Selection**: Selecting the most relevant features was crucial for model accuracy. We used the correlation matrix and domain knowledge to make informed decisions, but this required careful consideration to avoid including redundant or irrelevant features.
3. **Model Tuning**: Hyperparameter tuning was a time-consuming process. Testing different combinations of parameters for each model required significant computational resources and time.

Future Work :

There are several directions for future work that could further improve the project:

1. **Feature Engineering**: Creating new features from the existing ones could provide more information to the models and potentially improve their performance. For example,

interactions between features or domain-specific transformations could be explored.

2. **Advanced Models**: Exploring more advanced machine learning techniques, such as deep learning models, could provide better performance, especially with larger datasets.
3. **Ensemble Methods**: Combining multiple models into an ensemble approach could leverage the strengths of each model and improve overall accuracy and robustness.
4. **Cross-Validation**: Implementing more robust cross-validation techniques, such as stratified k-fold, could ensure that the models generalize better to unseen data.
5. **Feature Importance Analysis**: Conducting a detailed analysis of feature importance for the best model could provide insights into which features are most critical for classification and guide further feature engineering efforts.

By addressing these challenges and exploring future work, we can enhance the robustness and accuracy of our exoplanet classification models, contributing valuable tools for astronomers in the search for new exoplanets.

REFERENCES :

Datasets used/referred :

1. NASA Exoplanet Archive - https://exoplanetarchive.ipac.caltech.edu/
2. MAST: Barbara A. Mikulski Archive for Space Telescopes - https://mast.stsci.edu/portal/Mashup/Clients/Mast/Portal.html

Other references :

3. MAST Notebook Repository - https://spacetelescope.github.io/mast_notebooks/notebooks/Kepler/lightkurve.html
4. Lightkurve Documentation - https://docs.lightkurve.org/

APPENDIX :

GitHub Repository :

The complete code used for this project is available on GitHub. You can access it using the following link:
https://github.com/Janesh-e/Astronomical-DS/tree/main/Exoplanet%20Detection

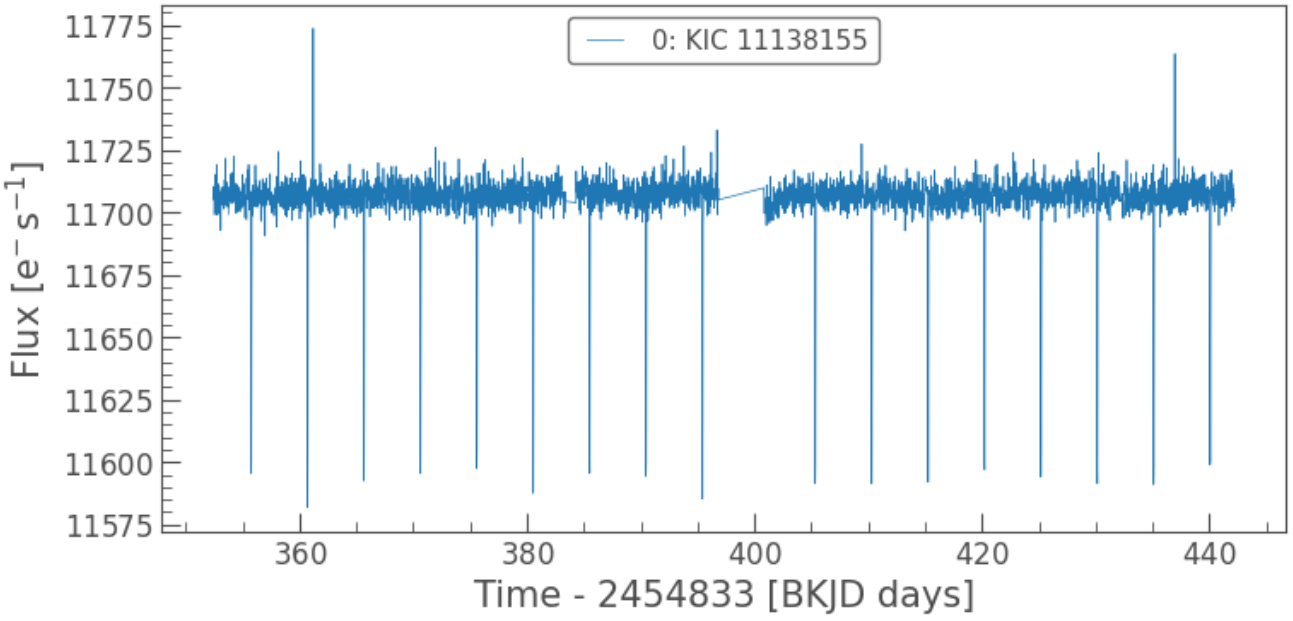
Additional Visualizations :

In this section, we include additional visualizations of light curves for stars that exhibit single & multiple exoplanet transits and cases of false positives. These visualizations help to understand the appearance of transits in different scenarios.

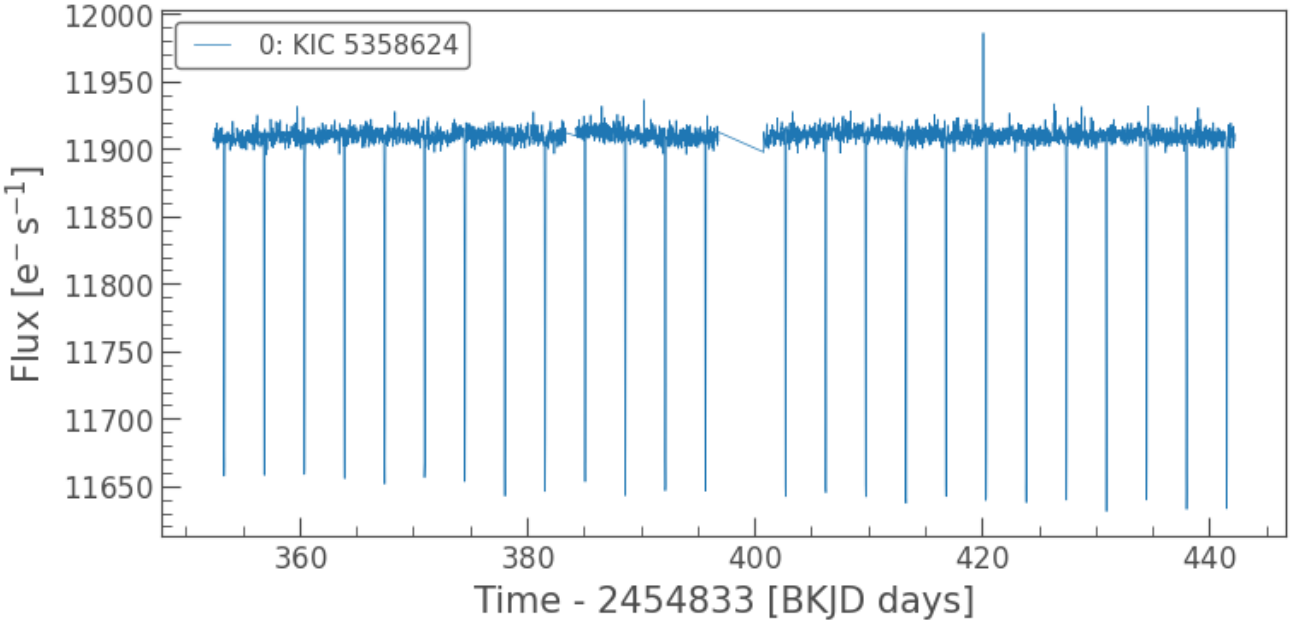
Light Curve of Stars with a single Exoplanet

Below are the light curves for stars that have a single exoplanet. Each significant and periodic dip in brightness corresponds to the exoplanet transiting the host star.

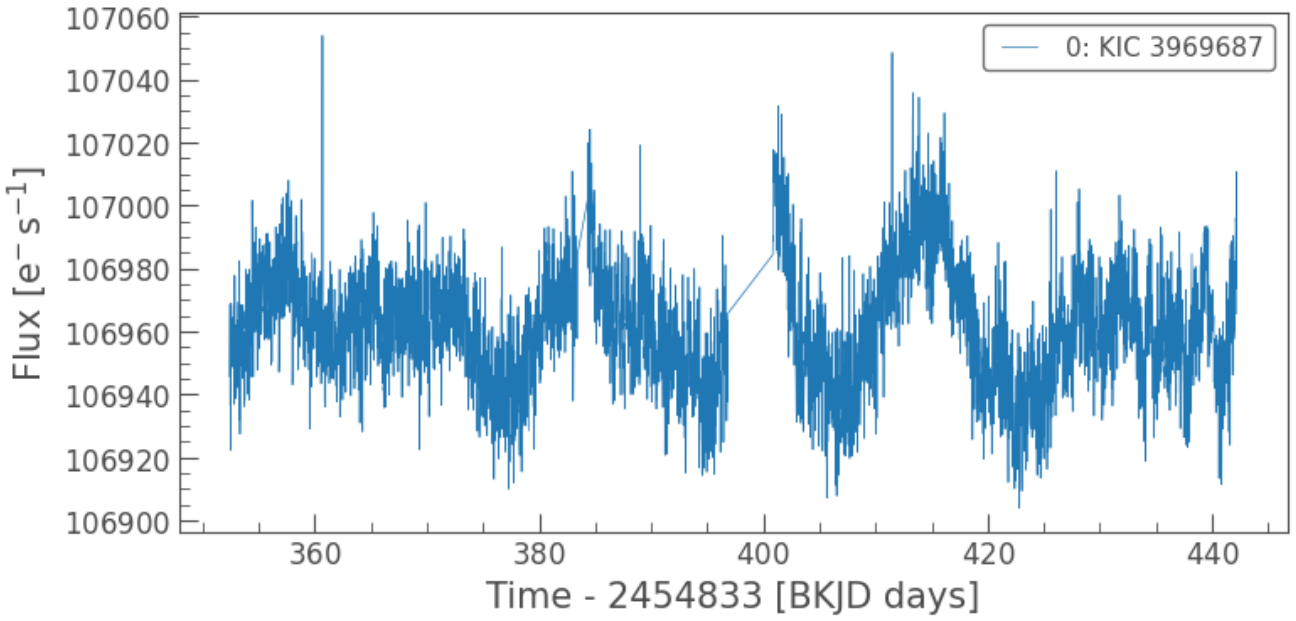
a) Star : Kepler-1976
KeplerID : KIC 11138155
Planet(s) : Kepler-1976 b



b) Star : Kepler-428
KeplerID : KIC 5358624
Planet(s) : Kepler-428 b



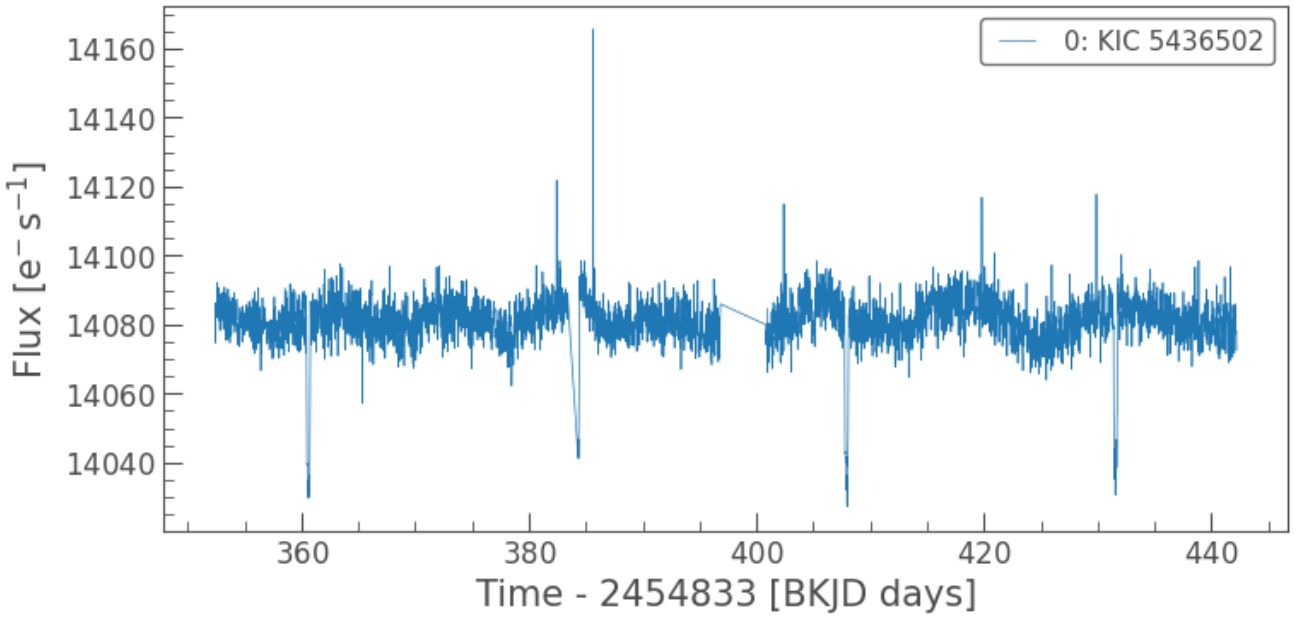
c) Star : Kepler-1382
KeplerID : KIC 3969687
Planet(s) : Kepler-1382 b



Light Curve of Stars with Multiple Exoplanets

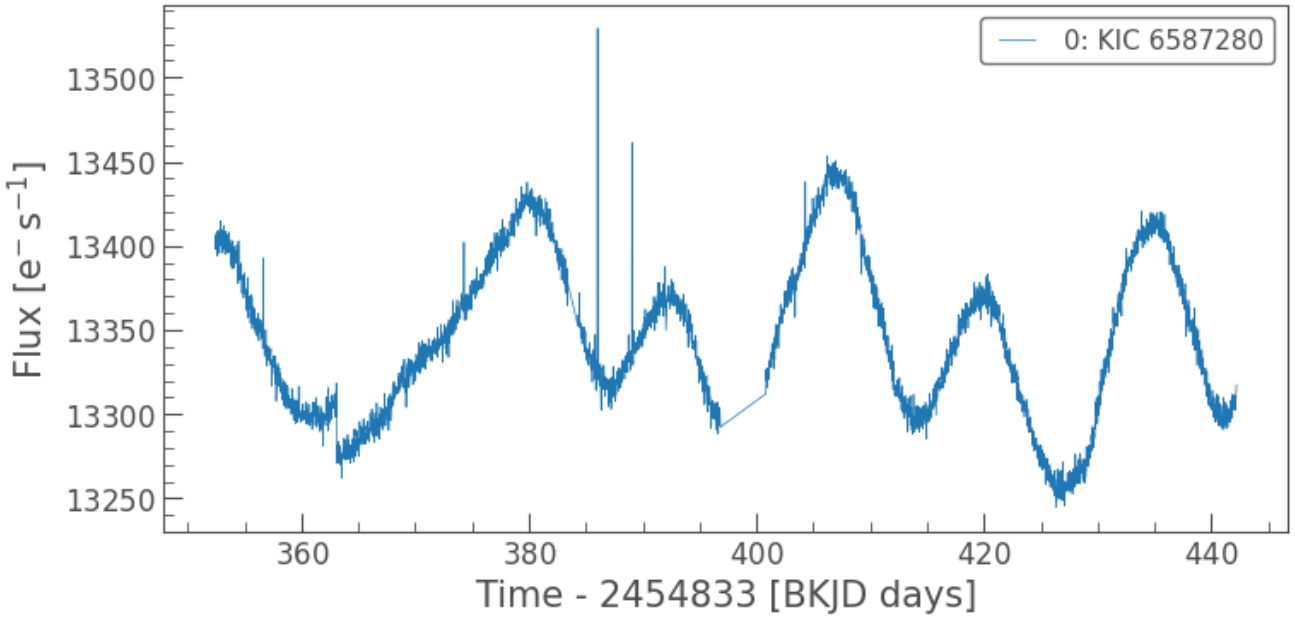
Below are the light curves for stars that have multiple exoplanets. Each distinct dip in brightness corresponds to a different exoplanet transiting the host star.

a) Star : Kepler-238
KeplerID : KIC 5436502
Planet(s) : Kepler-238 b , Kepler-238 c , Kepler-238 d , Kepler-238 e , Kepler-238 f (5 confirmed planets)

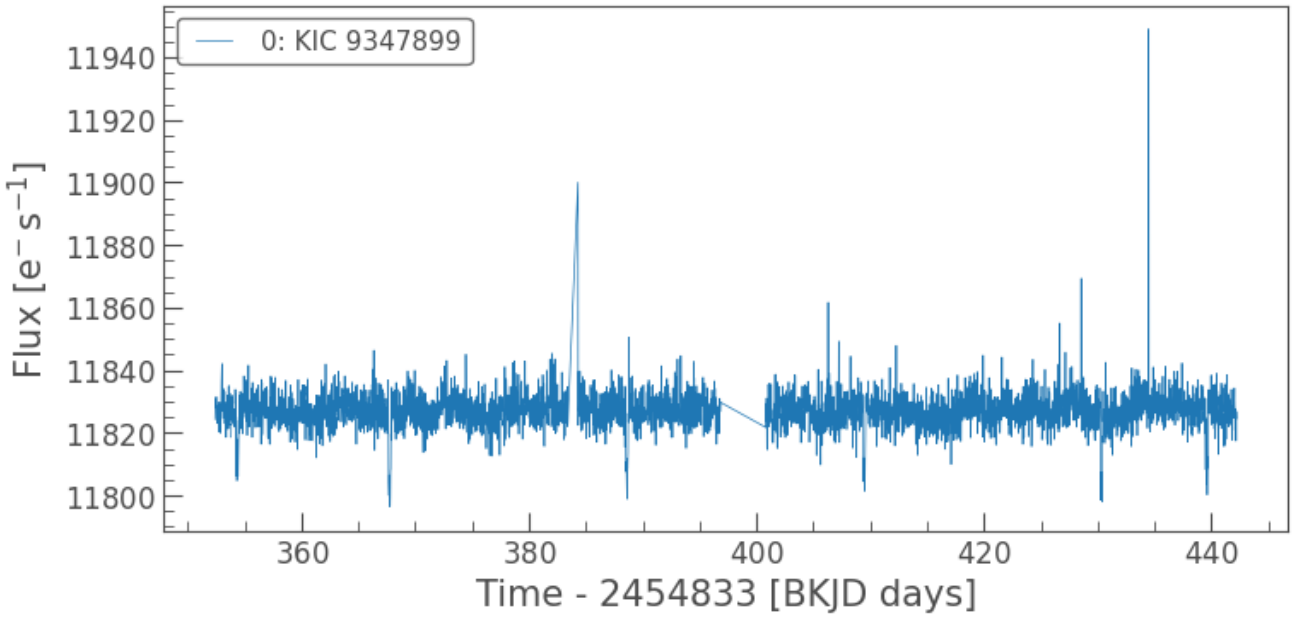


b) Star : Kepler-243
KeplerID : KIC 6587280

Planet(s) : Kepler-243 b , Kepler-243 c (2 confirmed planets)



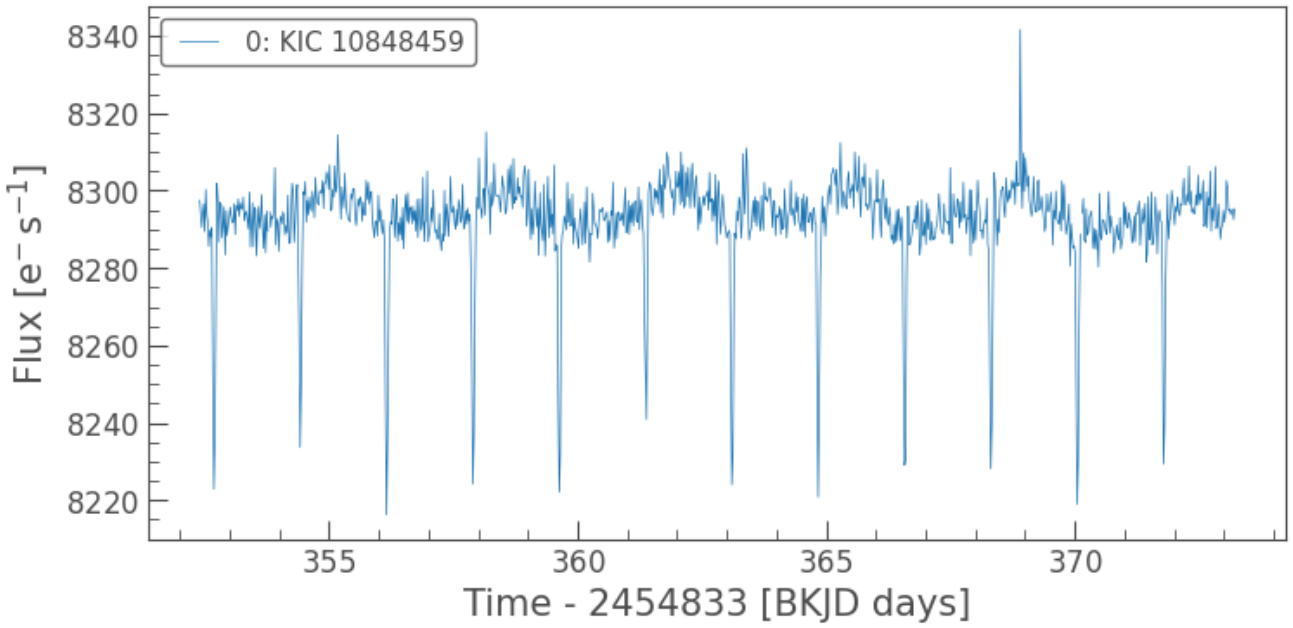
c) Star : Kepler-31
KeplerID : KIC 9347899
Planet(s) : Kepler-31 b , Kepler-31 c , Kepler-31 d (3 confirmed planets)
KOI-935.04 (1 candidate planet)



Light Curve of False positives

Below are the light curves for false positive stars. The light curves of these stars mimics the characteristics of a planetary transit but are actually caused by other astrophysical or instrumental phenomena.

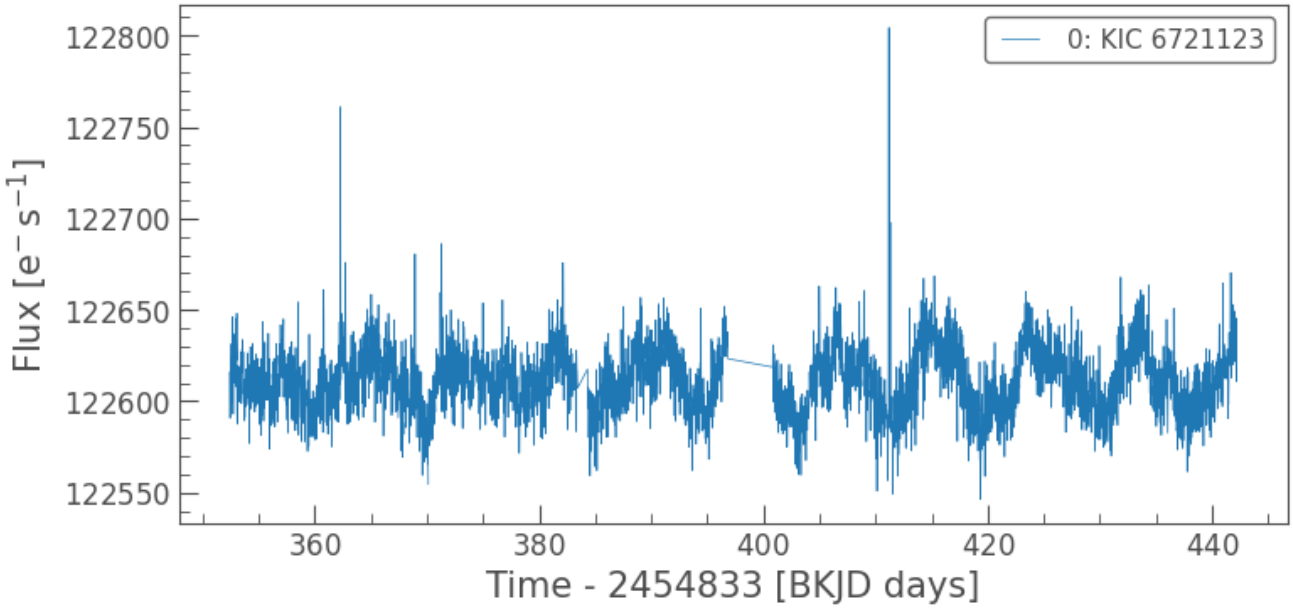
a) Star : KOI-754
KeplerID : KIC 10848459
Candidate Planet(s) : KOI-754.01



b) Star : KOI-114

KeplerID : KIC 6721123

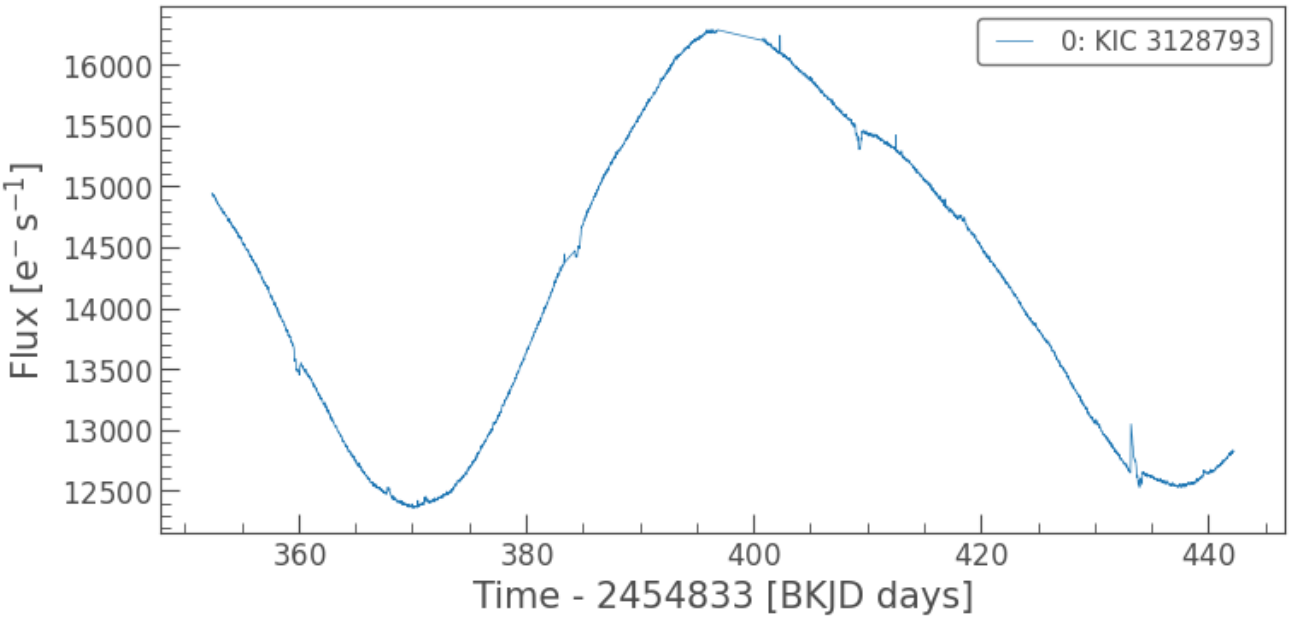
Candidate Planet(s) : KOI-114.01



c) Star : KOI-1786

KeplerID : KIC 3128793

Candidate Planet(s) : KOI-1786.01


0: KIC 3128793
Flux [e⁻ s⁻¹]
16000
15500
15000
14500
14000
13500
13000
12500
360
380
400
420
440
Time - 2454833 [BKJD days]