National College of Ireland

Configuration Manual

MSc Research Project
Programme Name

Arnab Hati
Student ID: x22107321

School of Computing
National College of Ireland

Supervisor: Christian Horn

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……Arnab Hati……

Student ID: ……x22107321……..……

Programme: ……Data Analytics…… **Year:** ……2023……..

Module: ……MSc Research Project……..……

Lecturer: ……Christian Horn……..……

Submission Due Date: ……14/12/2023……..……

Project Title: ……Exoplanet Detection by Transit Method……..……

Word Count: …2389…… **Page Count:** ……20……..……..……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: …Arnab Hati……

Date: …14/12/2023……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Arnab Hati
x22107321

1 Introduction

This document contains detailed information on the software and hardware configurations and components transit to run the code for the research project to perform the transit method for the exoplanet detection by Kepler data using various machine and deep learning models. The following steps can be considered as the configuration manual for running the code and achieving the desired results.

2 Hardware and Software Configuration

The following figure shows the technical details of the device and the Windows OS on which this research was conducted.

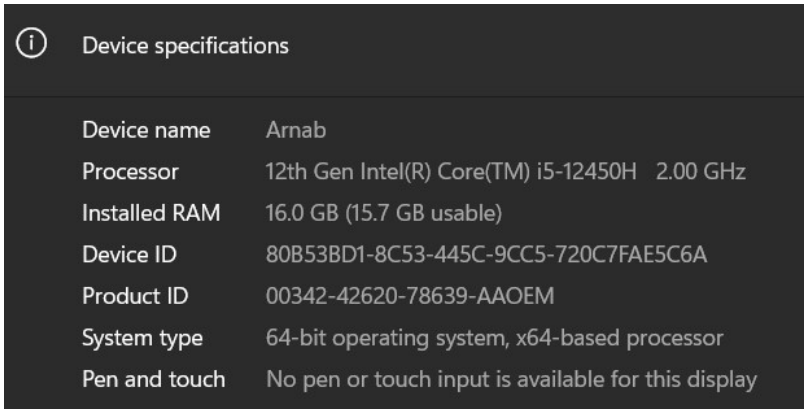


Figure 1: Device Specification

Figure 2: Windows Specifications

Python has been chosen as the language of programming and the research implementation is using Python 3.7. The setup is shown in the Figure below.

IDE	Jypyter Notebook, Anaconda
Programing Language	Python
Framework	Keras, Tensorflow
Libraries	Matplotlib, Numpy, Pandas, Seaborn, Stats, Scikit-Learn, XGBoost, CatBoost
Computation	CPU
Number of CPU	1

3 Dataset

The dataset for this thesis is available for download from NASA Directly. Here is the link to download the dataset https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=koi . The dataset consists of 49 columns with 9546 rows.

4 Import Libraries

Required Libraries are successfully executed shown in below figure.

```
#importing required libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
from scipy.stats import shapiro
from scipy.stats import anderson
from scipy.stats import probplot
from scipy.stats import boxcox
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import roc_curve, auc
from sklearn.linear_model import LogisticRegression
from statsmodels.discrete.discrete_model import Logit
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from collections import Counter
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_
from xgboost import XGBClassifier
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from catboost import CatBoostClassifier
```

5 Load the Dataset

Loaded the dataset using Pandas libraries and displayed the first 5 rows.

Load the dataste

```
# Read the data from the CSV file 'keplerData.csv' into a pandas DataFrame named 'kepler'.
kepler = pd.read_csv('keplerData.csv')
# Drop the 'kepid' column from the DataFrame 'kepler'. The axis parameter is set to 1 to indicate column-wise operation.
kepler.drop(['kepid'], axis=1, inplace=True)
# Display the first few rows of the DataFrame 'kepler' to inspect the data.
kepler.head()
```

	kepoi_name	kepler_name	koi_disposition	koi_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	...	koi_slogg	k
0	K00752.01	Kepler-227 b	CONFIRMED	CANDIDATE	1.000	0	0	0	0	9.488036	...	4.467	
1	K00752.02	Kepler-227 c	CONFIRMED	CANDIDATE	0.969	0	0	0	0	54.418383	...	4.467	
2	K00753.01	NaN	CANDIDATE	CANDIDATE	0.000	0	0	0	0	19.899140	...	4.544	
3	K00754.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.000	0	1	0	0	1.736952	...	4.564	
4	K00755.01	Kepler-664 b	CONFIRMED	CANDIDATE	1.000	0	0	0	0	2.525592	...	4.438	

6 EDA

6.1 Output variable

The code extracts unique values from the ‘koi’ column in the ‘kepler’ DataFrame, counts the occurrences of each unique value, and generates a bar plot to plot the distribution. Prints the count of the unique values in the column.

```
# Retrieve unique values from the 'koi_disposition' column in the DataFrame 'k
classes=kepler.koi_disposition.unique()
# Count the occurrences of each unique value in the 'koi_disposition' column a
counts = kepler.koi_disposition.value_counts().to_list()
# Create a bar plot using the 'classes' as x-axis values and 'counts' as y-axi
plt.figure(figsize=(5,4))
plt.bar(classes,counts)
plt.title('koi_disposition')
plt.xlabel('Different Class')
plt.ylabel('Value Counts')
# Print the count of each unique value in the 'koi_disposition' column.
print(kepler.koi_disposition.value_counts())
```

The code takes the rows in the ‘kepler’ DataFrame where the ‘koi_position’ value is ‘CANDIDATE’, changes the index, creates a bar plot, and prints the number of unique values in the updated ‘koi’_disposition column.

```
# Drop rows where koi_disposition is 'CANDIDATE'
kepler = kepler[kepler['koi_disposition'] != 'CANDIDATE']

# Reset the index after dropping rows
kepler.reset_index(drop=True, inplace=True)

# Retrieve unique values from the 'koi_disposition' column in the DataFrame 'k
classes=kepler.koi_disposition.unique()
counts = kepler.koi_disposition.value_counts().to_list()
plt.figure(figsize=(5,4))
plt.bar(classes,counts)
plt.title('koi_disposition')
plt.xlabel('Different Class')
plt.ylabel('Value Counts')
print(kepler.koi_disposition.value_counts())
```

6.2 Null Values remove

The below code snippet returns a data frame containing the top 30 rows with the highest missing percentage in the 'kepler' data frame, rounded to 2 decimal places.

```
# Rounding the values to two decimal places and presenting the top 30 columns with the highest percentage of missing values.
pd.DataFrame(round((kepler.isnull().sum() * 100/ len(kepler)),2).sort_values(ascending=False)).head(30)
```

The below code snippet iterates through the rows in the 'kepler' DataFrame that contain more than 80 % of missing data and prints the DataFrame information to display the remaining columns along with the non-null count.

```
# Dropping rows with more than 80% data missing
kepler = kepler.dropna(thresh=len(kepler) * .80, axis=1)
kepler.info()
```

The below code snippet takes the rows with missing values from the 'kepler' DataFrame and prints the DataFrame information to show the remaining number of non-null columns for each column.

```
# Drop the NULL values.
kepler = kepler.dropna()
kepler.info()
```

6.3 Check the numerical column

The code takes the DataFrame 'kepler' and finds the list of column names with integer data types (int64, float64) in it. Then, it prints the list of integer column names and the number of numbers in the data frame.

```
# Create a list of column names containing numeric data types ('int64' and 'float64') from the DataFrame 'kepler'.
numList = list(kepler.select_dtypes(include=['int64', 'float64']).columns)

# Print the list of numeric column names.
print(numList)

# Print the total number of numeric columns in the DataFrame 'kepler'.
print("Total number of numeric columns:", len(numList))
```

6.4 Visualization of Numeric Columns

Subplots are defined for each number column in the 'kepler' data frame. The code iterates through the number of columns and plots the histograms for each of the columns. The plots are sorted by the 'koi position' column. The plots are stacked vertically.

```
# Set up subplots
fig, axes = plt.subplots(nrows=len(numList), ncols=1, figsize=(6, 4 * len(numList)))

# Iterate through numeric columns and plot histograms
for i, column in enumerate(numList):
    sns.histplot(data=kepler, x=column, discrete=False, ax=axes[i], hue='koi_disposition')
    axes[i].set_title(f'Histogram of {column}')
    axes[i].set_xlabel(column)
    axes[i].set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()

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

7 Data Preprocessing

7.1 Statistical tests

The anderson-darling test is applied to each number column in the 'kepler' Data Frame. The below code snippet prints out the Anderson-darling statistic per column and checks whether the data conforms to a normal distribution with a significance level of 5.

Anderson-Darling Test: The Anderson-Darling test is used to assess whether a sample comes from a specific distribution, including normal distribution.

```
# Apply the Anderson-Darling Test on each numeric column
for column in numList:
    # Perform the Anderson-Darling test
    result = anderson(kepler[column])

    # Print the result
    print(f"\nColumn: {column}")
    print(f"Anderson-Darling Statistic: {result.statistic}")

    # Check if the data follows a normal distribution based on the 5% significance level
    if result.statistic > result.critical_values[2]:
        print("The data does not follow a normal distribution.")
    else:
        print("The data follows a normal distribution.")
```

The below code snippet takes each number column in the 'kepler' Data Frame and applies the Shapiro Wilk Test. The code prints the Shapiro-wilk statistic, and p-value and checks whether the data conforms to a normal distribution by using 5% significance.

Shapiro-Wilk Test - The Shapiro-Wilk test is used to check if a given sample comes from a normally distributed population. This test tests the null hypothesis that the data was drawn from a normal distribution.

```
# Apply the Shapiro-Wilk Test on each numeric column
for column in numList:
    # Perform the Shapiro-Wilk test
    stat, p_value = shapiro(kepler[column])

    # Print the result
    print(f"\nColumn: {column}")
    print(f"Shapiro-Wilk Statistic: {stat}")
    print(f"P-value: {p_value}")

    # Check if the data follows a normal distribution based on the 5% significance level
    if p_value > 0.05:
        print("The data follows a normal distribution.")
    else:
        print("The data does not follow a normal distribution.")
```

7.2 Data Transformation

Applied different transformations like log, and boxcox and plotted the distribution plotted the histogram plot to check whether the data is Gaussian distributed or not. The transformation is applied to all the numeric columns.

```
# Apply Box-Cox transformation
kepler['koi_period_boxcox'], lambda_value = boxcox(kepler['koi_period'])
```

```
# Create a histogram plot of the 'koi_period_boxcox' column.
plt.figure(figsize=(5,4))
sns.histplot(data = kepler, x = "koi_period_boxcox", discrete = False, kde = True, bins= 50)
plt.title('Histogram of koi_period_boxcox')
plt.xlabel('koi_period_boxcox')
plt.ylabel('Frequency')

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

7.3 Datatype Conversion

A sexagesimal-to-decimal function is defined in the code to convert the sexagesimal strings to decimals, specifically for RA coordinates. This function is then applied to the "ra_str" column in the 'kepler' DataFrame to create a new column called "ra_deg", where the RA values are expressed in decimals.

```
# Function to convert sexagesimal string to decimal degrees
def sexagesimal_to_degrees(sexagesimal):
    parts = sexagesimal.split('h')
    hours = float(parts[0])
    parts = parts[1].split('m')
    minutes = float(parts[0])
    parts = parts[1].split('s')
    seconds = float(parts[0])

    degrees = hours * 15 + minutes * 0.25 + seconds * (1/240)
    return degrees

# Apply the conversion function to the "ra_str" column
kepler['ra_deg'] = kepler['ra_str'].apply(sexagesimal_to_degrees)
# Now you have a new column "ra_deg" with the Right Ascension in decimal degrees
```

```
# Display the DataFrame with the new numeric 'dec_numeric' column
print(kepler[['ra_str', 'ra_deg']])
```

The code takes the degree, minute, and second components from the 'dec_str' column in the 'kepler' DataFrame and converts them to decimal degrees. The result is stored in the new column "dec_deg". If you need to, you can drop the intermediate columns 'dec_day', 'dec_min', and 'dec_sec' and display the DataFrame with the original 'dec_st' columns and the new 'dec deg' columns.

```
# Extract numeric components from the 'dec_str' column
kepler['dec_day'] = kepler['dec_str'].str.extract(r'([+-]?\d+)d').astype(float)
kepler['dec_min'] = kepler['dec_str'].str.extract(r'(\d+)m').astype(float)
kepler['dec_sec'] = kepler['dec_str'].str.extract(r'(\d+\.\d+)s').astype(float)
```

```
# Convert to decimal degrees
kepler['dec_deg'] = kepler['dec_day'] + kepler['dec_min'] / 60 + kepler['dec_sec'] / 3600
kepler['dec_deg']
```

```
0       48.141639
1       48.141639
2       48.285222
3       48.226194
4       48.224667
          ...
6437    47.290722
6438    46.973361
6439    47.093806
6440    47.176278
6441    47.121028
Name: dec_deg, Length: 6442, dtype: float64
```

```
# Drop intermediate columns if needed
kepler = kepler.drop(['dec_day', 'dec_min', 'dec_sec'], axis=1)

# Display the DataFrame with the new numeric 'dec_numeric' column
print(kepler[['dec_str', 'dec_deg']])
```

8 Feature Selection

8.1 Splitting X and Y

In the code, the dependent values are separated by koi_disposition from the independent values in the ‘kepler’ DataFrame. The independent values are assigned to the variable ‘X’ and the dependent values are assigned to variable ‘Y’. Next, the data type of the variable is converted to float64 data type in the 'X’ DataFrame and the information about ‘X' is printed to show the data types and the non-null count of each column.

```
# Separting dependent and independent values.
X= kepler.drop(['koi_disposition'],axis=1)
Y = kepler['koi_disposition']
```

```
# converting X to float.
X = X.apply(np.float64)
```

```
X.info()
```

8.2 Creating a function to plot the Confusion matrix and ROC curve.

The evaluate_model function in the code evaluates the performance of the machine learning model by using different metrics. The function calculates and prints the accuracy, the complete classification report, and the F1 score. It also shows the confusion matrix and the ROC (Receiver Operating Characteristic) curve. It also updates the global variables accuracy with the same values and updates the F1 (F1_score) with the same values.

```
accuracy = 0
F1_score = 0

def evaluate_model(model, X_test, y_test):

    # Calling the global veriables
    global accuracy
    global F1_score

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = round(accuracy_score(y_test, y_pred) * 100, 4)
    print("The accuracy is: ",accuracy,"\n")

    # Print full classification report
    print("Classification Report-\n", classification_report(y_test, y_pred))

    # Print F1 score
    F1_score = round(f1_score(y_test, y_pred)*100, 4)
    print("F1 Score:", F1_score)

    # Define class labels
    class_labels = ['FALSE POSITIVE', 'CONFIRMED']

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Plot confusion matrix with modified labels
    plt.figure(figsize=(4, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_labels, yticklabels=class_labels)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()
```

8.3 Without Feature Engineering

The code divides data into training data sets and testing data sets using 20% test size. First, it initializes Logistic regression model, fits Logistic regression model to training data, then uses evaluate_model on testing data to evaluate the model's performance. The function shows the model's accuracy, classification report, and F1 score. It also displays a confusion matrix and ROC curve for the Logistic regression model.

```
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(X, Y, test_size=0.20)
X_train_1.shape, X_test_1.shape, y_train_1.shape, y_test_1.shape
```

```
((5101, 41), (1276, 41), (5101,), (1276,))
```

```
model_1 = LogisticRegression()
model_1.fit(X_train_1, y_train_1)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
evaluate_model(model_1, X_test_1, y_test_1)
```

8.4 Feature Selection Using Logit Regression

The code divides the data into training data sets and testing data sets using 20% test size. Logit regression is applied to the training data using Logit. The training report is printed using the MLE (maximum likelihood estimation) method with a 500-iteration limit. The results are shown using the summary method of the Logit regression model.

```
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(X, Y, test_size=0.20)
X_train_2.shape, X_test_2.shape, y_train_2.shape, y_test_2.shape
```

```
((5101, 41), (1276, 41), (5101,), (1276,))
```

```
# Using Logit regression.
glm = Logit(y_train_2, X_train_2)
res = glm.fit(method='minimize', maxiter=500)
```

```
Optimization terminated successfully.
         Current function value: 0.023068
         Iterations: 443
         Function evaluations: 451
         Gradient evaluations: 447
```

```
#traing report of Logit regression
res.summary()
```

Logit Regression Results

The code looks for significant predictors (p-values < 0.05) in the Logistic regression results, storing them in the variable ‘significant_predictors. Next, it builds reduced training and test sets (‘X_train’ and ‘X_test’) that contain only the significant predictions. Finally, it fits a new

Logistic regression model ('model_2') using the reduced subset of predictions and evaluates the model's performance on the test data using the evaluate model() function.

```
# Identify predictors with high p-values and drop them
significant_predictors = res.pvalues[res.pvalues < 0.05].index
significant_predictors
```

```
Index(['koi_score', 'koi_fpflag_ss', 'koi_time0bk_err2', 'koi_impact_err2',
       'koi_duration_err2', 'koi_prad_err1', 'koi_tce_plnt_num',
       'koi_steff_err2', 'koi_srad', 'koi_srad_err1', 'koi_kepmag', 'dec_deg'],
      dtype='object')
```

```
X_train_reduced = X_train_2[significant_predictors]
X_test_reduced = X_test_2[significant_predictors]
```

```
X_train_reduced.shape, X_test_reduced.shape, y_train_2.shape, y_test_2.shape
```

```
((5101, 12), (1276, 12), (5101,), (1276,))
```

```
# Create a logistic regression model
model_2 = LogisticRegression()
```

```
# Fit the model using the reduced set of significant predictors
model_2.fit(X_train_reduced, y_train_2)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
evaluate_model(model_2, X_test_reduced, y_test_2)
```

8.5 Using Pearson Correlation

The code calculates the correlation matrix (cor) for the 'kepler' DataFrame and creates a heatmap (using seaborn) using the 'Reds' color map. Each cell in the heatmap is not annotated. The plot below shows the correlation between different columns of the DataFrame.

```
plt.figure(figsize=(8,8))
# Compute the correlation matrix for the DataFrame 'kepler' and store it in the variable 'cor'.
cor = kepler.corr()
# Generate a heatmap of the correlation matrix using seaborn, without annotating each cell, and using the 'Reds' color map.
sns.heatmap(cor, annot=False, cmap=plt.cm.Reds)
# Display the plot.
plt.show()
```

The below code snippet calculates the absolute relationship between each feature in the 'kepler' DataFrame and the target variable 'koi_position'. Select the features with a correlation higher than 0.5 to the target variable and store their indices in the 'relevant_features' variable. Convert this to a data frame. Create a new DataFrame 'data' containing only the corresponding features from the original DataFrame.

```
# Calculate the absolute correlation between each feature and the target variable 'koi_disposition'.
cor_target = abs(cor["koi_disposition"])
# Select features with a correlation greater than 0.5 with the target variable.
relevant_features = cor_target[cor_target>0.5]
relevant_features
```

```
koi_disposition    1.000000
koi_score          0.947584
koi_fpflag_ss      0.577558
koi_fpflag_co      0.511871
koi_prad_err1      0.513195
koi_prad_err2      0.516355
Name: koi_disposition, dtype: float64
```

```
# Convert the index of the relevant features to a DataFrame.
relevant_features = pd.DataFrame(relevant_features).index
relevant_features
```

```
Index(['koi_disposition', 'koi_score', 'koi_fpflag_ss', 'koi_fpflag_co',
       'koi_prad_err1', 'koi_prad_err2'],
      dtype='object')
```

```
# Create a new DataFrame 'data' containing only the relevant features.
data = kepler[relevant_features]
data
```

The code decouples the features ‘X’ and the target variable ‘y’ from the DataFrame ‘data’. It divides the data into training data and testing data with 20% test size. The Logistic regression model ‘model_3’ is trained on training data and evaluated with evaluate_model on testing data. Print the shapes of training and testing sets.

```
# Separate the features (X) and the target variable (y) from the DataFrame 'data'.
x= data.drop(['koi_disposition'],axis=1)
y = data['koi_disposition']
```

```
# Split the data into training and testing sets using a test size of 20%.
X_train_3, X_test_3, y_train_3, y_test_3 = train_test_split(x, y, test_size=0.20)
# Print the shapes of the training and testing sets.
X_train_3.shape, X_test_3.shape, y_train_3.shape, y_test_3.shape
```

```
((5101, 5), (1276, 5), (5101,), (1276,))
```

```
model_3 = LogisticRegression()
model_3.fit(X_train_3, y_train_3)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
evaluate_model(model_3, X_test_3, y_test_3)
```

8.6 Using Feature Importance Techniques

A logistic regression model (model_4) is created in the code and fitted to the features 'X' and target variable 'Y'. The code extracts the logistic regression coefficients and generates a DataFrame (dataFrame_importance) with columns 'Feature' and 'Importance' based on the logistic regression coefficient absolute values. DataFrame is sorted by descending order of importance. Horizontal bar plots are generated to show the feature importance.

```
# Create a logistic regression model instance.
model_4 = LogisticRegression()
# Fit the logistic regression model to the features 'X' and target variable 'Y'.
model_4.fit(X, Y)
```

```
LogisticRegression()
```

```
# Extract the coefficients of the logistic regression model.
coefficients = model_4.coef_[0]
# Display the coefficients as a numpy array.
coefficients
```

```
array([ 6.06870283e+00, -8.04854231e-03, -2.55660041e+00, -3.41213973e+00,
       -1.99938152e+00,  1.24587068e+00, -1.14311396e-01, -4.60810482e-03,
       -5.20640815e-01, -6.71745222e-01,  4.26395817e-03,  3.07263425e-01,
       -1.26960609e-01, -4.63470220e-01, -4.58306634e-01, -1.59543063e+00,
        7.56305723e-01, -1.80865002e-01,  1.48073605e-01, -9.56081269e-02,
       -1.17039784e+00, -3.94177041e-01,  5.04836461e-01, -2.05190550e-01,
       -3.35120227e-01,  1.15982457e-01, -7.12825585e-01, -6.72021310e-01,
        7.60011593e-01,  7.24195862e-03, -3.32421806e-01,  5.09739349e-03,
       -1.26497943e-02, -2.63680524e-01, -1.22284222e-01, -1.00688907e-01,
        6.66966986e-01,  1.93665286e-01,  2.26561381e-01, -1.41178861e-01,
        1.12549981e-02])
```

```
# Create a DataFrame 'feature_importance' with columns 'Feature' and 'Importance' based on the absolute values of the coefficient
feature_importance = pd.DataFrame({'Feature': X.columns, 'Importance': np.abs(coefficients)})
# Sort the DataFrame by 'Importance' in descending order.
feature_importance = feature_importance.sort_values('Importance', ascending=False)
# Create a horizontal bar plot of feature importance.
feature_importance.plot(x='Feature', y='Importance', kind='barh', figsize=(8, 6), color='darkgreen')
```

The code finds the top k significant features (i.e., k = 20) according to the feature importance as calculated from logistic regression models. The code compiles the list of features and then compiles the new DataFrame 'x', which contains only the selected features of the original DataFrame 'X.' The final DataFrame is shown below.

```
# Select the top k important features.
k = 20
selected_features = feature_importance.head(k)['Feature'].tolist()
selected_features
```

```
['koi_score',
 'koi_fpflag_co',
 'koi_fpflag_ss',
 'koi_fpflag_ec',
 'koi_duration_err1',
 'koi_period',
 'koi_prad',
 'koi_tce_plnt_num',
 'koi_duration_err2',
 'koi_insol_err2',
 'koi_model_snr',
 'koi_time0bk_err1',
 'koi_srad_err1',
 'koi_time0bk',
 'koi_prad_err2',
 'koi_impact_err2',
 'koi_duration',
 'koi_prad_err1',
 'koi_insol',
 'koi_steff_err1']
```

```
# Create a new DataFrame 'x' containing only the selected features from the original features 'X'.
x= X[selected_features]
# Display the selected features DataFrame.
x
```

Using the selected features ‘x’ and the target variable ‘Y’, the code divides the data into ‘training’ and ‘testing’ sets. The training set is 20% of the test size, and the testing set is 20%. The code then initializes the logistic regression model ‘model_5’, fits the model to the training set, and evaluates the model’s performance using evaluate_model on the test data. Print the shapes of training and testing sets.

```
X_train_5, X_test_5, y_train_5, y_test_5 = train_test_split(x, Y, test_size=0.20)
X_train_5.shape, X_test_5.shape, y_train_5.shape, y_test_5.shape
```

```
((5101, 20), (1276, 20), (5101,), (1276,))
```

```
model_5 = LogisticRegression()
model_5.fit(X_train_5, y_train_5)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
evaluate_model(model_5, X_test_5, y_test_5)
```

8.6 Using PCA

The code generates a PCA model with the goal of preserving 95% of variance in the raw data. The PCA model is fitted to the features ‘X’. The explained variance ratio is printed for each principal component. The original features are transformed using the fitted model. The converted data is passed to the variable ‘x’ and the shape of the transformed data is shown. The data is then divided into training data and testing data. A Logistic regression model (model_6) is trained on training data, and evaluated using evaluate_model on testing data. Print the shapes of training and testing data for reference.

```
# Create a PCA (Principal Component Analysis) model with the goal of retaining 95% of the variance in the original data.
pca = PCA(n_components=0.95, svd_solver='full')
```

```
# Fit the PCA model to the features 'X'.
pca.fit(X)
# Print the explained variance ratio of each principal component.
print(pca.explained_variance_ratio_)
# Print the singular values corresponding to each principal component.
print(pca.singular_values_)
# Transform the original features 'X' using the fitted PCA model and assign the result to the variable 'x'.
x= pca.transform(X)
# Display the shape of the transformed data.
x.shape
```

```
[0.91055431 0.07469923]
[6053.27252441 1733.7854902 ]

(6377, 2)
```

```
X_train_6, X_test_6, y_train_6, y_test_6 = train_test_split(x, Y, test_size=0.20)
X_train_6.shape, X_test_6.shape, y_train_6.shape, y_test_6.shape
```

```
((5101, 2), (1276, 2), (5101,), (1276,))
```

```
#model = DecisionTreeClassifier()
model_6 = LogisticRegression()
model_6.fit(X_train_6, y_train_6)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
evaluate_model(model_6, X_test_6, y_test_6)
```

8.7 Using Chi-Square

The chi2 function is used in the code to compare each feature in 'X' with the target variable 'Y'. The chi2 function can be found in scikit's library. The chi-squared scores of each feature are shown below.

```
# Compute chi-squared statistics between each feature in 'X' and the target variable 'Y'.
chi_scores = chi2(np.abs(X),Y)
# Display the computed chi-squared scores for each feature.
chi_scores
```

Code Generates a Pandas Series with p-values from Chi-squared test with column names 'X' as indices. Sort p-values descending in descending order. Create a bar plot using Plot.bar() to visualize the importance of each feature for predicting the target variable 'Y'. Plot in Slate Blue.

```
# Create a pandas Series containing p-values from the chi-squared test, indexed by the column names of 'X'.
p_values = pd.Series(chi_scores[1],index = X.columns)
# Sort the p-values in descending order, updating the 'p_values' Series in-place.
p_values.sort_values(ascending = False , inplace = True)
```

```
# Create a bar plot of the p-values using the 'plot.bar()' function.
p_values.plot.bar(color='slateblue')
```

The code generates a DataFrame called 'featureScores' based on the converted chi-square scores. The columns represent features and the scores are shown in the dataFrame.

```
# Create a DataFrame 'featureScores' from the transposed chi-squared scores, with columns representing featu
featureScores = pd.DataFrame(chi_scores).T
# Display the DataFrame showing features and their respective chi-squared scores.
featureScores
```

The code changes the name of the Chi-Scores columns in the 'featureScores' DataFrame to 'p-Value'. It also changes the column names for the original features 'X' to 'Features'. Finally, it filters the DataFrame to only include rows that have a p-Value of 0.05 or higher. Finally, it displays the 'features' column in the filtered 'factorScores' dataFrame.

```
# Rename the columns of the 'featureScores' DataFrame to 'Chi-Scores' and 'p-Value'.
featureScores.columns =['Chi-Scores', 'p-Value']
# Assign the column names of the original features 'X' to a new column 'Features' in the 'featureScores' DataFrame.
featureScores['Features'] = X.columns
# Filter the 'featureScores' DataFrame to include only rows where the 'p-Value' is greater than 0.05.
featureScores = featureScores[featureScores['p-Value']>0.05]
# Display the 'Features' column of the filtered 'featureScores' DataFrame.
featureScores['Features']
```

The code takes the 'Features' column from the 'featureScores' DataFrame and transforms it into a 'bestCols' based NumPy array. Then, it creates a new 'x' DataFrame that contains only the features selected from the original 'X' based on bestCols. The resulting 'BestCols' and 'DataFrame' are shown.

```
# Extract the 'Features' column from the 'featureScores' DataFrame and convert it to a NumPy array.
bestCols = np.array(featureScores['Features'])
# Display the resulting NumPy array containing the selected features.
bestCols
```

```
array(['koi_period_err2', 'koi_time0bk', 'koi_time0bk_err2',
       'koi_insol_err2', 'koi_steff', 'koi_slogg_err2', 'koi_kepmag',
       'ra_deg'], dtype=object)
```

```
# Create a new DataFrame 'x' containing only the selected features from the original features 'X' based on 'bestCols'.
x=X[bestCols]
# Display the resulting DataFrame with the selected features.
x
```

9 Model Preparation

9.1 XGBoost

XGBoost Classifier with random state (369) is created in the code. XGBoost is trained on the X_train and Y_train data sets, and XGBoost makes predictions on the test data set (X_test) using the trained model. The code calculates the accuracy, the weighted F1 score and shows the classification report. It also plots a confusion matrix to visualize the model's performance on the test data.

```
# Create an XGBoost classifier model with a specified random state (369).
xgb = XGBClassifier(random_state=369)
```

```
# Train (fit) the XGBoost classifier model on the training data (X_train and y_train).
xgb.fit(X_train, y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
```

```
# Use the trained XGBoost classifier model to make predictions on the testing data (X_test).
pred_XGB = xgb.predict(X_test)
```

```
# Calculate the accuracy of the XGBoost classifier predictions on the testing data with three decimal places.
accuracyXGB= np.round(accuracy_score(y_test, pred_XGB)*100,5)
print("Accuracy = ", accuracyXGB)
```

```
Accuracy =  98.98119
```

```
# Calculate the weighted F1 score of the XGBoost classifier predictions on the testing data.
f1= np.round(f1_score(y_test, pred_XGB, average ='weighted')*100,2)
print("F1 = ", f1)
```

9.2 CatBoost

The code will generate a CatBoost Classifier model (catBoostClassifier), train the model on the X_train and Y_train data sets, and use the trained model to generate predictions on the test data set (X_test), and print the results (catBoost classifier accuracy, weighted F1 score, etc.) on the testing data set.

```
cat = CatBoostClassifier()
```

```
cat.fit(X_train, y_train)
```

```
5:      learn: 0.4481513        total: 242ms    remaining: 40.1s
6:      learn: 0.4151694        total: 250ms    remaining: 35.4s
7:      learn: 0.3875122        total: 255ms    remaining: 31.7s
8:      learn: 0.3566081        total: 263ms    remaining: 28.9s
9:      learn: 0.3297559        total: 268ms    remaining: 26.5s
10:     learn: 0.3109501        total: 273ms    remaining: 24.6s
11:     learn: 0.2885121        total: 280ms    remaining: 23s
12:     learn: 0.2665888        total: 285ms    remaining: 21.6s
13:     learn: 0.2510966        total: 290ms    remaining: 20.4s
14:     learn: 0.2333119        total: 294ms    remaining: 19.3s
15:     learn: 0.2165394        total: 298ms    remaining: 18.3s
16:     learn: 0.2018615        total: 303ms    remaining: 17.5s
17:     learn: 0.1884861        total: 306ms    remaining: 16.7s
18:     learn: 0.1764877        total: 310ms    remaining: 16s
19:     learn: 0.1647629        total: 313ms    remaining: 15.4s
20:     learn: 0.1545004        total: 317ms    remaining: 14.8s
21:     learn: 0.1446836        total: 321ms    remaining: 14.3s
22:     learn: 0.1355705        total: 325ms    remaining: 13.8s
23:     learn: 0.1288222        total: 329ms    remaining: 13.4s
```

```
pred_Cat = cat.predict(X_test)
```

```
accuracyCat= np.round(accuracy_score(y_test, pred_Cat)*100,5)
print("Accuracy = ", accuracyCat)
```

```
Accuracy =  99.2163
```

```
# Calculate the weighted F1 score of the CatBoost classifier predictions on the testing data with three decimal places.
f1= np.round(f1_score(y_test, pred_Cat, average='weighted')*100,2)
print("F1 = ", f1)
```

9.3 Variational Encoder

The code transforms the training and test data for the neural network using Keras. It transforms the DataFrame X_train to a NumPy array and reshapes it to have an extra dimension of 1. Prints the shape of the transformed array. Set the random seed from Keras to be reproducible.

```
X_train = X_train.to_numpy().reshape(X_train.shape[0], X_train.shape[1], 1)
```

```
X_test = X_test.to_numpy().reshape(X_test.shape[0], X_test.shape[1], 1)
```

```
X_train.shape, X_test.shape
```

```
((5101, 20, 1), (1276, 20, 1))
```

```
keras.utils.set_random_seed(0)
```

The code defines the encoder layer in a neural network. The encoder layer is defined using Keras. The input data is defined as (X_test.shape [1], 1). The encoder is built using SimpleRNN layers. The first SimpleRNN layer has 512 units and is set to return sequences. The second layer has 128 units and is also set to return sequences and the third layer has 64 units. The last layer does not return any sequences. This layer serves as the final encoder layer.

```
# Defining the input layer for the encoder with the shape of the input data
encoder_inputs = keras.Input(shape=(X_test.shape[1],1))
# Creating the first SimpleRNN layer with 512 units and set to return sequences
# This layer processes input sequences and produces output sequences with the same length
encoder = layers.SimpleRNN(512, return_sequences=True)(encoder_inputs)
# Creating a second SimpleRNN layer with 128 units and set to return sequences
encoder = layers.SimpleRNN(128, return_sequences=True)(encoder)
# Creating a third SimpleRNN layer with 64 units
# This layer does not return sequences, indicating it's the final encoding layer
encoder = layers.SimpleRNN(64)(encoder)
```

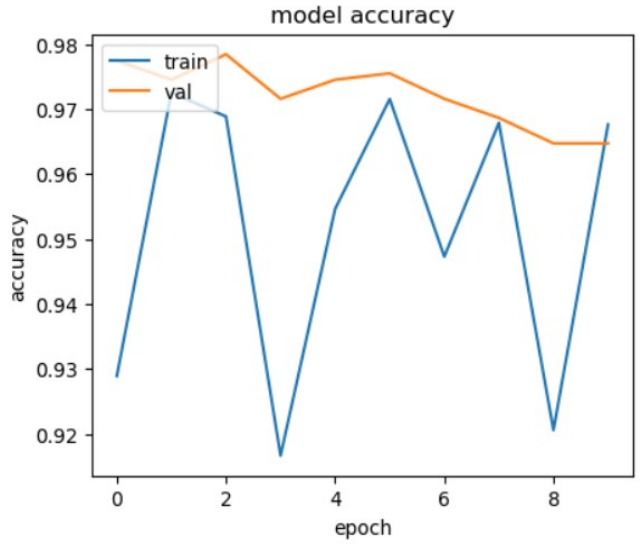
Using Keras, the code builds the output layer for the VAE neural network. The output layer consists of a Dense layer (64 units) and a Sigmoid activation function (1). To avoid overfitting, a Dropout layer (20% dropout rate) is added to the output layer. Finally, the decoder output layer (4 units) is created using a sigmoidal activation function. The overall model is named “VAR”. It is compiled with the Adam optimizer (Sparse categorical Cross-entropy loss) and a summary is printed.

```
# Creating a Dense layer with 64 units and sigmoid activation function for the output layer
output = layers.Dense(units=64, activation='sigmoid')(encoder)
# Adding a Dropout layer with a dropout rate of 20% to prevent overfitting
output = layers.Dropout(.2)(output)
# Creating a Dense layer with 4 units and sigmoid activation function for the decoder output
decoder = layers.Dense(units=4, activation='sigmoid')(output)
```

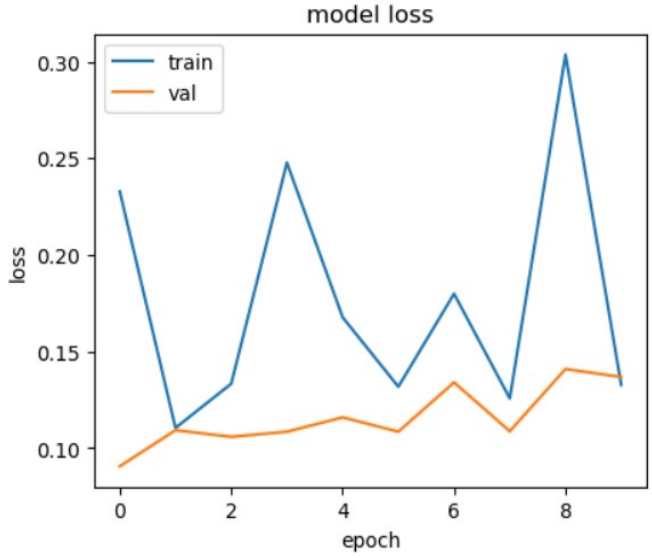
```
var = keras.Model(encoder_inputs, decoder, name="VAR")
var.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
var.summary()
```

Matplotlib is used in the code to plot training and validation accuracies over epochs for the neural network model. The x-axis indicates the number of training and validation epochs, and the y-axis indicates the accuracy of the model. The plot shows how the model's accuracy changes over training and validation.

```
plt.figure(figsize=(5,4))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```




```
# Plot Loss vs Epoch
plt.figure(figsize=(5,4))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



9.4 RNN

The code defines the Sequential model 'rnn' in Keras as follows: SimpleRNN Layer 32 Units (Input Shaped X_test.Shape[1]], 1) Dense Layer 10 Units (ReLU Activation) Next Layer 4 Units (Sigmoid Activation)

```
rnn = keras.Sequential([
    layers.SimpleRNN(32, input_shape=(X_test.shape[1], 1)),
    layers.Dense(10, activation='relu'),
    layers.Dense(4, activation='sigmoid')
])
rnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_3 (SimpleRNN)    (None, 32)                1088

 dense_2 (Dense)             (None, 10)                330

 dense_3 (Dense)             (None, 4)                 44

=================================================================
Total params: 1462 (5.71 KB)
Trainable params: 1462 (5.71 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Code compiles the keras sequential model 'rnn' with sparse categorical crosstabs loss as a metric and model accuracy as a metric. Prints the model summary. Train the model using X_train & y_train for 10 epoch with 20% validation split. History is stored in 'history' variable.

```
rnn.compile(loss="sparse_categorical_crossentropy", metrics=["accuracy"])
rnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_3 (SimpleRNN)    (None, 32)                1088

 dense_2 (Dense)             (None, 10)                330

 dense_3 (Dense)             (None, 4)                 44

=================================================================
Total params: 1462 (5.71 KB)
Trainable params: 1462 (5.71 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history = rnn.fit(X_train, y_train, epochs = 10, validation_split=0.20)
```

9.5 GRU

The code will compile the keras sequential model 'rnn' using sparse categorical cross entropy loss and accuracy as metrics. The model summary will be printed. The model will be trained using X_train and Y_train for 10 epochs, with 20% validation split. The training history will be stored in the 'history' variable.

```
keras.utils.set_random_seed(0)
```

```
gru = keras.Sequential([
    layers.GRU(32, input_shape=(X_test.shape[1], 1)),
    layers.Dense(10, activation='relu'),
    layers.Dense(4, activation='sigmoid')
])
```

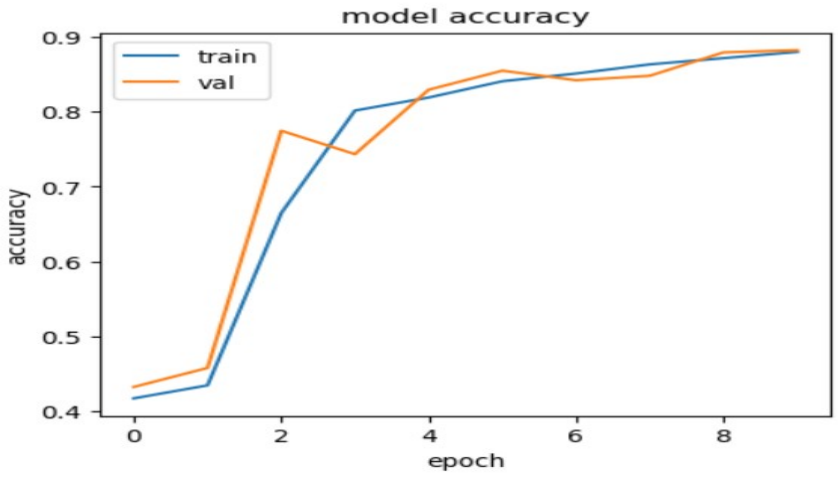
```
gru.compile(loss="sparse_categorical_crossentropy", metrics=["accuracy"])
gru.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 32)                3360

 dense_4 (Dense)             (None, 10)                330

 dense_5 (Dense)             (None, 4)                 44

=================================================================
Total params: 3734 (14.59 KB)
Trainable params: 3734 (14.59 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history = gru.fit(X_train, y_train, epochs = 10, validation_split=0.20)
```

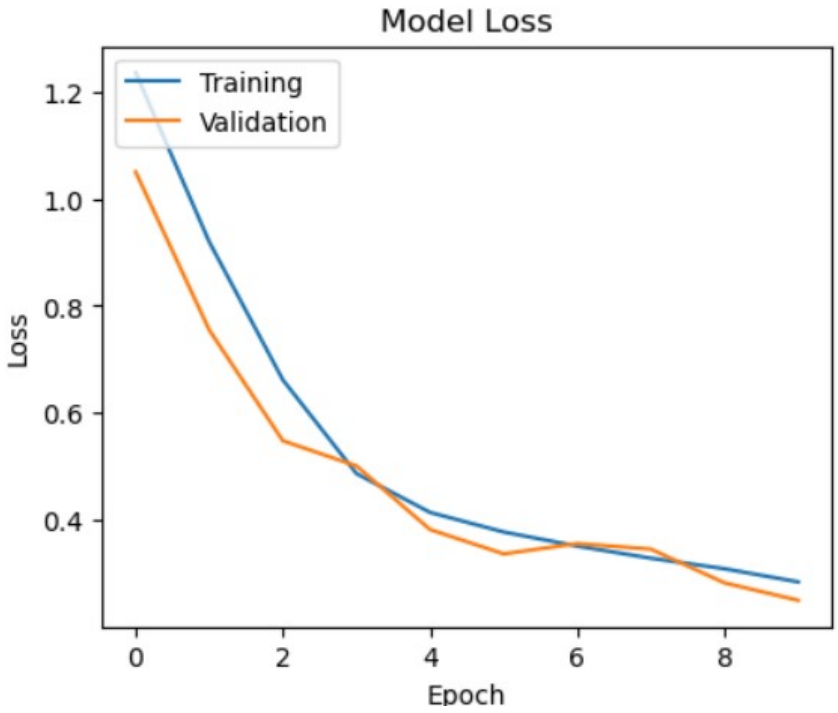
Matplotlib is used in the code to plot GRU training and validation accuracies over epochs. The x axis represents GRU training and the y axis represents GRU validation accuracies. The plot shows how the model's accuracy changes over training and validation.

```
plt.figure(figsize=(5,4))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



Matplotlib is used in the code to plot the training loss and validation loss over the epochs for the GRU neural network model. The x-axis indicates the number of training and validation epochs, and the y-axis indicates the loss. The plot shows how the model's loss changes over the training and validation period.

```
plt.figure(figsize=(5, 4))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.show()
```



9.6 Evaluation

In the code, we sort a data frame called modelScores according to the ‘Accuracy’ column by descending order. Then, we use Seaborn to create a bar plot. The ‘x’ axis represents different models and the ‘y’ axis represents their accuracy scores.

```
# Assuming modelScores is your DataFrame
plt.figure(figsize=(5, 4))
sns.barplot(x='Models', y='Accuracy', data=modelScores, palette='viridis')
plt.xticks(rotation=55, ha='right')  # Rotate x-axis labels for better readability
plt.xlabel('Models')
plt.ylabel('Accuracy Scores')
plt.title('Accuracy Scores for Different Models')
plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```