DIGITAL FAIR SKILL 35.0
STAR TYPE CLASSIFICATION WITH

K-Nearest Neighbors

Created by: Naufal Fathi Rizqy Fadhilah

INTRODUCTION

The dataset contains information about various types of stars, including temperature, luminosity, radius, and absolute magnitude.

Research on stars and their classification is essential in astronomy. K-Nearest Neighbors is a machine learning algorithm well-suited for classification tasks.

Objective: To apply the K-Nearest Neighbors algorithm to classify star types based on their characteristics.

LIBRARY IMPORT

```
[ ]  import pandas as pd  # Import the Pandas library for data manipulation and analysis
     import seaborn as sns  # Import the Seaborn library for creating data visualizations such as graphs or charts
     import matplotlib.pyplot as plt  # Import Matplotlib to generate graphs or plot data
     import numpy as np  # Import Numpy for mathematical operations such as arrays and statistical functions
     from scipy.stats import boxcox  # Import the Box-Cox function for data transformation to approximate a normal distribution
     from sklearn.preprocessing import LabelEncoder  # Import LabelEncoder to convert categorical data into numerical values
     from sklearn.model_selection import train_test_split  # Import the function to split data into training and testing sets
     from sklearn.preprocessing import StandardScaler  # Import StandardScaler for data standardization
     from sklearn.neighbors import KNeighborsClassifier  # Import the K-Nearest Neighbors algorithm for classification
     from sklearn.metrics import classification_report  # Import the function to generate a model evaluation report
     from sklearn.metrics import confusion_matrix  # Import the function to create a confusion matrix
```

DATA DESCRIPTION

Displays the last 5 rows and dataset size (number of rows and columns) using data.tail() and data.shape.

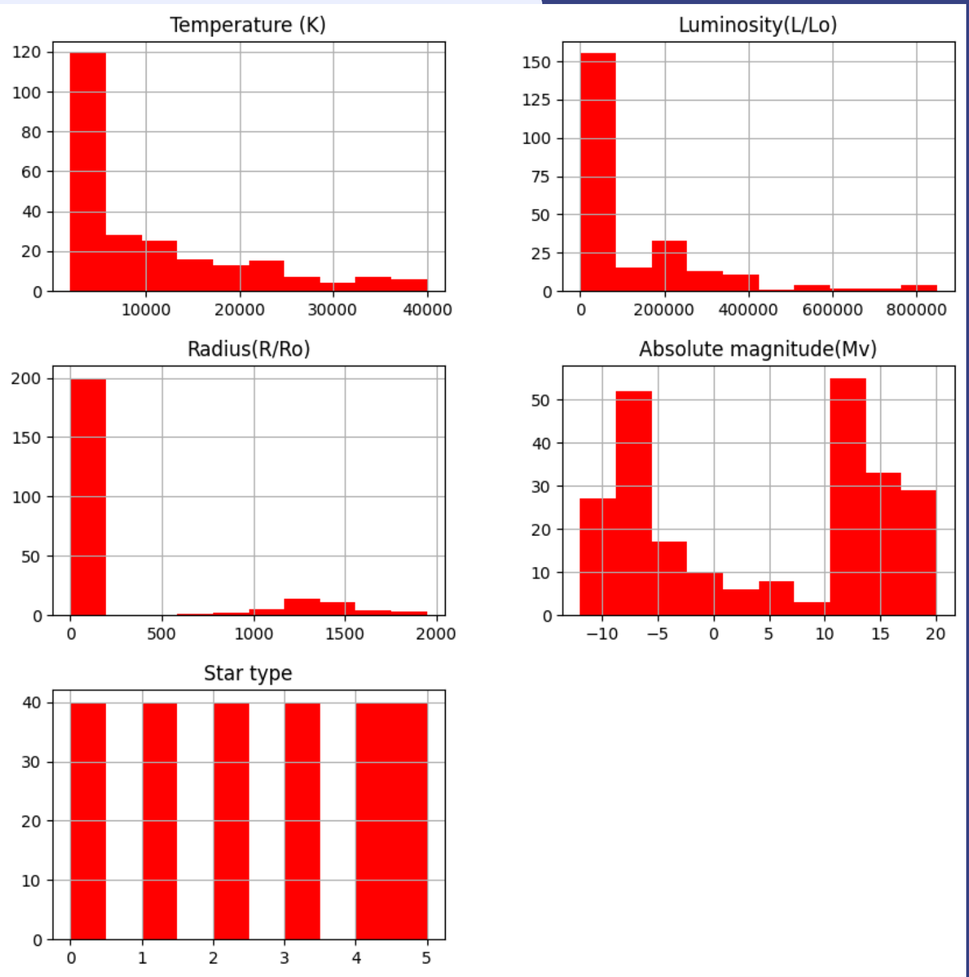
```
[ ]  data = pd.read_csv('/content/6 class csv.csv') # read file csv
     data.tail() # displat the last 5 lines of data
```

	Temperature (K)	Luminosity(L/Lo)	Radius(R/Ro)	Absolute magnitude(Mv)	Star type	Star color	Spectral Class
235	38940	374830.0	1356.0	-9.93	5	Blue	O
236	30839	834042.0	1194.0	-10.63	5	Blue	O
237	8829	537493.0	1423.0	-10.73	5	White	A
238	9235	404940.0	1112.0	-11.23	5	White	A
239	37882	294903.0	1783.0	-7.80	5	Blue	O

```
data.shape # Display the number of rows and columns
```

```
(240, 7)
```

DATA EXPLORATION



DATA VISUALIZATION

1. Histogram: Displays data distribution using data.hist().

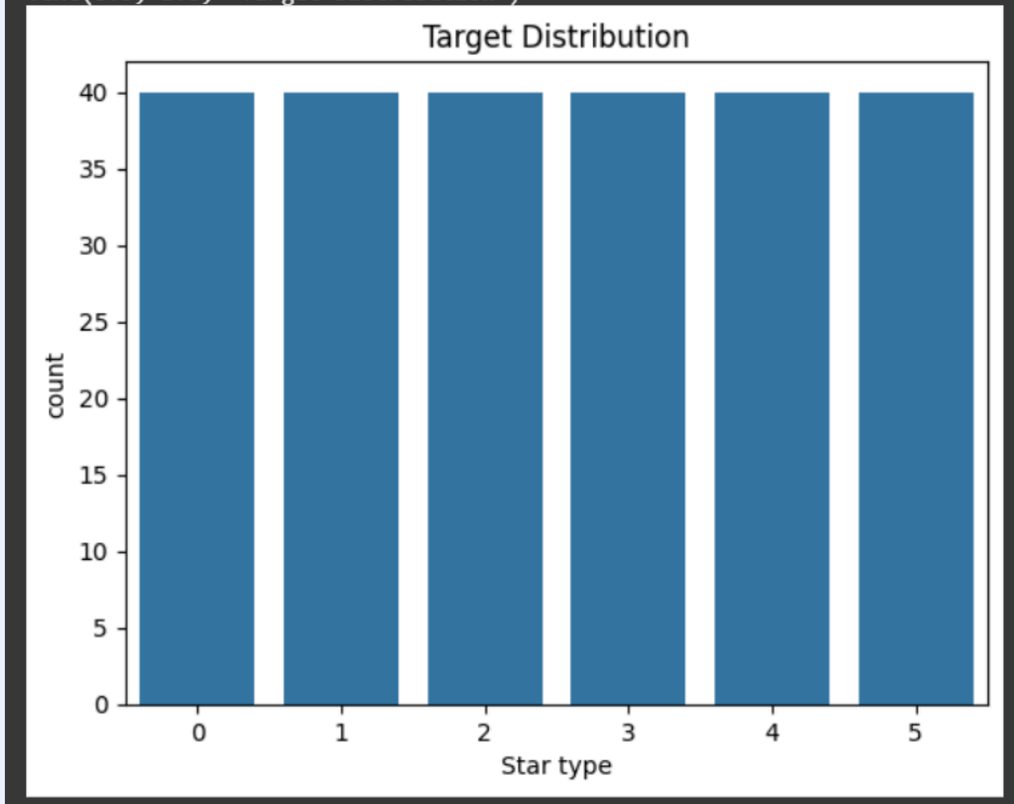
DATA VISUALIZATION

2. Boxplot: Detects outliers in each variable using the boxplot() function.

```
def boxplot(data):
    # Defines a function named 'boxplot' to create boxplots for multiple variables in the DataFrame 'data'

    variables = ['Star type', 'Temperature (K)', 'Luminosity(L/Lo)', 'Radius(R/Ro)', 'Absolute magnitude(Mv)', 'Spectral Class']

    fig, axes = plt.subplots(2, 3, figsize=(10, 5))
    # Creates a 2x3 grid for subplots with a total size of 10x5 inches

    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    # Adjusts the spacing between plots both vertically and horizontally

    for i, var in enumerate(variables):
        row, col = i // 3, i % 3

        sns.boxplot(y=var, data=data, color='green', ax=axes[row, col])
        # Creates a boxplot for the variable 'var' in a specific subplot with the color green

        axes[row, col].set_title(var)
        # Sets the title for each subplot with the variable name

    plt.show()
```

```
# Creating count plot for variabel 'Star type' from DataFrame 'data_shorten'
sns.countplot(x='Star type', data=data_shorten)
plt.title('Target Distribution')
```

```
Text(0.5, 1.0, 'Target Distribution')
```



DATA VISUALIZATION

3 Countplot: Shows the distribution of the target class (star type) using sns.countplot().

DATA CLEANING

1. Removing outliers using the remove_outlier() function.

```
# to clean the data from outliers that can affect the data.
def remove_outlier(data):
  columns = data.select_dtypes(include=['int', 'float']).columns
  cleaned_data = data.copy()
  for column in columns:
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3-Q1
    lower_bound = Q1-1.5*IQR
    upper_bound = Q3+1.5*IQR
    cleaned_data = cleaned_data[(cleaned_data[column]>=lower_bound)&(cleaned_data[column]<=upper_bound)]

  return cleaned_data

data_shorten = remove_outlier(data_shorten)
```

DATA CLEANING

2. Removing missing values and duplicate data using data.dropna() and data.drop_duplicates().

```
[ ]  data = data.dropna()
```

```
[ ]  # remove duplicate data
     data.drop_duplicates(inplace= True)
```

```
     # displays unique data
     data.nunique()
```

	0
Temperature (K)	228
Luminosity(L/Lo)	208
Radius(R/Ro)	216
Absolute magnitude(Mv)	228
Star type	6
Star color	19
Spectral Class	7

dtype: int64

DATA PREPROCESSING

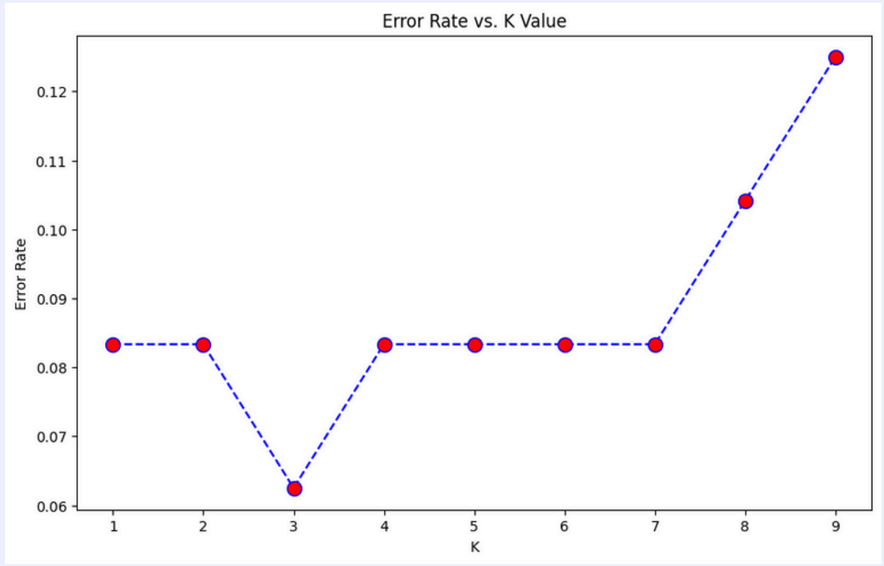
ENCODING

Converts categorical data into numerical values using LabelEncoder for columns such as 'Luminosity', 'Radius', 'Absolute Magnitude', 'Star Color', and 'Spectral Class'.
Standardization: Scales feature data using StandardScaler.
Data Splitting: Splits data into training (80%) and testing (20%) sets using train_test_split.

```
# coding process

le = LabelEncoder()

data['Luminosity(L/Lo)'] = le.fit_transform(data['Luminosity(L/Lo)'])
data['Radius(R/Ro)'] = le.fit_transform(data['Radius(R/Ro)'])
data['Absolute magnitude(Mv)'] = le.fit_transform(data['Absolute magnitude(Mv)'])
data['Star color'] = le.fit_transform(data['Star color'])
data['Spectral Class'] = le.fit_transform(data['Spectral Class'])
```

```
[ ] # separating features (independent variables) from targets (dependent variables)
    X = data.drop(columns='Star type')
    y = data['Star type']

[ ] # feature standardization (scaling the feature data)
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

[ ] # splitting data by dividing 80% train data and 20% testing data
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# determine the best value for parameter K (number of neighbors)
error_rate = []

for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(x_train,y_train)
    pred_i = knn.predict(x_test)
    error_rate.append(np.mean(pred_i != y_test))

plt.figure(figsize=(10,6))
plt.plot(range(1,10),error_rate,color="blue", linestyle="dashed", marker="o",
         markerfacecolor="red", markersize=10)
plt.title("Error Rate vs. K Value")
plt.xlabel("K")
plt.ylabel("Error Rate")
```



MODELING WITH K-NEAREST NEIGHBORS

1. CHOOSING K VALUE

Determines the optimal K value by analyzing error rates using loops and visualization.

2.MODEL TRAINING

Trains the KNN model with the optimal K value using knn.fit().

3.PREDICTION

Makes predictions on test data using knn.predict().

```python
# modeling process
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(x_train, y_train)
y_pred = knn.predict(x_test)
```

```python
# model evaluation
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       0.88      1.00      0.93         7
           2       1.00      1.00      1.00         6
           3       0.88      0.88      0.88         8
           4       0.88      0.88      0.88         8
           5       1.00      0.91      0.95        11

    accuracy                           0.94        48
   macro avg       0.94      0.94      0.94        48
weighted avg       0.94      0.94      0.94        48
```

MODEL EVALUATION

1. CLASSIFICATION REPORT

Displays precision, recall, f1-score, and accuracy using classification_report().

2.CONFUSION MATRIX

Visualizes model performance using confusion_matrix() and sns.heatmap().

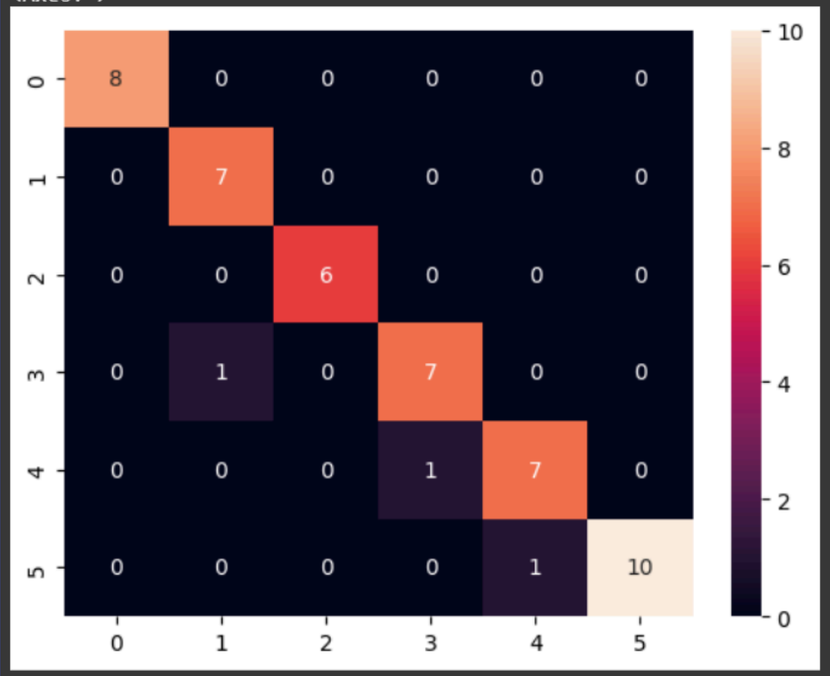
3.PREDICTION

Makes predictions on test data using knn.predict().

MODEL EVALUATION

```
# for calculating matrices and creating heatmaps for clearer visualization
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d')
```

```
<Axes: >
```




```
# to train the model and make predictions
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(criterion='entropy', random_state=42)

dt.fit(x_train, y_train)

y_pred_dt_entropy = dt.predict(x_test)

print(classification_report(y_test, y_pred_dt_entropy))
```

```
              precision    recall  f1-score   support

           0       1.00      0.88      0.93         8
           1       0.88      1.00      0.93         7
           2       1.00      1.00      1.00         6
           3       1.00      0.88      0.93         8
           4       0.89      1.00      0.94         8
           5       1.00      1.00      1.00        11

    accuracy                           0.96        48
   macro avg       0.96      0.96      0.96        48
weighted avg       0.96      0.96      0.96        48
```

HANDLING DATA IMBALANCE (OVERSAMPLING)

1. SMOTE

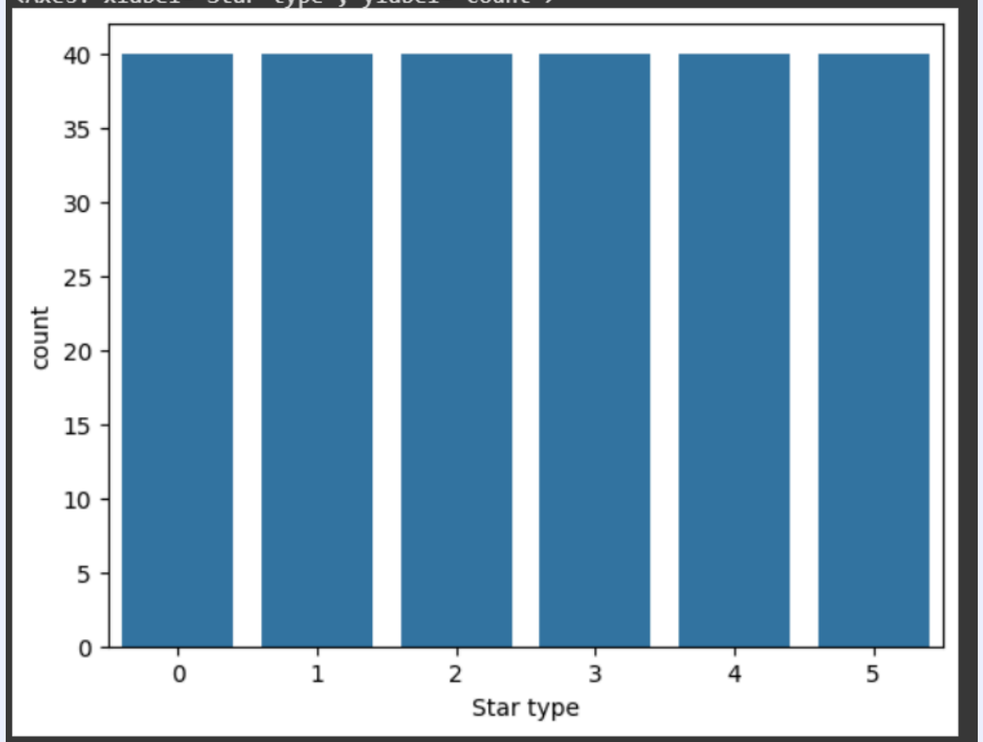
Applies the SMOTE technique to handle class imbalance using SMOTE().

2.MODEL EVALUATION

Retrains and reevaluates the KNN model using the oversampled dataset.

```
#oversampling
from imblearn.over_sampling import SMOTE

sm = SMOTE()
x_sm, y_sm = sm.fit_resample(X, y)

sns.countplot(x=y_sm)
```

```
<Axes: xlabel='Star type', ylabel='count'>
```



HANDLING DATA IMBALANCE (UNDERSAMPLING)

1. NEARMISS

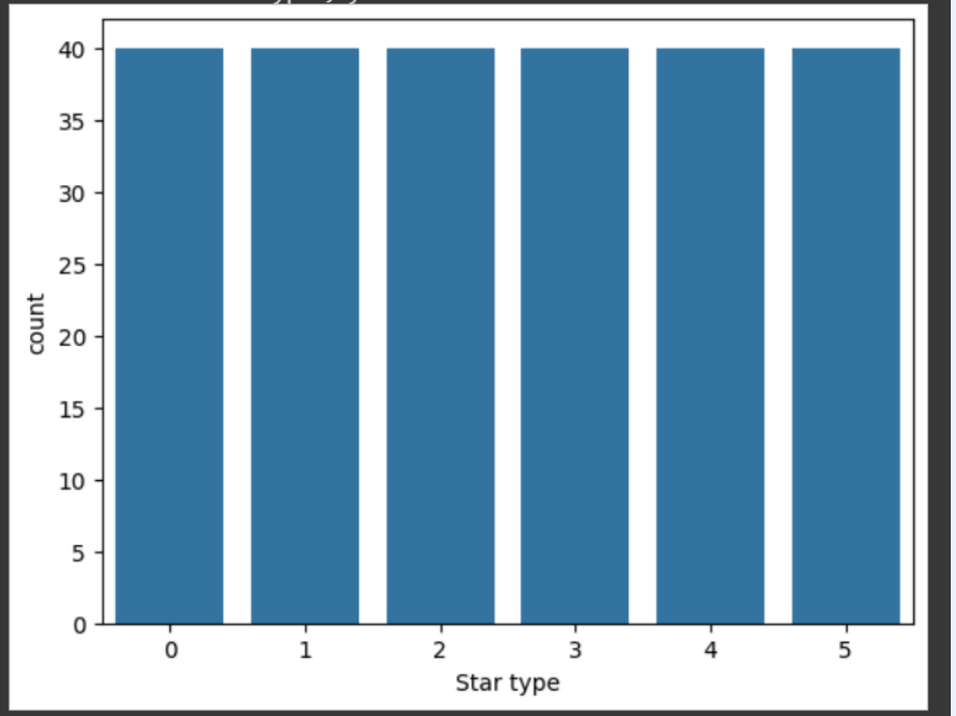
Applies the NearMiss undersampling technique using NearMiss().

2.MODEL EVALUATION

Retrains and reevaluates the KNN model using the undersampled dataset.

```
#Undersampling
from imblearn.under_sampling import NearMiss

nr = NearMiss()
x_nr, y_nr = nr.fit_resample(X, y)

sns.countplot(x=y_nr)
```

```
<Axes: xlabel='Star type', ylabel='count'>
```



DECISION TREE

1. MODELING WITH DECISION TREE

Trains a Decision Tree model using Gini and Entropy criteria.

2.MODEL EVALUATION

Evaluates the Decision Tree model using a classification report.

```
# to train the model and make predictions using 'DecisionTreeClassifier'
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(criterion='gini', random_state=42)
dt.fit(x_train, y_train)
y_pred_dt_gini = dt.predict(x_test)
print(classification_report(y_test, y_pred_dt_gini))
```

```
              precision    recall  f1-score   support

           0       1.00      0.88      0.93         8
           1       0.88      1.00      0.93         7
           2       1.00      1.00      1.00         6
           3       1.00      0.88      0.93         8
           4       0.89      1.00      0.94         8
           5       1.00      1.00      1.00        11

    accuracy                           0.96        48
   macro avg       0.96      0.96      0.96        48
weighted avg       0.96      0.96      0.96        48
```

CONCLUSION

Summarizes the outcomes of KNN modeling, including model performance and the best technique for handling data imbalance.

Provides recommendations for model improvement or future research directions.

THANK YOU