Université Mohammed V
Faculté des Sciences
Rabat

CODE DOCUMENTATION

Student Performance Prediction

REALISED BY : EL BATTAH Ahmed EALAOUI Oumaima
SPONSORISED BY : Mr Anouar Riad Solh

CLEAN DATA

Load the CSV file

```
import pandas as pd

# Replace 'nom_du_fichier.csv' with the actual name of your file
df = pd.read_csv('/content/drive/MyDrive/StudentPerformanceFactors.csv')

# Display the first 5 rows
print(df.head())
```

Check the structure of the dataset

```
# General information about the columns and their types
print(df.info())

# General statistics of numerical columns
print(df.describe())
```

Check for missing values

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

# Check uniquevalues of categorical variables
# List unique values in each categorical column

for col in df.select_dtypes(include=['object']).columns:
    print(f"{col}: {df[col].unique()}")
```

CLEAN DATA

Clean

```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# 1️⃣ Load the dataset from Google Drive
file_path = "/content/drive/MyDrive/StudentPerformanceFactors.csv"  # Correct path
df = pd.read_csv(file_path)

# 2️⃣ Remove duplicates
df.drop_duplicates(inplace=True)

# 3️⃣ Fill in missing values
# For numerical columns, fill with the mean
df = df.fillna(df.mean(numeric_only=True))

# For categorical columns, fill with the most frequent value (mode)
for col in df.select_dtypes(include=['object']).columns:
    df[col] = df[col].fillna(df[col].mode()[0])

# 4️⃣ Encode categorical variables
for col in df.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])

# 5️⃣ Handle outliers
def handle_outliers(df):
    numeric_cols = df.select_dtypes(include=['int64', 'float64']).columns
    for col in numeric_cols:
        # Calculate the first and third quartile, as well as the IQR (Interquartile Range)
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1

        # Calculate the bounds to identify outliers
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Replace outliers with the bounds (or use the median, adjust according to preference)
        if col == 'Tutoring_Sessions':  # Specific handling for the 'Tutoring_Sessions' column
            median_value = df[col].median()  # Calculate the median
            df[col] = df[col].apply(lambda x: median_value if x < lower_bound or x > upper_bound else x)
        else:
            df[col] = df[col].clip(lower=lower_bound, upper=upper_bound)

        # Identify and display outliers
        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
        print(f"🔍 Outliers in column '{col}': {len(outliers)} detected.")
    return df

df = handle_outliers(df)

# 6️⃣ Explicitly convert numerical columns to 'int' or 'float'
for col in df.select_dtypes(include=['int64', 'float64']).columns:
    try:
        # If the column is float64, round it and convert to int64
        if df[col].dtype == 'float64':
            df[col] = df[col].round().astype('int64')  # Round and convert to int64
        else:
            df[col] = pd.to_numeric(df[col], errors='raise')  # Explicit conversion
    except ValueError:
        print(f"Error converting column {col} to numeric.")

# 7️⃣ Save the cleaned dataset
cleaned_file_path = "/content/drive/MyDrive/Cleaned_student-performance.csv"
df.to_csv(cleaned_file_path, index=False)
print(f"✅ Cleaned dataset saved as: {cleaned_file_path}")
```

CLEAN DATA

Verification

```
# --- Check for duplicates, missing values, data types, and outliers ---

# Check for duplicates
def check_duplicates(df):
    duplicates = df[df.duplicated()]
    if not duplicates.empty:
        print(f"🔴 There are {len(duplicates)} duplicate rows in the dataset.")
    else:
        print("🟢 No duplicate rows found in the dataset.")

check_duplicates(df)

# Check for missing values
def check_missing_values(df):
    missing = df.isnull().sum()
    if missing.any():
        print("🔴 Missing values detected:")
        print(missing[missing > 0])
    else:
        print("🟢 No missing values detected in the dataset.")

check_missing_values(df)

# Check data types
def check_data_types(df):
    print("📝 Data types of the columns:")
    print(df.dtypes)
    # Check that all columns are of type 'int' or 'float' (as required)
    if df.dtypes.value_counts().get('int64', 0) == df.shape[1]:
        print("🟢 All columns are of type 'int64'.")
    else:
        print("🔴 Warning: Some columns are not of type 'int64'.")

check_data_types(df)

# Check for outliers in the dataset
def check_outliers(df):
    print("📊 Checking for outliers:")
    numeric_cols = df.select_dtypes(include=['int64']).columns
    for col in numeric_cols:
        q1 = df[col].quantile(0.25)
        q3 = df[col].quantile(0.75)
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr
        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
        print(f"🔍 Outliers in column '{col}': {len(outliers)} detected.")

check_outliers(df)
```

Preparing data for the model

Selection of features and target

```
# Separate independent variables (X) from the target variable (y)
X = df.drop(columns=['Exam_Score'])  # Independent variables
y = df['Exam_Score']                 # Target variable (student performance)
```

Split the data into training and testing

```
# Separate independent variables (X) from the target variable (y)
X = df.drop(columns=['Exam_Score'])  # Independent variables
y = df['Exam_Score']                 # Target variable (student performance)
```

Data Normalization

```
from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler on the training data and transform the training data
X_train_scaled = scaler.fit_transform(X_train)

# Transform the test data using the same scaler
X_test_scaled = scaler.transform(X_test)

# Verify the normalization by printing the first 5 rows of the normalized training data
print(f"Example of normalized data (X_train_scaled): \n{X_train_scaled[:5]}")
```

K-MEANS

K-means algorithme

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, silhouette_samples

# ========= Data Preparation =========
# We assume that X_train_scaled (normalized data) and y_train (target values) are already defined.
# Example if needed:
# scaler = StandardScaler()
# X_train_scaled = scaler.fit_transform(X_train)

# ========= Dimensionality Reduction with PCA =========
# Reduce the data to 2 dimensions to facilitate clustering and visualization
pca = PCA(n_components=2, random_state=42)
X_train_pca = pca.fit_transform(X_train_scaled)

# ========= Finding the Optimal Number of Clusters =========
silhouette_scores = []
k_values = range(2, 7)
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=50)
    kmeans.fit(X_train_pca)
    score = silhouette_score(X_train_pca, kmeans.labels_)
    silhouette_scores.append(score)
    print(f"k = {k} -> Silhouette Score = {score:.2f}")

plt.figure(figsize=(8,6))
plt.plot(list(k_values), silhouette_scores, marker='o', linestyle='-', color='blue')
plt.title("Silhouette Score for Different Numbers of Clusters")
plt.xlabel("Number of Clusters")
plt.ylabel("Silhouette Score")
plt.grid(True)
plt.show()

best_k = list(k_values)[np.argmax(silhouette_scores)]
print(f"The best number of clusters is: {best_k}")

# ========= Final Application of K-means with the Best k =========
kmeans_final = KMeans(n_clusters=best_k, random_state=42, n_init=50)
kmeans_final.fit(X_train_pca)
silhouette_final = silhouette_score(X_train_pca, kmeans_final.labels_)
print(f"Silhouette Score for {best_k} clusters: {silhouette_final:.2f}")

# ========= Visualization of the Clusters =========
plt.figure(figsize=(8,6))
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1],
            c=kmeans_final.labels_, cmap='viridis', alpha=0.7)
plt.title(f"K-means Clustering (k={best_k}) after PCA")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.colorbar(label='Cluster')
plt.show()

# ========= Visualization of the Centroids =========
centers_final = kmeans_final.cluster_centers_
plt.figure(figsize=(8,6))
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1],
            c=kmeans_final.labels_, cmap='viridis', alpha=0.6)
plt.scatter(centers_final[:, 0], centers_final[:, 1],
            c='red', marker='X', s=200, label='Centroids')
plt.title(f"Centroids of Clusters (k={best_k})")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.legend()
plt.show()

# ========= Adding Final Cluster Labels to DataFrame for Further Analysis =========
df_clustered_final = pd.DataFrame(X_train_pca, columns=['PC1', 'PC2'])
df_clustered_final['Cluster'] = kmeans_final.labels_
df_clustered_final['Exam_Score'] = y_train.values
print("Mean characteristics for each cluster (best_k):")
print(df_clustered_final.groupby('Cluster').mean())
```

Analyse data for the model

Transforme the Data

```
import pandas as pd

# Load your data
df = pd.read_csv("/content/drive/MyDrive/Cleaned_student-performance.csv")  # Replace with your CSV file
path

# Transform continuous variables into categorical ones
df['Étudie beaucoup'] = df['Hours_Studied'].apply(lambda x: 1 if x > 20 else 0)
df['Présence élevée'] = df['Attendance'].apply(lambda x: 1 if x > 80 else 0)
df['Bon élève'] = df['Previous_Scores'].apply(lambda x: 1 if x > 70 else 0)

# Encode categorical variables using one-hot encoding
df = pd.get_dummies(df, columns=['Motivation_Level', 'Parental_Involvement'])

# Drop unnecessary columns
df = df.drop(columns=['Hours_Studied', 'Attendance', 'Previous_Scores'])

# Display a preview of the transformed data
print(df.head())
```

APRIORI

APRIORI

```
from mlxtend.frequent_patterns import apriori, association_rules
import networkx as nx
import matplotlib.pyplot as plt

# --- Preparation for Apriori ---
# Convert all boolean columns to integers (0 and 1)
for col in df.columns:
    if df[col].dtype == 'bool':
        df[col] = df[col].astype(int)

# List of binary columns relevant for Apriori
binary_cols = [
    'Étudie beaucoup', 'Présence élevée', 'Bon élève',
    'Motivation_Level_0', 'Motivation_Level_1', 'Motivation_Level_2',
    'Parental_Involvement_0', 'Parental_Involvement_1', 'Parental_Involvement_2'
]

# Create the DataFrame for Apriori analysis
df_apriori = df[binary_cols]

# Convert the DataFrame to boolean type to improve performance and avoid warnings
df_apriori = df_apriori.astype(bool)

# --- Applying the Apriori Algorithm ---
# Generate frequent itemsets with a minimum support threshold
frequent_itemsets = apriori(df_apriori, min_support=0.05, use_colnames=True)

# Extract association rules using lift as the metric, with a minimum threshold of 1.0
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)

# Display the top 10 association rules sorted in descending order by lift
print("The top 10 association rules by lift:")
print(rules.sort_values(by='lift', ascending=False).head(10))

# --- Visualizing Association Rules as a Graph ---
# Create a directed graph to represent the rules
G = nx.DiGraph()
for idx, row in rules.iterrows():
    for antecedent in row['antecedents']:
        for consequent in row['consequents']:
            G.add_edge(antecedent, consequent, weight=row['lift'])

# Draw the graph
plt.figure(figsize=(10, 6))
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray',
        node_size=2000, font_size=10)
plt.title("Graph of Association Rules")
plt.show()

# --- Exporting the Association Rules ---
rules.to_csv("regles_association.csv", index=False)
print("Association rules have been exported to 'regles_association.csv'")
```

KNN

Creating a categorical target variable

```
import pandas as pd

# Load the cleaned dataset
df = pd.read_csv("/content/drive/MyDrive/Cleaned_student-performance.csv")

# Create a new column "Score_Category"
# 'High' if Exam_Score is greater than or equal to 67, otherwise 'Low'
df['Score_Category'] = df['Exam_Score'].apply(lambda x: 'High' if x >= 67 else 'Low')

# Display a preview of the Exam_Score and Score_Category columns
print(df[['Exam_Score', 'Score_Category']].head())
```

Preparing Data for KNN

```
from sklearn.model_selection import train_test_split

# Selecting relevant features for prediction.
# Adjust the selection based on your analysis.
features = df.drop(columns=['Exam_Score', 'Score_Category'])  # Independent variables
target = df['Score_Category']  # Target variable (High or Low category)

# Splitting the data into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.3, random_state=42)

# Displaying the size of the training and testing sets
print(f"Training set size: {X_train.shape}")
print(f"Testing set size: {X_test.shape}")
```

Data Normalisation

```
from sklearn.preprocessing import StandardScaler

scaler_knn = StandardScaler()
X_train_scaled = scaler_knn.fit_transform(X_train)
X_test_scaled = scaler_knn.transform(X_test)
```

Application of K-NN and evaluation

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Create the K-NN model with an initial number of neighbors (e.g., k=5)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_scaled, y_train)  # Train the model on the scaled training data

# Make predictions on the test set
y_pred = knn.predict(X_test_scaled)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy (K-NN): {accuracy:.2f}")

# Display the confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Display the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

Optimization of the k parameter

```python
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score

k_values = range(1, 21)
cv_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    # Using 5-fold cross-validation on the training set
    scores = cross_val_score(knn, X_train_scaled, y_train, cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())

# Display validation scores for each k
plt.figure(figsize=(8,6))
plt.plot(k_values, cv_scores, marker='o', linestyle='-', color='blue')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Mean Cross-Validation Score')
plt.title('Cross-Validation for K-NN')
plt.grid(True)
plt.show()

# Determine the best k with the highest cross-validation score
best_k = k_values[cv_scores.index(max(cv_scores))]
print(f"The optimal number of neighbors (k) is: {best_k}")
```

KNN

Retrain the K-NN model

```
# Retrain the K-NN model with the optimal k
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train_scaled, y_train)

# Evaluate the optimized model on the test set
y_pred_best = knn_best.predict(X_test_scaled)
best_accuracy = accuracy_score(y_test, y_pred_best)
print(f"Accuracy of the optimized K-NN model (k={best_k}): {best_accuracy:.2f}")
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Low', 'Medium', 'High'], yticklabels=
['Low', 'Medium', 'High'])
plt.title('Confusion Matrix - K-NN')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
print("Rapport de classification:")
print(classification_report(y_test, y_pred))
```