Student Performance Prediction

By Yassine Mouadi

Glossary of Paragraphs:

1. Introduction

1.1 The Predictive Model and the Dataset Used

The dataset utilized for this analysis contains key information about student academic behaviors and backgrounds, referred to as input features:

- **StudyTimeWeekly**: Represents the hours spent studying per week.
- **Absences**: Represents the number of absences from school.
- **ParentalEducation**: Represents the parent's education level.
- **Extracurricular**: Represents participation in extracurricular activities, encoded as binary (yes/no).

This study aims to develop a predictive model that accurately forecasts student performance, enabling educators to make informed decisions for timely interventions and optimal resource allocation. The primary goal is to predict the Grade Point Average (GPA) based on these input features, with GPA serving as a proxy for exam scores.

1.2 The Objectives of This Analysis

Library Imports

We import the necessary libraries for data processing, visualization, and model evaluation:

```
import numpy as np
import pandas as pd
import sys
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
import warnings
warnings.filterwarnings('ignore')
```

Dataset Loading

We load the `rabieelkharoua/students-performance-dataset` dataset, ensuring proper handling of errors:

```
file_path = "/home/yassine/.cache/kagglehub/datasets/rabieelkharoua/students-perf
try:
    data = pd.read_csv(file_path)
    print("Data loaded successfully.")
except Exception as e:
    print("An error occurred while loading the dataset: %s" % str(e))
```

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a critical step to analyze and summarize the dataset's main characteristics, identifying data quality issues.

First, we load the data using pandas:

```
data = pd.read_csv(file_path)
```

We display the first and last five rows to inspect the data:

```
display(data.head())
display(data.tail())
```

By default, five rows are displayed. More rows can be shown using `data.head(n)` or `data.tail(n)`.

We obtain a concise summary of the DataFrame with:

```
data.info()
```

This identifies missing values and data types. We then generate statistical summaries using:

```
data.describe()
```

This provides:

- **Count**: Number of non-null values.
- **Mean**: Average value.
- **Std**: Standard deviation.
- **Min/Max**: Range of values.
- **25%, 50%, 75%**: Quartiles.

Executed as:

```
display(data.head())
display(data.tail())
data.info()
display(data.describe())
```

Handling Missing and Duplicate Values

We check for missing values and duplicates:

```
data.isnull().sum()
data.duplicated().sum()
```

Missing values are filled with the mean of their respective columns, as the mean is optimal for linear regression under the mean squared error cost function, preserving the average value and data trends. Infinite values are replaced with NaN and filled similarly. Duplicates are removed:

```
data.fillna(data.mean(numeric_only=True), inplace=True)
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.fillna(data.mean(numeric_only=True), inplace=True)
data.drop_duplicates(inplace=True)
```

Univariate Analysis

Univariate analysis examines each feature independently to understand its behavior.

Numerical Features:

We define an array of numerical features: `StudyTimeWeekly`, `Absences`, `ParentalEducation`, `GPA`. We create a figure for histograms:

```
plt.figure(figsize=(8, 16))
```

This creates a figure with width=8 and height=16. We loop through the features to create histograms:

```
plt.subplot(4, 1, i + 1)
plt.hist(data[feature], bins=20, edgecolor='black')
plt.xlabel(feature)
plt.ylabel('Frequency')
```

`plt.subplot(4, 1, i + 1)` divides the figure into 4 rows and 1 column, with `i + 1` selecting the current slot. `plt.hist(data[feature], bins=20, edgecolor='black')` plots a histogram with 20 bins and black-edged bars.

Categorical Features:

Features like `Extracurricular` and `ParentalEducation` are categorical despite numerical encoding. We plot bar charts in a figure of width=14 and height=7:

```
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, i + 1)
counts = data[feature].value_counts()
plt.bar(counts.index, counts.values, color=['blue', 'red'])
```

`counts = data[feature].value_counts()` counts category occurrences, with `counts.index` as categories and `counts.values` as counts.

Bivariate Analysis

This explores relationships between features and GPA using a 2x2 grid of scatter plots:

```
plt.figure(figsize=(15, 10))
i = 0
for feature in selected_features:
    i += 1
    plt.subplot(2, 2, i)
    plt.scatter(X_train_raw[feature], Y_train, alpha=0.5)
    plt.title('GPA vs. %s' % feature)
    plt.xlabel(feature)
    plt.ylabel('GPA')
plt.tight_layout()
plt.show()
```

Points are semi-transparent (`alpha=0.5`) to visualize dense clusters.

Data Standardization

We use `StandardScaler` from `sklearn.preprocessing` to standardize data, ensuring features with different scales (e.g., `StudyTimeWeekly` [0-20] vs. `ParentalEducation` [0-4]) are treated fairly:

fit(): Calculates the mean and standard deviation.

transform(): Standardizes data using $z = \frac{x - \text{mean}}{\text{std}}$.

Formulas:

- Mean: $E(X) = \frac{1}{n} \sum x_i$
- Standard Deviation: $\sigma(X) = \sqrt{\frac{1}{n} \sum (x_i - E(X))^2}$
- Standardization: $z = \frac{x - E(X)}{\sigma(X)}$
- Inverse Transformation: $x = z \cdot \sigma(X) + E(X)$

Data Preparation

We select features for predicting GPA, separate GPA as the target, and split data into training and testing sets to prevent overfitting. `selected_features` defines predictor columns, `X_raw` contains these columns, and GPA is extracted as a NumPy array.

Linear Model

We define a `predict` function for the linear model:

```
def predict(w, b, x):
    return np.dot(w, x) + b
```

This computes $\text{prediction} = \mathbf{w} \cdot \mathbf{x} + b$, using the dot product for weights and features.

Cost Function

The cost function $J(w, b)$ measures prediction error:

$$J(w, b) = \frac{1}{2m} \sum_{i=1}^{m} \left[(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) - y^{(i)} \right]^2$$

Where:

- $\mathbf{w}$: Weight vector ($w_1, w_2, \ldots, w_n$).
- $\mathbf{x}^{(i)}$: Feature vector of the i-th example.
- b: Bias term.
- m: Number of training examples.
- $\mathbf{w} \cdot \mathbf{x}^{(i)}$: Dot product ($w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$).

Implemented as:

```
def cost_function(X, y, w, b):
    m = len(X)
    cost_sum = 0.0
    for i in range(m):
        predictions = predict(w, b, X[i])
        y_i = y[i]
        cost_sum += np.power((predictions - y_i), 2)
    total_cost = (1 / (2 * m)) * cost_sum
    return total_cost
```

Gradient Descent

Gradient descent minimizes the cost function by updating weights and bias:

$$w_j = w_j - \alpha \frac{\partial J(w, b)}{\partial w_j}$$

$$b = b - \alpha \frac{\partial J(w, b)}{\partial b}$$

Where:

- α: Learning rate (e.g., 0.01).
- $\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left[(f_{w,b}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \right]$.
- $\frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left[f_{w,b}(\mathbf{x}^{(i)}) - y^{(i)} \right]$.

Implemented as:

```
def gradient_descent(X, y, learning_rate, iterations):
    n_features = X.shape[1]
    w = np.zeros(n_features)
    b = 0.0
    m = len(X)
    cost_history = []
    for i in range(iterations):
        f_wb_predictions = np.zeros(m)
        for j in range(m):
            f_wb_predictions[j] = predict(w, b, X[j])
        dj_dw = np.zeros(n_features)
        dj_db_sum = 0.0
        for j in range(m):
            for k in range(n_features):
                dj_dw[k] += (f_wb_predictions[j] - y[j]) * X[j][k]
            dj_db_sum += (f_wb_predictions[j] - y[j])
        dw = (1.0 / m) * dj_dw
        db = (1.0 / m) * dj_db_sum
        w = w - learning_rate * dw
        b = b - learning_rate * db
        cost = cost_function(X, y, w, b)
        cost_history.append(cost)
    return w, b, cost_history
```

Model Training

We train the model using gradient descent with scaled data:

```
learning_rate = 0.01
iterations = 1000
print("Phase 1: Gradient Descent initiated for model training:")
w_learned, b_learned, cost_history = gradient_descent(X, Y, learning_rate, iterat
```

The `cost_history` list records the value of the cost function at each iteration.

Model Evaluation

We evaluate the model using Mean Squared Error (MSE) and R-squared score on training and test data to assess performance and check for overfitting. A lower MSE indicates a more accurate model, while the R-squared score measures how well the model explains the variance in GPA.

Performance Metrics

We calculate MSE and R-squared scores for both training and test sets:

```
print("\nModel Evaluation:")
y_train_pred = predict(X_train_scaled, w_learned, b_learned)
y_test_pred = predict(X_test_scaled, w_learned, b_learned)

train_mse = mean_squared_error(Y_train, y_train_pred)
train_r2 = r2_score(Y_train, y_train_pred)

test_mse = mean_squared_error(Y_test, y_test_pred)
test_r2 = r2_score(Y_test, y_test_pred)

print("Training set MSE: %s" % str(round(train_mse, 4)))
print("Testing set MSE: %s" % str(round(test_mse, 4)))
print("Training set R^2 Score: %s" % str(round(train_r2, 4)))
print("Testing set R^2 Score: %s" % str(round(test_r2, 4)))
```

Prediction Clipping

Predictions are clipped to ensure GPA values are within the logical range [0.0, 4.0]:

```
y_test_pred_clipped = np.clip(y_test_pred, 0.0, 4.0)

clipped_mse = mean_squared_error(Y_test, y_test_pred_clipped)
clipped_r2 = r2_score(Y_test, y_test_pred_clipped)

print("Test MSE after clipping predictions: %s" % str(round(clipped_mse, 4)))
print("Test R^2 after clipping predictions: %s" % str(round(clipped_r2, 4)))
```

> **Visualizing Predictions**

We visualize the relationship between actual and predicted GPA values on the test set:

```
print("\nVisualizing Predictions:")
plt.figure(figsize=(10, 6))
plt.scatter(Y_test, y_test_pred_clipped, alpha=0.7)
plt.plot([Y_test.min(), Y_test.max()], [Y_test.min(), Y_test.max()], 'r--', lw=2)
plt.title('Actual vs. Predicted GPA (Test Set)')
plt.xlabel('Actual GPA')
plt.ylabel('Predicted GPA')
plt.grid(True)
plt.show()
```

Thanks for Reading

This employs predictive modeling methodologies to inform data-driven decision-making in educational contexts. I gratefully acknowledge the time and consideration of readers engaging with this study. For further discussion or collaborative opportunities, correspondence may be directed to me in my LinkedIn.