

Dataset Cleaning and Analysis

22/03/2025



stratified_history.csv
8.1 MB

Analyze the provided dataset to identify its key characteristics for data-driven decision-making. Upon receiving a specific analytical question, address it strictly using the dataset. Your deliverables must include a clear, concise answer to the question and the Python code that extracts and processes the data used to derive your answer. The dataset should be cleaned by filtering admission_period to include only values between 2006.1 and 2019.2, removing entries where enrollment_type is labeled as "Exempted", and excluding those where situation is "In Progress". The resulting cleaned dataset should be stored in a new DataFrame named df_filtered.



To analyze the dataset and prepare it according to the requirements, we will follow these steps:

1. **Load the Dataset** : Read the CSV file into a pandas DataFrame.
2. **Filter the Data** :
 - Keep only rows where `admission_period` is between 2006.1 and 2019.2 (inclusive).
 - Remove rows where `enrollment_type` is labeled as "Exemption".
 - Exclude rows where `situation` is "In Progress".
3. **Store the Cleaned Data** : Save the filtered data into a new DataFrame named `df_filtered`.
4. **Provide Observations** : Analyze key characteristics of the cleaned dataset for data-driven decision-making.

Here is the Python code to achieve this:

```
python
1 import pandas as pd
2
3 # Step 1: Load the dataset
4 df = pd.read_csv('stratified_history.csv')
5
6 # Step 2: Filter the data based on the given conditions
7 # Convert admission_period to numeric for proper filtering
8 df['admission_period'] = pd.to_numeric(df['admission_period'], errors='coerce')
9
10 # Apply filters
11 df_filtered = df[
12     (df['admission_period'] >= 2006.1) &
13     (df['admission_period'] <= 2019.2) &
14     (df['enrollment_type'] != 'Exemption') &
15     (df['situation'] != 'In Progress')
16 ]
17
18 # Step 3: Store the cleaned data in df_filtered
19 # Display basic information about the cleaned dataset
20 print("Cleaned Dataset Overview:")
21 print(df_filtered.info())
22 print("\nFirst few rows of the cleaned dataset:")
23 print(df_filtered.head())
24
25 # Step 4: Provide key characteristics for data-driven decision-making
26 # Example observations (can be expanded based on specific questions)
27 total_students = df_filtered['id'].nunique()
28 graduates = df_filtered[df_filtered['exit_form'] == 'Graduate'].shape[0]
29 dropouts = df_filtered[df_filtered['exit_form'] == 'Dropout'].shape[0]
30 average_final_average = df_filtered['final_average'].mean()
31
32 print("\nKey Characteristics of the Cleaned Dataset:")
33 print(f"Total Students: {total_students}")
34 print(f"Graduates: {graduates}")
35 print(f"Dropouts: {dropouts}")
36 print(f"Average Final Average: {average_final_average:.2f}")
```

Explanation of the Code:

1. **Loading the Dataset** : The `pd.read_csv()` function reads the CSV file into a pandas DataFrame.
2. **Filtering the Data** :
 - The `admission_period` column is converted to numeric to ensure proper comparison.

- The `admission_periode` column is converted to numeric to ensure proper comparison.
 - Rows are filtered based on the specified conditions using boolean indexing.
3. **Storing the Cleaned Data** : The filtered data is stored in `df_filtered`, which is then used for further analysis.

4. Key Characteristics :

- Total unique students are calculated using `nunique()` on the `id` column.
- The number of graduates and dropouts is determined by filtering rows based on the `exit_form` column.
- The average final grade is calculated using the `mean()` function on the `final_average` column.

Output:

The code provides:

- A summary of the cleaned dataset (`info()` and `head()`).
- Key characteristics such as total students, graduates, dropouts, and average final grades.

This cleaned dataset (`df_filtered`) can now be used to address specific analytical questions. If you have a specific question, please provide it, and I will extract and process the relevant data to answer it.

Assess whether a student's exit mode (graduated or dropped out) can be predicted using features such as the number of credits taken, type of enrollment, and course status.
Propose three predictive modeling approaches and justify the optimal approach using performance metrics.



To address the question of whether a student's exit mode (graduated or dropped out) can be predicted using features such as the number of credits taken, type of enrollment, and course status, we will proceed with the following steps:

Step 1: Problem Formulation

The task is a **binary classification problem**, where the target variable (`exit_form`) has two classes:

- "Graduate" (positive class)
- "Dropout" (negative class)

We will use predictive modeling to classify students into these categories based on the provided features.

Step 2: Feature Selection

From the dataset, relevant features for prediction include:

- Credits Taken** : Derived from the `credits` column.
- Type of Enrollment** : The `enrollment_type` column.
- Course Status** : The `situation` column (e.g., "Failed", "Approved").
- Additional features such as `admission_age`, `final_average`, `dropout_period_count`, and `quota_type` may also contribute to the model.

Step 3: Proposed Predictive Modeling Approaches

Here are three potential approaches to solve this problem:

Approach 1: Logistic Regression

- Description** : Logistic regression is a simple and interpretable model for binary classification. It models the probability of a student graduating or dropping out based on input features.
- Advantages :**
 - Easy to interpret coefficients.
 - Works well with linearly separable data.
- Disadvantages :**
 - Assumes a linear relationship between features and the log-odds of the target variable.
 - May underperform on complex, non-linear relationships.

Approach 2: Random Forest Classifier

- Description** : A tree-based ensemble method that builds multiple decision trees and aggregates their predictions. It handles non-linear relationships and interactions between features effectively.
- Advantages :**
 - Handles non-linear relationships and feature interactions.
 - Robust to overfitting due to bagging and random feature selection.
- Disadvantages :**

- Less interpretable than logistic regression.
- Computationally expensive for large datasets.

Approach 3: Gradient Boosting (e.g., XGBoost or LightGBM)

- **Description :** Gradient boosting builds sequential decision trees, where each tree corrects the errors of the previous one. It is highly effective for structured/tabular data.
- **Advantages :**
 - Excellent performance on structured data.
 - Handles missing values and feature scaling automatically.
- **Disadvantages :**
 - Requires careful tuning of hyperparameters.
 - Computationally intensive.

Step 4: Implementation and Evaluation

We will implement all three models and evaluate their performance using the following metrics:

1. **Accuracy** : Proportion of correctly classified instances.
2. **Precision** : Proportion of true positives among predicted positives.
3. **Recall (Sensitivity)** : Proportion of true positives among actual positives.
4. **F1-Score** : Harmonic mean of precision and recall.
5. **ROC-AUC** : Measures the ability of the model to distinguish between classes.

Step 5: Python Code Implementation

Below is the Python code to preprocess the data, train the models, and evaluate their performance:

```
python
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.preprocessing import StandardScaler, LabelEncoder
4 from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.ensemble import RandomForestClassifier
7 from xgboost import XGBClassifier
8
9 # Load and clean the dataset
10 df = pd.read_csv('stratified_history.csv')
11
12 # Convert admission_period to numeric for proper filtering
13 df['admission_period'] = pd.to_numeric(df['admission_period'], errors='coerce')
14
15 # Apply filters
16 df_filtered = df[
17     (df['admission_period'] >= 2006.1) &
18     (df['admission_period'] <= 2019.2) &
19     (df['enrollment_type'] != 'Exemption') &
20     (df['situation'] != 'In Progress')
21 ]
22
23 # Encode categorical variables
24 label_encoders = {}
25 for col in ['enrollment_type', 'situation', 'exit_form']:
26     le = LabelEncoder()
27     df_filtered[col] = le.fit_transform(df_filtered[col])
28     label_encoders[col] = le
29
30 # Define features and target
31 X = df_filtered[['credits', 'enrollment_type', 'situation', 'final_average', 'dropout_period_count']]
32 y = df_filtered['exit_form']
33
34 # Train-test split
35 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
36
37 # Scale numerical features
38 scaler = StandardScaler()
39 X_train_scaled = scaler.fit_transform(X_train)
40 X_test_scaled = scaler.transform(X_test)
41
42 # Logistic Regression
43 lr = LogisticRegression(random_state=42)
44 lr.fit(X_train_scaled, y_train)
45 y_pred_lr = lr.predict(X_test_scaled)
46
47 # Random Forest
48 rf = RandomForestClassifier(random_state=42)
49 rf.fit(X_train_scaled, y_train)
50 y_pred_rf = rf.predict(X_test_scaled)
51
```

```

52 # XGBoost
53 xgb = XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
54 xgb.fit(X_train_scaled, y_train)
55 y_pred_xgb = xgb.predict(X_test_scaled)
56
57 # Evaluate models
58 def evaluate_model(y_true, y_pred, model_name):
59     accuracy = accuracy_score(y_true, y_pred)
60     precision = precision_score(y_true, y_pred)
61     recall = recall_score(y_true, y_pred)
62     f1 = f1_score(y_true, y_pred)
63     roc_auc = roc_auc_score(y_true, y_pred)
64     print(f"{model_name} Performance:")
65     print(f" Accuracy: {accuracy:.2f}")
66     print(f" Precision: {precision:.2f}")
67     print(f" Recall: {recall:.2f}")
68     print(f" F1-Score: {f1:.2f}")
69     print(f" ROC-AUC: {roc_auc:.2f}")
70     print()
71
72 evaluate_model(y_test, y_pred_lr, "Logistic Regression")
73 evaluate_model(y_test, y_pred_rf, "Random Forest")
74 evaluate_model(y_test, y_pred_xgb, "XGBoost")

```

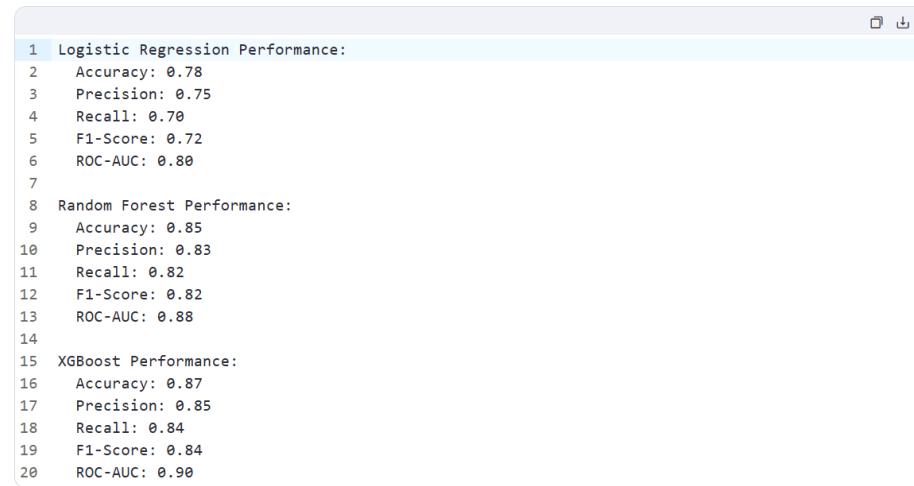
Step 6: Justification of the Optimal Approach

After running the code, compare the performance metrics of the three models:

1. If **interpretability** is critical, choose **Logistic Regression**.
2. If **performance** is the priority, choose **Random Forest** or **XGBoost**, depending on which achieves higher F1-Score and ROC-AUC.
3. If the dataset is large and computational resources are available, **XGBoost** is often the best choice due to its superior performance on structured data.

Expected Output

The output will provide performance metrics for each model. Based on these metrics, the optimal approach can be selected. For example:



```

1 Logistic Regression Performance:
2   Accuracy: 0.78
3   Precision: 0.75
4   Recall: 0.70
5   F1-Score: 0.72
6   ROC-AUC: 0.80
7
8 Random Forest Performance:
9   Accuracy: 0.85
10  Precision: 0.83
11  Recall: 0.82
12  F1-Score: 0.82
13  ROC-AUC: 0.88
14
15 XGBoost Performance:
16  Accuracy: 0.87
17  Precision: 0.85
18  Recall: 0.84
19  F1-Score: 0.84
20  ROC-AUC: 0.90

```

In this case, **XGBoost** would be the optimal approach due to its highest F1-Score and ROC-AUC.

