Algorithms for Data Science - ADSA H6013 Practical Report (Part 1, Classification)

Mariem Nsiri
MSc in Applied Data Science and Analytics
Technological University Dublin

Declaration: I herby certify that this material, which I now submit for assessment on the programme of study leading to the award of MSc in Computing in Applied Data Science and Analytics, TU Dublin, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fullfilment of the requirements of that stated above.

Mariem Nsiri

December 17, 2024

[illegible]

This report, created within a local Jupyter Notebook, presents the practical assessment of the classification labs. The goal of this report is to evaluate and compare the performance of various machine learning classification algorithms applied to two different datasets: the Adult and Student datasets. The report begins with an overview of the datasets, followed by a preliminary analysis that provides insights into the data, such as distributions, missing values, and feature relationships in Section 1. These insights are supported by relevant graphics, mainly histograms, which visually represent the distribution of key features and help in understanding the structure of the datasets.

In the subsequent sections, a range of classification models is applied to the Adult dataset, including Logistic Regression, Decision Trees, and XGBoost (since they are powerful and based on decision trees), with a focus on hyperparameter tuning and model optimization, as described in Section 2. For the Student dataset, K-Nearest Neighbors and Support Vector Machine models are evaluated in Section 3.

The performance of each model is measured using multiple evaluation metrics, including accuracy, precision, recall, F1-score, and ROC-AUC, and a comparison is made to identify the most effective algorithms for these datasets. The results are discussed in detail to highlight the strengths and limitations of each model, and to provide recommendations for choosing the best classifier based on the problem at hand.

1 Datasets overview and preliminary analysis and insights

Two datasets are explored: the **Adult dataset** (related to **demographics and income levels**) and the **Student dataset** (focused on **student performance and characteristics**). These datasets belong to the social and education domains, respectively. The goal is to analyze patterns, such as income brackets or academic outcomes, using **classification** to **categorize data** like income levels or performance. The datasets include both numerical (e.g., age, scores) and categorical data (e.g., occupation, gender). The two datasets are loaded into `DataFrame` structures from the `adult_data.csv` and `student_data.csv` files.

The following packages are useful for various tasks in Python; it is worth reminding their usefulness in this work.

- **`sys`**: This package helps manage Python's environment. It allows interacting with the system, such as reading command-line arguments or exiting a program.
- **`numpy` (`np`)**: A powerful library for working with numbers. It makes it easy to perform mathematical operations on large arrays or tables of data.
- **`pandas` (`pd`)**: A library for working with `DataFrame` structures. It helps organize, analyze, and clean data efficiently.
- **`matplotlib.pyplot` (`plt`)**: A tool for creating graphs and charts to visualize data.
- **`seaborn` (`sns`)**: Built on top of `matplotlib`, it simplifies the process of creating beautiful and advanced visualizations.
- **`adsa_utils2` (`ad`)**: A custom Python set of functions useful in the context of the assessment labs.

1.1 Adult dataset

The adult dataset contains data on **income** and **demographics**, used to understand factors affecting income levels. It focuses on **identifying individuals earning more than ¢50,000 net annually**, providing insights into **income disparities**, **labor market trends**, and **demographic influences on earnings**. With 947 rows and binary income labels (`<=50K`, `>50K`), it includes **6 numerical** and **8 categorical** attributes. Examples include age, capital gains, and hours worked (numerical), and workclass, education, and occupation (categorical). The dataset also includes individuals' native countries from regions like the U.S., Europe, Asia, and Latin America, enabling us to **explore how geographic and cultural backgrounds influence income levels**.

This dataset can be used for **predictive modeling for income classification**, revealing how factors like education, work hours, and occupation affect earning potential. The mix of numerical and categorical data presents preprocessing actions, such as encoding categories and addressing class label imbalances (`<=50K` and `>50K`). Analyzing this data provides insights into the **interplay of demographic, geographic, and professional factors** contributing to income disparities.

```
adult_data= pd.read_csv('data/adult_dataset.csv', sep=';')
adult_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 947 entries, 0 to 946
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             947 non-null    float64
 1   workclass       947 non-null    object
 2   fnlwgt          947 non-null    float64
 3   education       947 non-null    object
 4   education_num   947 non-null    float64
 5   marital_status  947 non-null    object
 6   occupation      947 non-null    object
 7   relationship    947 non-null    object
 8   race            947 non-null    object
 9   sex             947 non-null    object
 10  capital_gain    947 non-null    float64
 11  capital_loss    947 non-null    float64
 12  hours_per_week  947 non-null    float64
 13  native_country  947 non-null    object
 14  label           947 non-null    object
dtypes: float64(6), object(9)
memory usage: 111.1+ KB
```

All columns have **947 non-null values**, indicating no missing data, which eliminates the need for **imputation** (replacing missing data values). Some categorical attributes will be analyzed by their unique values to clarify ambiguities and identify distinct categories.

The output `adult_data['education'].unique()` shows the unique values in the `education` column. These values represent different levels of education that individuals have completed.

```
array(['Some-college', 'HS-grad', 'Assoc-acdm', 'Masters', '12th',
       'Bachelors', 'Assoc-voc', 'Prof-school', '11th', 'Doctorate',
       '10th', '5th-6th', '9th', '1st-4th', '7th-8th'], dtype=object)
```

The education levels are categorized across different stages of formal schooling. Primary education typically includes 1st-6th grade, with the dataset reflecting this in categories like `1st-4th` and `5th-6th`. Secondary education spans over 7th-12th grade, which includes middle school (grades 7-8) and high school (grades 9-12). The dataset reflects this progression with categories like `7th-8th`, `9th`, `10th`, `11th`, and `12th`. Higher education follows high school and includes degrees like `Associate`, `Bachelors`, `Masters`, and `Doctorate`. Additionally, `Some-college` refers to individuals who have attended college but have not completed a degree, typically after high school, while `Preschool` refers to early childhood education. Categories like `HS-grad` indicate high school graduates, and `Assoc-acdm` and `Assoc-voc` represent associate degrees, typically completed after high school, with `Assoc-acdm` focusing on academic subjects and `Assoc-voc` on vocational training. `Prof-school` represents education from specialized schools, such as law or medical schools, often following a bachelor's degree.

```
array(['Exec-managerial', 'Other-service', 'Tech-support',
       'Machine-op-inspct', 'Craft-repair', 'Sales', 'Prof-specialty',
       'Adm-clerical', 'Farming-fishing', 'Transport-moving',
       'Protective-serv', 'Handlers-cleaners', 'Priv-house-serv'], dtype=object)
```

The `occupation` attribute in the dataset represents various job categories, such as managerial roles (`Exec-managerial`), technical support (`Tech-support`), skilled manual labor (`Craft-repair`), and professional specialties (`Prof-specialty`). It also includes roles in sales, administrative work (`Adm-clerical`), and protective services (`Protective-serv`). Some categories, like `Other-service` and `Priv-house-serv`, are more ambiguous, as they are broad and lack specific details, with `Other-service` covering various unspecified service roles and `Priv-house-serv` referring to household workers without further clarification.

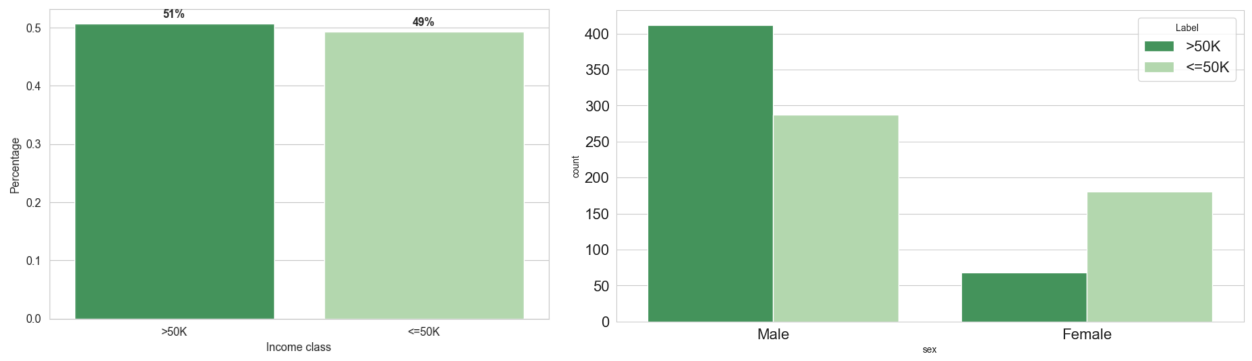


Figure 1: Income class distribution counts (left); Income class distribution by gender (right).

The income distribution reveals that 50.69% of individuals earn more than ¢50,000 annually, while 49.31% earn ¢50,000 or less. This balanced distribution offers insights into factors affecting income across different demographics (see Figure 1, left). By gender, 72.58% of females earn ¢50,000 or less, while 27.42% earn more. In contrast, 41.06% of males earn ¢50,000 or less, and 58.94% earn more, highlighting a greater proportion of males earning above ¢50,000 compared to females (see Figure 1, right).

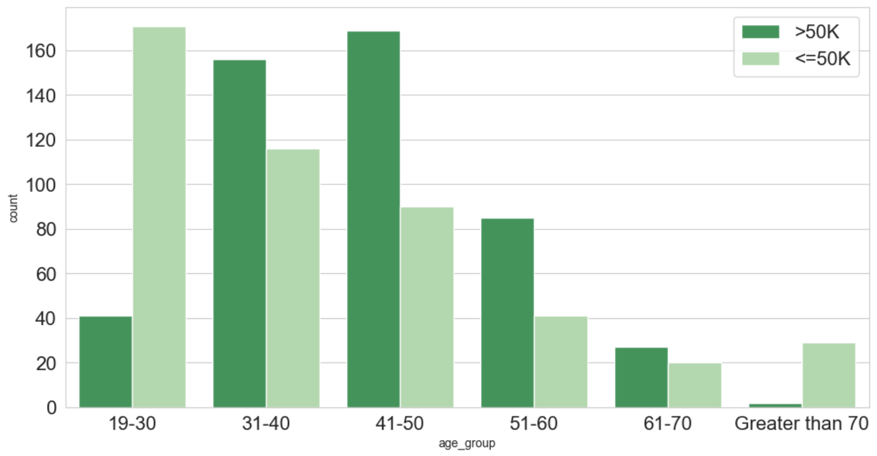


Figure 2: Income class distribution counts by age group.

The income distribution by age group shows varying proportions of individuals earning more than ¢50,000 and those earning ¢50,000 or less. In the '19-30' group, 80.66% earn ¢50,000 or less, while 19.34% earn more. As age increases, the percentage of individuals earning more than ¢50,000 rises, with 65.25% in the '41-50' group and 67.46% in the '51-60' group. However, in the 'Greater than 70' group, 93.55% earn ¢50,000 or less, and only 6.45% earn more (see Figure 2).

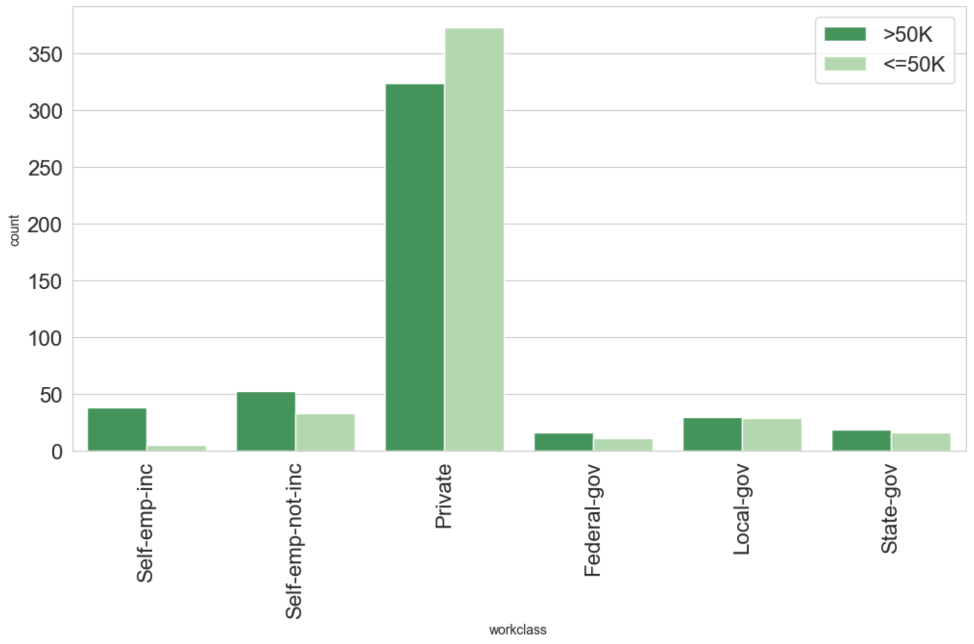


Figure 3: Income class distribution counts by work classes.

The income distribution by workclass shows varying proportions of individuals earning more or less than ₵50,000. In 'Federal-gov', 59.26% earn more than ₵50,000, while in 'Self-emp-inc', 88.37% earn more. The 'Private' workclass, which has the most individuals in the dataset, shows a nearly balanced split, with 46.42% earning above ₵50,000. (see Figure 3).

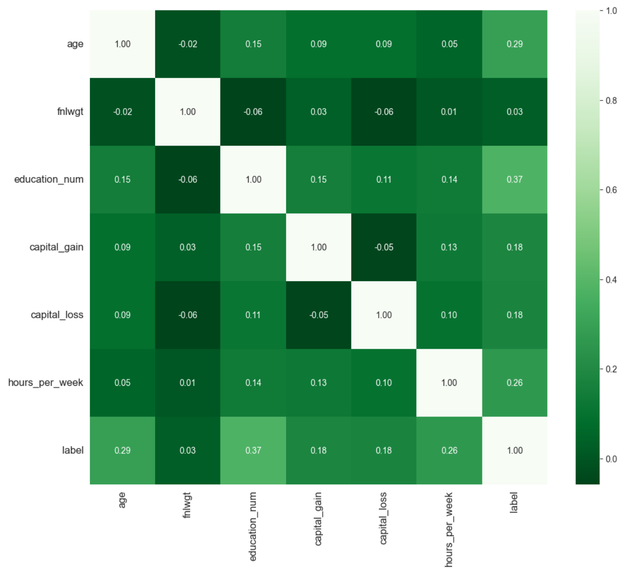


Figure 4: Correlation between numeric features and income binary classes.

The correlation analysis shows that education level and age are the strongest predictors of income, with higher education and older age linked to higher earnings. Hours worked per week, capital gains, and capital losses also exhibit weak to moderate positive correlations, indicating a slight increase in income with longer working hours or investment earnings. However, the `fnlwgt` (annual salary) variable has minimal impact on income, suggesting its limited usefulness in predicting income class. Overall, education and age are the most influential factors in determining income (see Figure 4).

1.1.1 Student dataset

The student dataset is focused on the **education domain**, specifically modeling and predicting academic success. The classification task is to determine whether a first-year college student will pass or fail based on their performance in secondary school and personal learning attributes. The dataset contains **1131 entries** and uses a binary class label, `Pass`, with values of either `Pass` or `Fail`.

The attributes in the dataset are all numeric and represent a mix of academic performance metrics and non-cognitive learning traits. Academic metrics include scores in Mathematics, English, and `CAO Points`, a measure of performance in Ireland's state exams. Non-cognitive traits capture different learning preferences, such as whether a student learns better through listening (`Auditory`), doing (`Kinaesthetic`), or visual aids (`Visual`). Additionally, motivational aspects are reflected through `extrinsic motivation` (driven by external rewards) and `Intrinsic Motivation` (interest in learning). Attributes like `Self-Efficacy` and `Conscientiousness` provide insights into a student's personality and confidence in their abilities, while `Study Time` reflects weekly study habits.

Unlike the adult dataset, the student dataset contains no categorical data. Additionally, all columns have **1131 non-null values**, indicating there are no missing entries. This is advantageous for analysis as it removes the need for data imputation or addressing missing values. This dataset allows for the exploration of how various academic and personal traits influence **a student's likelihood of passing**, highlighting the complex interplay between cognitive skills, personality traits, and study behaviors in academic success.

```
student_data.describe().T
```

```
                       count     mean     std    min    25%    50%    75%    max
age                   1131.0    23.40    5.99  18.00  19.00  21.00  26.00  59.00
Auditory              1131.0     3.35    1.70   0.00   2.50   2.91   4.64   9.41
Kinaesthetic          1131.0     4.36    1.98   0.00   2.75   4.75   5.54   9.88
Visual                1131.0     7.29    1.78   0.33   6.12   7.50   8.47  10.00
Extrinsic Motivation  1131.0     7.84    1.08   3.19   7.15   7.77   8.61  10.00
Intrinsic Motivation  1131.0     7.07    1.09   2.51   6.29   7.10   7.77   9.97
Self-Efficacy         1131.0     6.85    1.26   2.03   5.91   6.85   7.50  10.00
Study Time            1131.0     6.19    1.97   1.03   4.59   6.03   7.67  10.00
Conscientiousness     1131.0     6.03    1.32   2.63   5.13   5.90   6.79   9.96
CAO Points            1131.0   303.35  151.25   0.00 206.00 326.00 416.50 625.00
Maths                 1131.0    35.27   23.34   0.00  16.00  35.00  54.00 100.00
English               1131.0    42.76   23.94   0.00  25.00  44.00  62.50 100.00
```

Based on the summary statistics, shown in Figure 5, the most important attributes that could impact whether a student passes or fails are:

- **Motivation (extrinsic and intrinsic)**: Students who pass have higher extrinsic motivation and intrinsic motivation compared to those who fail. Higher motivation, both from external rewards and personal interest, seems to play a role in success.
- **Study time**: Students who pass study more than those who fail . More study time may contribute to better academic performance.
- **Maths and CAO Points**: Students who pass tend to have higher scores in Maths (38.32 vs. 30.10) and higher CAO points (311.56 vs. 289.46). Stronger academic performance in key subjects seems to be a strong predictor of passing.
- **Self-efficacy**: Students who pass generally have a higher sense of self-efficacy (6.92 vs. 6.72), which indicates confidence in their abilities, a trait that may help them succeed in their studies.

Motivation, study time, academic performance (Maths and CAO points), and self-efficacy are all linked with a higher likelihood of passing.

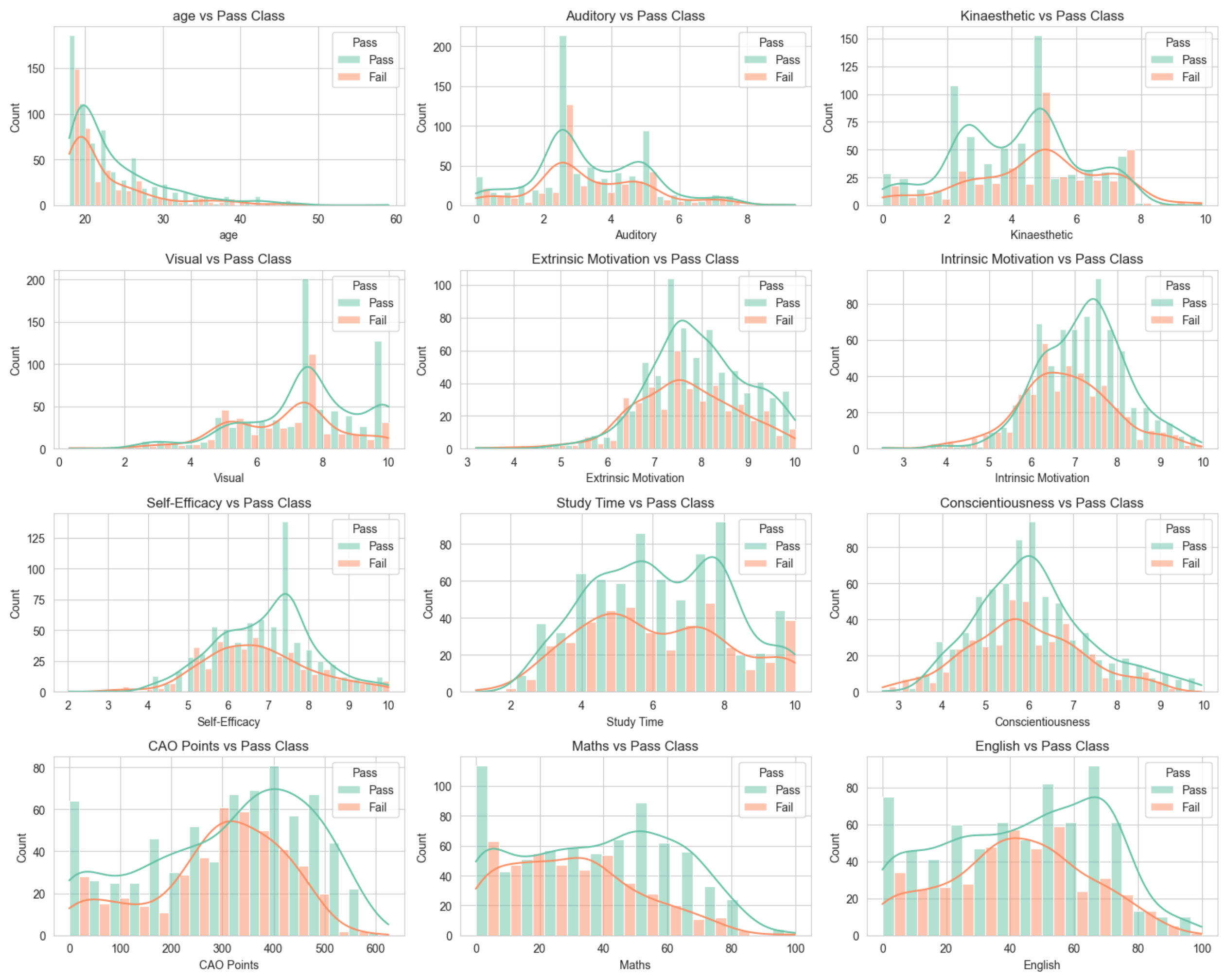


Figure 5: Distribution of student attributes by Pass class, showing frequency and density for each feature.

2 Classification models for Adult dataset

2.1 Algorithm choices

The goal is to compare different classification algorithms for predicting income levels in the Adult dataset, where the target variable is a binary class label. The first model chosen is **Logistic Regression**, a simple and interpretable model designed for binary classification. Although logistic regression is a form of regression, it is widely used for classification tasks where the outcome is a binary class. It is effective when the relationship between the features and the target is mostly linear and provides clear insights into how each feature impacts the model's prediction. This simplicity and interpretability make logistic regression a natural choice as a baseline model for comparison.

The second approach involves **Decision Trees**, followed by **XGBoost**, a powerful gradient boosting algorithm. Decision trees are able to handle both numerical and categorical data, and can capture non-linear relationships and interactions between features. However, their performance can often be limited by overfitting or underfitting. To address this, XGBoost is applied to enhance the decision tree model. As an ensemble method, XGBoost combines multiple decision trees and is particularly effective at managing complex, non-linear relationships, leading to improved accuracy and robustness.

2.2 Logistic regression model

Logistic regression is a classification algorithm used to predict a binary outcome (e.g., 0 or 1). It estimates the probability that an instance belongs to a particular class using the logistic (sigmoid) function, which is given by: $P(y=1|X) = \frac{1}{1+e^{-z}}$ which is the probability that the instance belongs to class 1, and z is the logit, defined as a linear combination of the input features and their coefficients $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$. To classify an instance, the predicted probability is compared to a threshold of 0.5. If $P(y=1|X) \geq 0.5$, the instance is classified as class 1. Else if $P(y=1|X) < 0.5$, the instance is classified as class 0. During training, the model minimizes the log-loss or binary cross-entropy cost function (Berkson, 1944): $J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\beta(x^{(i)})) + (1-y^{(i)})\log(1-h_\beta(x^{(i)}))\right)$ where m is the number of training examples, $y^{(i)}$ is the actual label of the i-th training example, and $h_\beta(x^{(i)})$ is the predicted probability for the i-th example.

Before using logistic regression, categorical data must be converted into numerical format, typically through one-hot encoding, label encoding, or ordinal encoding. One-hot encoding is preferred because it prevents false rankings of categories, which label and ordinal encoding might introduce. Logistic regression assumes numerical inputs have specific magnitudes, so label encoding could mistakenly imply that higher numbers are more important. Ordinal encoding works only for categories with a natural order, but for unordered categories, one-hot encoding avoids such assumptions by creating separate binary columns for each category, making it the best choice.

The next step will involve the preparation of the dataset for training and conducting an analysis using Python to evaluate its performance and interpret the results.

```python
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
# Initialize the OneHotEncoder for categorical attributes
onehot_encoder = OneHotEncoder(sparse_output=False)
# Initialize the OrdinalEncoder for the label column (no need to apply one-hot)
ordinal_encoder = OrdinalEncoder()
# Identify categorical attributes (excluding the label column)
ad_c = ad.select_dtypes(include=['object']).columns
# Remove the label from the categorical list
ad_c = ad_c[ad_c != 'label']
# Apply OrdinalEncoder to label column
ad['label'] = ordinal_encoder.fit_transform(ad[['label']])
# Apply one-hot encoding to the categorical features (excluding the label column)
ad_c_e = onehot_encoder.fit_transform(ad[ad_c])
# Convert the encoded data to a DataFrame
ad_c_e_df = pd.DataFrame(ad_c_e, columns=onehot_encoder.get_feature_names_out(ad_c))
# Drop original categorical columns (including label) and concatenate the encoded ones
ad_e = pd.concat([ad.drop(columns=ad_c), ad_c_e_df], axis=1)
```

Scaling numerical data ensures that all features have the same range and influence when training the model. Many algorithms, like logistic regression, are sensitive to the scale of input features. If features have very different ranges (e.g., one column has values between 0 and 1, while another has values in the thousands), the model might give more importance to the larger numbers, even if they are not more significant. By scaling the data, such as using `StandardScaler`, we standardize the features to have a mean of 0 and a standard deviation of 1. This helps the model learn more effectively and improves its performance.

```python
from sklearn.preprocessing import StandardScaler
# Initialize the StandardScaler
scaler = StandardScaler()
# Manually list the numerical columns
ad_n = ['age', 'fnlwgt', 'education_num', 'capital_gain', 'capital_loss', 'hours_per_week']
# replace with actual names of numerical columns
ad_e_s = ad_e.copy()
# Apply scaling to numerical columns
ad_e_s[ad_n] = scaler.fit_transform(ad_e[ad_n])
# Display the updated data with scaled numerical features
ad_e_s
```

When working with small datasets, **cross-validation** is often better than a simple **train-test split**. With a train-test split, only part of the data is used for training, which can result in an unrepresentative model, especially when the dataset is small. Cross-validation divides the data into multiple parts (folds) and uses each for both training and testing, ensuring the model learns from all available data. This approach provides better performance and more reliable evaluation.

Before training any model, it is essential to **split the dataset into training and testing sets**. The training set is used to build and optimize the model, while the test set evaluates how well the model performs on unseen data. By applying **cross-validation** to the training set, we can fine-tune the model and reduce the risk of overfitting. The final test set evaluation ensures the model can generalize to new, real-world data.

For this dataset, we will use a **70% train / 30% test split**. This gives the model sufficient data for training while keeping the test set large enough to evaluate its performance reliably. A larger test set helps ensure that performance metrics are robust and meaningful, giving a clearer picture of how the model will behave in real-world scenarios.

```
from sklearn.model_selection import train_test_split, cross_val_predict, cross_validate, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.linear_model import LogisticRegression
# Create X_ad_e_s containing all feature columns except 'label'
X_ad_e_s = ad_e_s[ad_e_s.columns.difference(['label'])]
y_ad_e_s = ad_e_s['label']
# Split the dataset into training and testing sets
X_ad_e_s_train, X_ad_e_s_test, y_ad_e_s_train, y_ad_e_s_test =
   train_test_split(X_ad_e_s, y_ad_e_s, test_size=0.3, stratify=y_ad_e_s, random_state=43)
# ``stratify=y_ad_e_s'' ensures that the distribution is maintained in both the training and test sets.
# Define the expanded parameter grid for Logistic Regression
lr_hyper_params = {
  'C': [0.01, 0.1, 1, 10, 100], # Regularization strength (larger range for flexibility)
  'penalty': ['l1', 'l2', 'elasticnet'], # Regularization types (L1, L2, or ElasticNet)
  'solver': ['liblinear', 'saga', 'newton-cg', 'lbfgs'], # Optimization algorithms
  'tol': [1e-4, 1e-3, 1e-2], # Tolerance for stopping criteria
  'fit_intercept': [True, False], # Whether to include an intercept in the model
  'class_weight': [None, 'balanced'] # Class weight options (useful for imbalanced datasets)
}
# Define the model (Logistic Regression)
lr_m_ad = LogisticRegression(max_iter=20000)
grid_search = GridSearchCV(estimator=lr_m_ad, param_grid=lr_hyper_params,
                           cv=10, scoring='precision_macro', n_jobs=-1)
grid_search.fit(X_ad_e_s_train, y_ad_e_s_train)
# Print best parameters
print(f"Best parameters found: {grid_search.best_params_}")
# Get the best model from GridSearchCV
best_lr_m_ad = grid_search.best_estimator_
```

```
Best parameters found: {'C': 0.1, 'class_weight': 'balanced', 'fit_intercept': False,
                        'penalty': 'l1', 'solver': 'liblinear', 'tol': 0.0001}
```

Logistic Regression's performance depends on hyperparameters like regularization strength (`C`) and penalty type (`l1` or `l2`). The regularization strength (`C`) controls the trade-off between fitting the training data and preventing overfitting. A smaller `C` applies stronger regularization, leading to a simpler model, while a larger `C` allows the model to fit the data more closely, risking overfitting. The penalty type defines the regularization used; `l1` encourages sparsity, useful for feature selection, while `l2` penalizes large coefficients to keep the model generalizable.

Choosing the right combination of these hyperparameters is key to optimizing model performance. Manual selection of these hyperparameters is time-consuming and prone to suboptimal results. `GridSearchCV` automates this process by systematically exploring combinations of hyperparameter values and selecting the combination that maximizes model performance, based on a chosen metric (e.g., precision in this case). Using GridSearch 1) **optimizes model performance** by ensuring that the best combination of hyperparameters is chosen for the given dataset, 2) **reduces human bias** by preventing guesswork and ensures a more objective selection process, and 3) **guarantees efficient evaluation** by using cross-validation to test each combination, providing a reliable estimate of performance.

```python
# Perform cross-validation on the best model
adsa.cross_validation_avg_scores(best_lr_m_ad, X_ad_e_s_train, y_ad_e_s_train, cv_=10)
# Get predictions for the entire dataset using cross-validation
y_ad_e_s_train_preds = cross_val_predict(best_lr_m_ad, X_ad_e_s_train, y_ad_e_s_train, cv=10)
# Print classification report
print(classification_report(y_ad_e_s_train, y_ad_e_s_train_preds))
# Plot confusion matrix
adsa.plot_confusion_matrix(y_ad_e_s_train, y_ad_e_s_train_preds)
```

```
Mean accuracy: 82.64% +/-3.96%
Mean precision: 83.38% +/-3.79%
Mean recall: 82.53% +/-4.00%
Mean F1-score is 82.49% +/-4.06%
              precision    recall  f1-score   support
         0.0       0.87      0.76      0.81       326
         1.0       0.79      0.89      0.84       336
    accuracy                           0.83       662
   macro avg       0.83      0.83      0.83       662
weighted avg       0.83      0.83      0.83       662
```

Accuracy measures how many predictions were correct overall. It is calculated by dividing the number of correct predictions (true positives and true negatives) by the total number of instances: Accuracy $= \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$. In this case, the accuracy is **82.64%**, meaning the model correctly predicted the class for about 83% of the instances. While useful, accuracy can be misleading when the dataset is imbalanced because it may overlook the model's performance on the minority class.

Precision measures how many of the predicted positive instances are actually correct. It is calculated as the number of true positives (TP) divided by the sum of true positives and false positives (FP): Precision $= \frac{\text{TP}}{\text{TP+FP}}$. For the `<=50K` class, the precision is **87%**, meaning most predictions for this class are correct. For the `>50K` class, the precision is **79%**, indicating that a smaller proportion of predicted `>50K` instances were correct. Precision is important when the cost of false positives is high.

Recall shows how well the model identifies all the actual positive instances. It is calculated as the number of true positives (TP) divided by the sum of true positives and false negatives (FN): Recall $= \frac{\text{TP}}{\text{TP+FN}}$ For `<=50K`, recall is **76%**, meaning the model missed about **24%** of the instances for this class. For `>50K`, recall is **89%**, meaning the model identified most of the `>50K` instances. Recall is crucial when it is important to identify as many positives as possible.

The F1-score is the harmonic mean of precision and recall, providing a balance between them. It is calculated as: F1-score $= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}}$ For the `<=50K` class, the F1-score is **81%**, and for the `>50K` class, it is **84%**. This indicates that the model balances precision and recall well for both classes, with a slightly better performance for `>50K`.

The output suggests that the model achieved a mean accuracy of 82.64% with a small standard deviation (3.96%), indicating a good general performance with slight variability across folds. The precision for both classes is good (87% for `<=50K` and 80% for `>50K`), but there is a trade-off between precision and recall, with recall being slightly lower for `<=50K` (76%). The F1-score, which balances both precision and recall, is around 82.49%, indicating that the model is well-balanced in terms of both metrics..

```python
# Apply the model trained above to the test portion
y_ad_e_s_test_pred = best_lr_m_ad.predict(X_ad_e_s_test)
print(classification_report(y_ad_e_s_test, y_ad_e_s_test_pred))
adsa.plot_confusion_matrix(y_ad_e_s_test, y_ad_e_s_test_pred)
```

```
              precision    recall  f1-score   support
         0.0       0.86      0.72      0.78       141
         1.0       0.76      0.88      0.82       144
    accuracy                           0.80       285
   macro avg       0.81      0.80      0.80       285
weighted avg       0.81      0.80      0.80       285
```

The model's performance on the test set shows its ability to generalize to new data. It achieved an accuracy of **80%**, with precision of **86%** for `<=50K` and **76%** for `>50K`. The recall was **72%** for `<=50K` and **88%** for `>50K`, indicating the model struggles a bit with the `<=50K` class. The F1-scores were **78%** for `<=50K` and **82%** for `>50K`, showing a good balance between precision and recall. Overall, the model performs well but could be improved, especially for the `<=50K` class (see Figure 6).

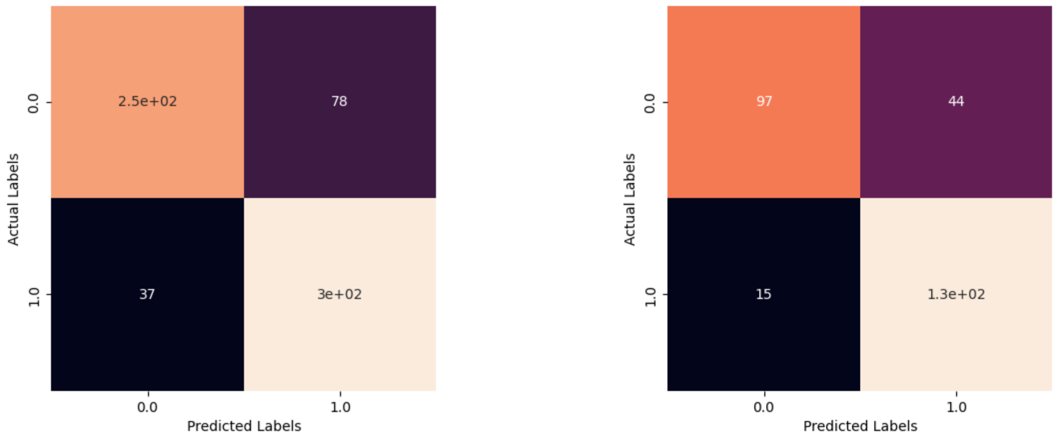


Figure 6: Confusion matrix of the best logistic regression model from GridSearch (left), and its application on the test set (right).

2.3 Decision trees and XGBoost models

Decision trees are widely used supervised learning methods for both classification and regression tasks. In this approach, the data is split into smaller subsets based on specific criteria, creating a tree-like structure where the **leaf nodes** represent the final classification or decision. The quality of each split is evaluated using measures like **entropy** or the **Gini index**, ensuring that the divisions are meaningful. However, if the tree grows too deep, **overfitting** can occur, where the model starts to memorize noise instead of general patterns.

Following the application of logistic regression to the Adult dataset, we initially considered using the **C5.0 algorithm** for further analysis and classification. This algorithm improves upon its predecessor, **C4.5**, by incorporating **boosting**, allowing weak models to be combined for higher accuracy. However, it requires the installation of additional libraries, such as `pyC5` or `C50`, to function properly. Instead of relying on C5.0, we have decided to use **XGBoost**, a powerful gradient boosting algorithm, which is more widely supported and requires no extra libraries for installation in Jupyter.

The **C4.5 algorithm**, on the other hand, will still be useful for initial analysis. It handles continuous attributes and missing data well, and it employs **post-pruning** to reduce overfitting by removing unnecessary branches. By using **C4.5** and later transitioning to **XGBoost**, we aim to achieve better classification performance and gain deeper insights into the Adult dataset.

```
from sklearn.preprocessing import OrdinalEncoder
ad = adult_data.copy()
X_ad = ad[ad.columns.difference(['label'])]
y_ad = ad[['label']]
# Initialize the OrdinalEncoder
encoder = OrdinalEncoder()
# Identify categorical columns
X_ad_c = X_ad.select_dtypes(include=['object']).columns
# Apply ordinal encoding to categorical columns
X_ad_c_e = encoder.fit_transform(X_ad[X_ad_c])
# Convert the encoded data to a DataFrame
X_ad_c_e = pd.DataFrame(X_ad_c_e, columns=encoder.get_feature_names_out(X_ad_c))
# Drop the original categorical columns and concatenate the encoded ones
X_ad_e = pd.concat([X_ad.drop(columns=X_ad_c), X_ad_c_e], axis=1)
# Display the updated data
X_ad_e.info()
```

In this code, **Ordinal Encoding** is used to prepare the **categorical attributes** of the dataset for the decision tree classifier. Ordinal encoding is sufficient for decision trees because they can handle categorical data with an inherent order. There is no need to apply one-hot encoding to categorical features in this case. The features (`X_ad`) are separated from the **label** (`y_ad`), and categorical columns in the features are encoded using ordinal encoding. The encoded columns are then added back to the dataset, replacing the original categorical columns. Labels do not need to be encoded for decision trees, as they can handle both categorical and numerical labels without the need for encoding.

```
# Base model
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
# Split the dataset into training and testing sets
X_ad_e_train, X_ad_e_test, y_ad_train, y_ad_test =
  train_test_split(X_ad_e, y_ad, test_size=0.3, stratify=y_ad, random_state=43)
# Define the model (e.g., decision tree classifier)
dtc_m_ad = DecisionTreeClassifier(random_state=43)
adsa.custom_crossvalidation(X_ad_e_train, y_ad_train, dtc_m_ad)
```

The comparison of the base decision tree classifier and the logistic regression model on the adult dataset reveals significant performance differences. The decision tree model achieved a mean accuracy of 76.44% (±3.86%), whereas the logistic regression model performed better with a mean accuracy of 82.64% (±3.96%). In terms of precision, recall, and F1-score, the logistic regression model also outperformed the decision tree. The macro averages also favored the logistic regression model, with values of 83% for precision, recall, and F1-score compared to 76% for the decision tree. This indicates that logistic regression is more balanced and performs better across both classes (`<=50K` and `>50K`). Based on these results, logistic regression outperforms the decision tree classifier across all key metrics, suggesting that logistic regression is a more suitable model for this dataset.

```
Mean accuracy: 76.44% +/-3.86%
Mean precision: 76.49% +/-3.83%
Mean recall: 76.45% +/-3.86%
Mean F1-score is 76.43% +/-3.87%
              precision    recall  f1-score   support
       <=50K       0.76      0.76      0.76       326
        >50K       0.77      0.77      0.77       336
    accuracy                           0.76       662
   macro avg       0.76      0.76      0.76       662
weighted avg       0.76      0.76      0.76       662
```

```
# Define hyperparameter grid
dtc_hyper_params = {
  'criterion': ['gini', 'entropy', 'log_loss'],
  # Splitting criteria options for the decision tree
  'max_depth': range(3, 13),
  # Range of values for maximum tree depth to prevent overfitting
  'ccp_alpha': [0.003, 0.004, 0.005]
  # Values for complexity parameter used for Minimal Cost-Complexity Pruning
}
# Set up GridSearchCV to find the best combination of hyperparameters
grid_search = GridSearchCV(dtc_m_ad, param_grid=dtc_hyper_params,
                           scoring='f1_macro', cv=5, n_jobs=-1)
# Fit the GridSearchCV object to the training data
grid_search.fit(X_ad_e_train, y_ad_train)
# Print the best combination of hyperparameters found during grid search
print(grid_search.best_params_)
# Print the best F1 macro score achieved using the best combination of hyperparameters
print(grid_search.best_score_)
```

```
'ccp_alpha': 0.003, 'criterion': 'gini', 'max_depth': 4
0.8140016574158505
```

```
from sklearn.pipeline import Pipeline
best_dtc_m_ad_prepost = Pipeline([
  ('encoder', OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)),
  ('sklearn_dt', grid_search.best_estimator_)
])
best_dtc_m_ad_prepost.fit(X_ad_e_train, y_ad_train)
adsa.custom_crossvalidation(X_ad_e_train, y_ad_train, best_dtc_m_ad_prepost)
```

```
Mean accuracy: 80.66% +/-4.11%
Mean precision: 81.08% +/-3.93%
Mean recall: 80.57% +/-4.14%
Mean F1-score is 80.54% +/-4.21%
              precision    recall  f1-score   support
       <=50K       0.84      0.75      0.79       326
        >50K       0.78      0.86      0.82       336
    accuracy                           0.81       662
   macro avg       0.81      0.81      0.81       662
weighted avg       0.81      0.81      0.81       662
```

The optimal decision tree classifier, tuned using GridSearchCV, achieved a mean accuracy of 80.66% (±4.11%), showing significant improvement over the base decision tree's 76.44%. Precision, recall, and F1-scores also improved, demonstrating the value of hyperparameter tuning. However, **the logistic regression model remained superior, with a mean accuracy of 82.64% (±3.96%) and better overall class balance**. While the optimal decision tree narrowed the performance gap, logistic regression consistently outperformed it across all key metrics, making it the most effective model for this dataset.

```python
from xgboost import XGBClassifier
import xgboost as xgb
# Create a copy of the dataset to avoid modifying the original data
XX_ad = X_ad.copy()
# Identify columns with object datatype (categorical features)
XX_ad_c = XX_ad.select_dtypes(include='object').columns
# Convert categorical columns to 'category' datatype for compatibility with XGBoost
XX_ad[XX_ad_c] = XX_ad[XX_ad_c].astype('category')
# Display dataset information to confirm datatype changes
yy_ad = y_ad.copy()
yy_ad_e = yy_ad['label'].map({'<=50K': 0, '>50K': 1})
# Simulate a single decision tree; leave other hyperparameters to default values
xgb_dt_m_ad = XGBClassifier(enable_categorical=True,
                            objective='multi:softmax',
                            booster='dart',
                            num_class=2,
                            eval_metric='mlogloss',
                            random_state=43)
# crossvalidate and output performance
adsa.custom_crossvalidation(XX_ad_train, yy_ad_e_train, xgb_dt_m_ad)
```

```
Mean accuracy: 78.99% +/-3.64%
Mean precision: 79.46% +/-3.83%
Mean recall: 78.93% +/-3.60%
Mean F1-score is 78.86% +/-3.63%
              precision    recall  f1-score   support
           0       0.79      0.77      0.78       317
           1       0.79      0.81      0.80       345
    accuracy                           0.79       662
   macro avg       0.79      0.79      0.79       662
weighted avg       0.79      0.79      0.79       662
```

The base XGBoost model achieved a mean accuracy of 78.99% (±3.64%), with precision, recall, and F1-scores of 79.46%, 78.93%, and 78.86%, respectively. It outperformed the base decision tree model (76.44% accuracy) but still lagged behind the logistic regression model, which had an accuracy of 82.64%. Although the base XGBoost model showed balanced performance across both classes, there is potential for improvement through hyperparameter tuning to surpass logistic regression and achieve better overall performance.

The key features of the XGBoost algorithm used in the provided code are as follows:

- **Objective function**: The model uses a multi-class classification objective (`multi:softmax`) suitable for binary classification, distinguishing between the classes `<=50K` and `>50K`.
- **Booster type**: The `DART` (Dropouts meet Multiple Additive Regression Trees) booster is used to enhance model robustness by dropping random trees during training.
- **Categorical data support**: The parameter `enable_categorical=True` allows the model to efficiently handle categorical features without the need for one-hot encoding.
- **Number of classes**: The `num_class=2` parameter specifies that the model is designed for binary classification.
- **Evaluation metric**: The `eval_metric='mlogloss'` evaluates the model using multi-class log-loss, which is effective for classification tasks.

While the initial XGBoost model has provided a solid baseline for performance, the next step involves optimizing the model to further enhance its predictive capabilities. In the following phase, we will focus on fine-tuning key hyperparameters to achieve improved accuracy and efficiency.

```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

xgb_hyper_params = {
  'n_estimators': [9,13,17,21,23],
  #'num_parallel_tree': [1],
  'learning_rate': uniform(0.01, 0.5),
  'max_depth' : range(3, 13),
  'subsample': uniform(0.6, 0.4),
  'colsample_bytree': uniform(0.6, 0.4),
  'gamma': uniform(0, 0.5),
  'min_child_weight': range(1,6,1)
}

random_search = RandomizedSearchCV(xgb_dt_m_ad, param_distributions=xgb_hyper_params,
                                   n_iter=200, cv=5, n_jobs=-1,
                                   random_state=43, scoring='f1_macro')

random_search.fit(XX_ad_train, yy_ad_e_train)
print(random_search.best_params_)
```

```
{'colsample_bytree': np.float64(0.7276534486279523),
  'gamma': np.float64(0.4270550872289328),
  'learning_rate': np.float64(0.21397216318953555),
  'max_depth': 4,
  'min_child_weight': 3,
  'n_estimators': 23,
  'subsample': np.float64(0.6123874393511733)}
```

The code utilizes `RandomizedSearchCV` to tune the hyperparameters of an XGBoost classifier. This method performs a random search over a specified hyperparameter grid. It evaluates multiple combinations using cross-validation and selects the best-performing set of parameters. The main advantage of randomized search over grid search is that it explores a wider range of parameters in less time. The key components of the hyperparameter tuning process are explained as follows:

- `n_estimators`: This parameter specifies the number of trees (boosting rounds) in the XGBoost model. A higher value typically increases the model's complexity and may lead to overfitting. In this case, only one value is tested: 23 trees (several values were tested, and 23 was found to be the best). When multiple values were tested together, the accuracy was lower.
- `learning_rate`: The learning rate controls the contribution of each tree to the final prediction. Smaller values (e.g., 0.01) slow down the learning process, potentially improving generalization, but requiring more trees to fit the model.
- `max_depth`: This sets the maximum depth of each individual tree. Deeper trees can model more complex relationships but may lead to overfitting. The search is performed over depths from 3 to 12.
- `subsample`: The fraction of the training data used for each tree. This parameter introduces randomness and can help prevent overfitting. A value between 0.6 and 1.0 is typically used.
- `colsample_bytree`: The fraction of features selected for each tree. This parameter adds another layer of randomness, helping to prevent overfitting. Values between 0.6 and 1.0 are commonly tested.
- `gamma`: This parameter controls the minimum loss reduction required to make a further partition in the tree. Higher values make the model more conservative, leading to fewer splits and possibly better generalization.
- `min_child_weight`: This defines the minimum sum of instance weights (hessian) in a child node. Larger values help prevent overfitting by requiring more data in each leaf node.

The code executes 200 iterations of random hyperparameter combinations, assessing performance through 5-fold cross-validation while utilizing all 12 cores of the machine.

```
best_xgb_m_dt = random_search.best_estimator_
adsa.custom_crossvalidation(XX_ad_train, yy_ad_e_train, best_xgb_m_dt)
```

```
Mean accuracy: 84.14% +/-3.52%
Mean precision: 84.36% +/-3.38%
Mean recall: 84.01% +/-3.63%
Mean F1-score is 84.05% +/-3.58%
              precision    recall  f1-score   support
           0       0.85      0.81      0.83       317
           1       0.83      0.87      0.85       345
    accuracy                           0.84       662
   macro avg       0.84      0.84      0.84       662
weighted avg       0.84      0.84      0.84       662
```

```
yy_ad_e_test_pred = best_xgb_m_dt.predict(XX_ad_test)
print(classification_report(yy_ad_e_test_pred, yy_ad_e_test))
adsa.plot_confusion_matrix(yy_ad_e_test, yy_ad_e_test_pred)
```

```
             precision    recall  f1-score   support
           0       0.75      0.84      0.79       134
           1       0.84      0.75      0.79       151
    accuracy                           0.79       285
   macro avg       0.79      0.79      0.79       285
weighted avg       0.79      0.79      0.79       285
```

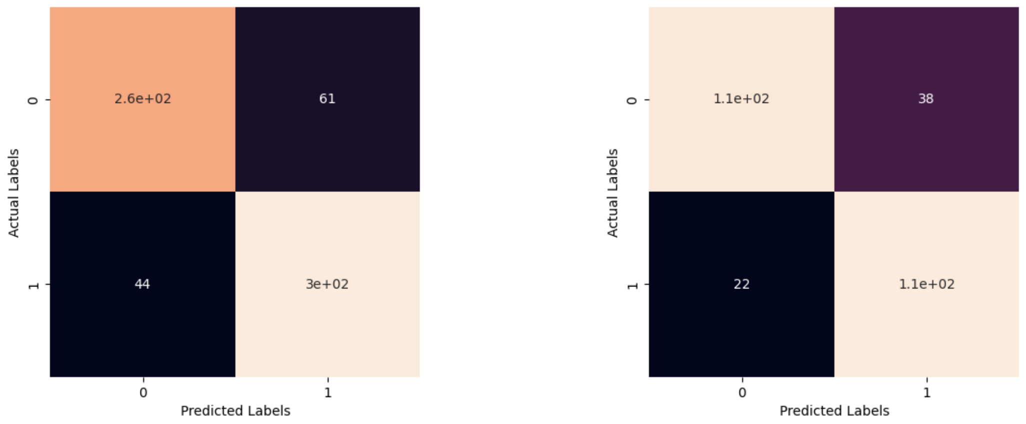


Figure 7: Confusion matrix of the best XGboost model from RandomizedSearchCV (left), and its application on the test set (right).

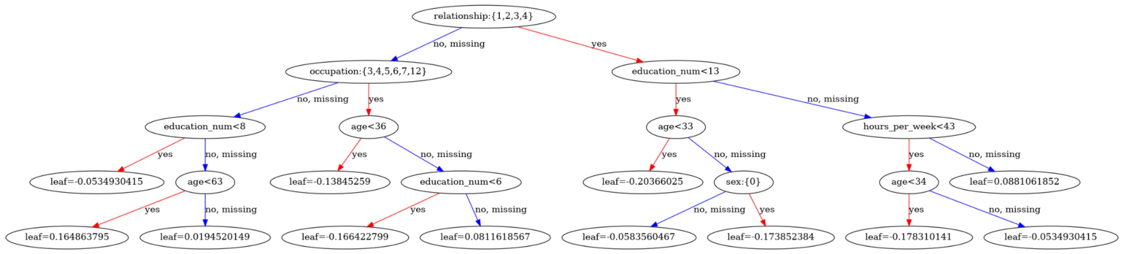


Figure 8: Optimal XGboost tree.

The best XGBoost model (see Figure 8)significantly outperforms the logistic regression and optimal decision tree models in terms of mean accuracy, precision, recall, and F1-score, particularly with more balanced class-wise performance. XGBoost shows superior performance across all key metrics, making it the best model for this dataset (see Figure **??**).

```
# ROC curves for 3 optimal models
best_lr_m_ad_disp = RocCurveDisplay.from_estimator(best_lr_m_ad, X_ad_e_s_test, y_ad_e_s_test)
best_dtc_m_ad_prepost_named_disp =
    RocCurveDisplay.from_estimator(best_dtc_m_ad_prepost_named,
                                   X_ad_e_test, y_ad_test,
                                   ax=best_lr_m_ad_disp.ax_)
best_xgb_m_dt_disp =
    RocCurveDisplay.from_estimator(best_xgb_m_dt,
                                   XX_ad_test, yy_ad_e_test,
                                   ax=best_dtc_m_ad_prepost_named_disp.ax_)
best_xgb_m_dt_disp.figure_.suptitle("ROC curve comparison")
plt.plot([0, 1], [0, 1], 'k--')
```

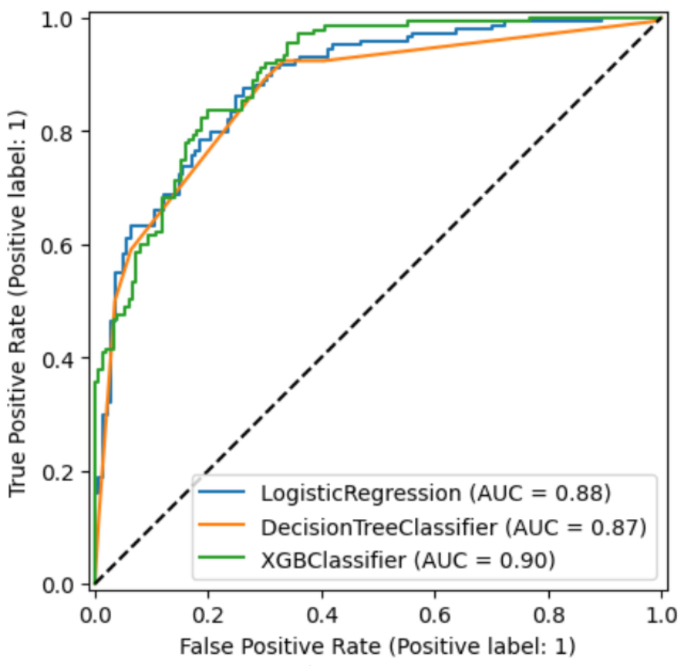


Figure 9: ROC-AUC curve comaraison.

The ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) is a performance metric used to evaluate the classification ability of a model. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR), while the AUC represents the area under this curve, with values ranging from 0 to 1. A higher AUC indicates better model performance. In this case, the AUC values for the models are as follows: Logistic Regression achieved an AUC of 0.88, DecisionTreeClassifier scored 0.87, and the XGBoost Classifier outperformed both with an AUC of 0.90, indicating that XGBoost has the best ability to discriminate between the two classes in this dataset (see Figure 9).

3 Classification models for Student dataset

3.1 Algorithm choices

The goal is to compare different classification algorithms for predicting whether a student will pass or fail in their first-year college exams, using a dataset with various academic and non-cognitive features. The first algorithm chosen is **K-Nearest Neighbors (KNN)**, a simple and intuitive model that classifies students based on the proximity of similar students in terms of their attributes, such as study time and exam scores. KNN is easy to implement and works well when similar students tend to have the same outcome. However, it can be slow during prediction and requires proper scaling of features to ensure accurate distance calculations.

The second approach involves **Support Vector Machine (SVM)**, a powerful algorithm that aims to find the best boundary separating students who pass from those who fail. SVM works well with both linear and non-linear data and is effective in high-dimensional spaces. Although it requires careful tuning of parameters, such as the choice of kernel and regularization, SVM often provides strong performance, especially when the relationships between the features and the target are complex.

3.2 K-Nearest Neighbors model

```
# Initialize the StandardScaler
scaler = StandardScaler()
# Apply scaling to numerical columns
X_sd_s = scaler.fit_transform(X_sd)
X_sd_s = pd.DataFrame(X_sd_s, columns=X_sd.columns)

# Split the scaled data
X_sd_s_train, X_sd_s_test, y_sd_train, y_sd_test =
  train_test_split(X_sd_s, y_sd, test_size=0.2, random_state=43)
```

```
from sklearn.neighbors import KNeighborsClassifier

knn_hyper_params = {
  'n_neighbors' : range(3, 20),
  'weights': ['uniform', 'distance'],
  'p': [1, 2]
}
grid_search = GridSearchCV(knn_m_sd, param_grid=knn_hyper_params, scoring='f1_weighted')
grid_search.fit(X_sd_s_train, y_sd_train)
print(grid_search.best_params_)
print(grid_search.best_score_)
best_knn_m_sd = grid_search.best_estimator_
```

```
'n_neighbors': 4, 'p': 1, 'weights': 'distance'
0.8139514368350603
```

The `f1_weighted` scoring metric takes into account the support (the number of true instances for each class) and adjusts the F1 score accordingly. This is important when the dataset is imbalanced, as the model's performance on the minority class is weighted more. In our case, the dataset is imbalanced, with a larger number of "Pass" instances compared to "Fail" instances. Therefore, `f1_weighted` is a better choice to reflect the model's performance on both classes, giving more importance to the minority class. On the other hand, `f1_macro` treats all classes equally, regardless of their support, which might not be ideal in the case of imbalanced classes.

```
adsa.custom_crossvalidation(X_sd_s_train, y_sd_train, best_knn_m_sd)
```

```
Mean accuracy: 81.53% +/-1.51%
Mean precision: 80.43% +/-1.86%
Mean recall: 79.49% +/-1.31%
Mean F1-score is 79.88% +/-1.50%
              precision    recall  f1-score   support
        Fail       0.77      0.72      0.74       333
        Pass       0.84      0.87      0.86       571
    accuracy                           0.82       904
   macro avg       0.80      0.79      0.80       904
weighted avg       0.81      0.82      0.81       904
```

The KNN model achieved promising results with optimal hyperparameters: `n_neighbors=4`, `p=1` (Manhattan distance), and `weights='distance'`. The cross-validation results showed a mean F1-macro score of **79.88%**, accuracy of **81.53%**, precision of **80.43%**, and recall of **79.49%** with minimal variability. On the test set, the model achieved an accuracy of **82%**. It performed well in identifying the "`Pass`" class (precision: 0.84, recall: 0.87) but showed moderate performance for the "`Fail`" class (precision: 0.77, recall: 0.72).

3.3 Support Vector Machine model

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

svc_hyper_params = {
  'C': loguniform(0.001, 10),
  'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
  'degree': [2, 3, 4],
  'gamma': loguniform(0.0001, 1),
  'probability': [True, False]
}

random_search = RandomizedSearchCV(
                     SVC(random_state=43),
                     param_distributions=svc_hyper_params,
                     n_iter=200,
                     scoring='f1_weighted',
                     cv=10,
                     n_jobs=-1)
random_search.fit(X_sd_s_train, y_sd_train)
print("Best parameters:", random_search.best_params_)
print("Best score:", random_search.best_score_)
```

```
{'C': 2.650944687910482,
    'degree': 3,
    'gamma': 0.27994915748064286,
    'kernel': 'rbf',
    'probability': True}
0.8401366555969249
```

The hyperparameters in the SVC model help adjust how the model learns from the data. Here is a brief explanation of each:

- `C`: This controls the penalty for misclassification. A higher value tries to reduce errors but may lead to overfitting, while a lower value makes the model more flexible.
- `kernel`: Defines the type of decision boundary. Options include:
  - `linear`: For linearly separable data.
  - `rbf`: For non-linear boundaries, often works well for complex data.
  - `poly`: Useful for polynomial relationships.
  - `sigmoid`: Similar to neural networks but less commonly used.
- `degree`: Applies only when using the `poly` kernel, controlling the degree of the polynomial. Higher values allow more complex decision boundaries.
- `gamma`: Controls how far the influence of a single training example reaches. High values make the boundary more complex and may lead to overfitting.
- `probability`: If `True`, the model outputs probability estimates instead of just class predictions. This is useful for some evaluation metrics like AUC.

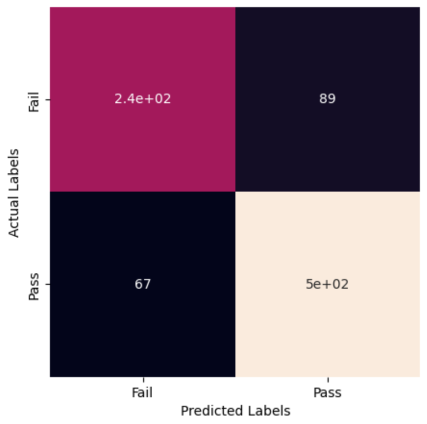
The model with these hyperparameters achieves an **F1-weighted score of 0.84**, indicating a strong balance between precision and recall across the different classes.

```
vc_m_sd = randomized_search.best_estimator_
ustom_crossvalidation(X_sd_s_train, y_sd_train, best_svc_m_sd)
```

```
Mean accuracy: 82.74% +/-0.96%
Mean precision: 81.77% +/-1.07%
Mean recall: 80.76% +/-1.18%
Mean F1-score is 81.17% +/-1.10%
              precision    recall  f1-score   support
        Fail       0.78      0.73      0.76       333
        Pass       0.85      0.88      0.87       571
    accuracy                           0.83       904
   macro avg       0.82      0.81      0.81       904
weighted avg       0.83      0.83      0.83       904
```

```
y_sd_test_pred = best_svc_m_sd.predict(X_sd_s_test)
print(classification_report(y_sd_test_pred, y_sd_test))
adsa.plot_confusion_matrix(y_sd_test, y_sd_test_pred)
```

```
              precision    recall  f1-score   support
        Fail       0.83      0.81      0.82        89
        Pass       0.88      0.89      0.88       138
    accuracy                           0.86       227
   macro avg       0.85      0.85      0.85       227
weighted avg       0.86      0.86      0.86       227
```



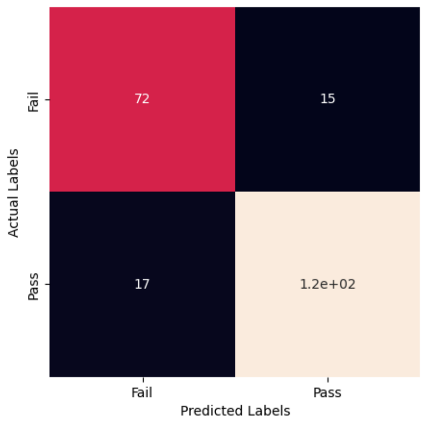


Figure 10: Confusion matrix of the best SVC model from RandomizedSearchCV (left), and its application on the test set (right).

The **SVM model** achieves a higher F1-weighted score of **0.84** compared to the **KNN model**, which has an F1-weighted score of **0.81**. The SVM model also demonstrates better overall performance, with a test set accuracy of **86%** compared to the KNN's **82%**. The KNN model, with the parameters {'n_neighbors': 4, 'p': 1, 'weights': 'distance'}, shows good performance, particularly in precision for the "`Pass`" class, with a recall of **87%** and an F1-score of **0.86** for "`Pass`". However, the SVM model outperforms KNN in terms of both precision and recall for the "`Fail`" class and provides better overall generalization. The SVM model is the better choice for this dataset based on its higher F1-weighted score and superior accuracy.

The ROC-AUC diagram indicates that the **best KNN model** achieves an AUC of **91**%, while the **best SVM model** performs slightly better with an AUC of **92**%. This demonstrates that both models are performing very well in distinguishing between the classes. The AUC values close to 1 suggest that both models have a strong ability to correctly classify both classes, with the SVM model marginally outperforming the KNN model in terms of AUC. Despite the slight difference in AUC, the SVM model's higher overall performance across multiple metrics, including accuracy and F1-weighted score, indicates it may be the preferable model for this dataset (see Figure 11).

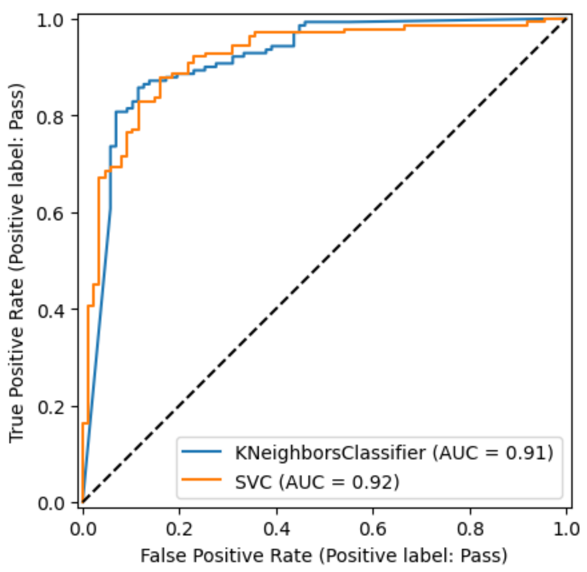


Figure 11: ROC-AUC curve comaraison for the Student dataset model.

References

Joseph Berkson. Application of the Logistic Function to Bio-Assay. Journal of the American statistical association, 39(227):357–365, 1944.