Assignment-4: Decision Tree & XGBooster

Mahbub Ahmed Turza
ID: 2211063042
North South University
mahbub.turza@northsouth.edu

November 15, 2024

Abstract

Machine learning has become a cornerstone of predictive modeling across diverse domains, providing powerful tools for regression, classification, and decision-making tasks. This report investigates the performance of two widely used algorithms—Decision Tree Regressor/Classifier and XGBoost Regressor/Classifier—on four distinct datasets: `Car Dataset`, `DT Brain Cancer`, `DT-Wage`, and `DT-Credit`. These datasets represent a variety of tasks, including regression, binary classification, and multi-class classification, ensuring a comprehensive evaluation.

The study employs metrics such as Mean Squared Error (MSE), accuracy, and classification reports to assess model performance. Results reveal the trade-offs between simplicity and predictive power, with Decision Trees excelling in interpretability and XGBoost demonstrating superior accuracy and generalization. By analyzing the algorithms' strengths and weaknesses across diverse datasets, this report offers insights into their suitability for different types of predictive tasks. The findings contribute to better-informed decisions when selecting machine learning models for real-world applications.

I. Introduction

In recent years, machine learning has become an indispensable tool for predictive modeling in various domains, enabling accurate forecasting and decision-making. This report explores the application of machine learning algorithms on four distinct datasets: `Car Dataset`, `DT Brain Cancer`, `DT-Wage`, and `DT-Credit`. These datasets encompass a wide range of prediction tasks, from regression and classification to multi-class predictions, making them ideal for evaluating the strengths and weaknesses of different algorithms.

A. Objective

The primary objective of this study is to compare the performance of two popular machine learning models—**Decision Tree Regressor** and **XGBoost Regressor/Classifier**—across the four datasets. The models are evaluated using key performance metrics such as Mean Squared Error (MSE), R-squared (R^2), accuracy, and residual analysis to assess their suitability for each dataset and task. Additionally, the study aims to provide insights into the trade-offs between model interpretability and predictive accuracy.

B. Datasets

The four datasets used in this study are as follows:

- **DT-Credit:** Multi-class Classification
- **Brain Cancer:** Binary Classification
- **Wage Dataset:** Regression
- **Credit Dataset:** Regression

C. Significance of Study

The selection of these datasets allows for an evaluation of the models under varying conditions and complexities. The diversity in dataset structure, feature types, and target variables provides a robust testbed for understanding how different algorithms handle different types of prediction problems. By analyzing the performance of Decision Trees and XGBoost across these datasets, this study contributes to the growing body of knowledge on the applicability of machine learning models in real-world scenarios.

D. Report Structure

The remainder of this report is structured as follows:

- **Methodology:** Details the preprocessing steps, hyperparameter tuning, and model evaluation strategies.
- **Results:** Presents the performance metrics of the models on each dataset.
- **Discussion:** Compares and contrasts the findings across datasets and models.
- **Conclusion:** Summarizes the key takeaways and provides recommendations for model selection based on task requirements.

This comprehensive approach ensures that the report provides valuable insights into the capabilities and limitations of the chosen machine learning models across diverse datasets and tasks.

II. Methodology

A. For Car Dataset:

- **Data Preprocessing :** To start, we load and preprocess the car dataset, which contains features such as buying price, maintenance cost, number of doors, passenger capacity, trunk size, and safety level. Each feature contributes to determining the car's overall acceptability.

```
            import pandas as pd
            from sklearn.model_selection import train_test_split
            from sklearn.preprocessing import LabelEncoder

            # Load dataset
            data = pd.read_csv('car_data.csv')
            print(data.head())

            # Initialize label encoder
            label_encoder = LabelEncoder()

            # Encoding each categorical column
            for col in data.columns:
                data[col] = label_encoder.fit_transform(data[col])
            print(data.head())

```

  Each categorical feature is converted into numeric form using `LabelEncoder` from `sklearn.preprocessing`, as both models require numerical input.

- **Splitting Data into Training and Testing Sets:** To evaluate model performance, the dataset is split into training and testing sets.

```
        X = data.drop('class', axis=1)
        y = data['class']
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

```

  Here, the target variable `class` is separated from features `X`. An 80-20 split is used, where 80% of the data is used for training, and 20% is reserved for testing.

- **Model Selection and Training: 1) Decision Tree Classifier:** The Decision Tree classifier is initialized and trained on the training set.

```
        from sklearn.tree import DecisionTreeClassifier

        dt_model = DecisionTreeClassifier(random_state=42)
        dt_model.fit(X_train, y_train)

```

  2) XGBoost Classifier: The XGBoost model is initialized and trained with specific configuration parameters.

```
        from xgboost import XGBClassifier

        xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss',
    random_state=42)
        xgb_model.fit(X_train, y_train)

```

  The `XGBClassifier` is initialized with `use_label_encoder=False` and `eval_metric='mlogloss'`, as required by XGBoost for multi-class classification.

- **Model Evaluation :** The models are evaluated on accuracy, precision, recall, and F1-score.

```
        from sklearn.metrics import accuracy_score, classification_report

        dt_pred = dt_model.predict(X_test)
```

```
        xgb_pred = xgb_model.predict(X_test)

        dt_accuracy = accuracy_score(y_test, dt_pred)
        xgb_accuracy = accuracy_score(y_test, xgb_pred)

        print("Decision Tree Accuracy:", dt_accuracy)
        print("XGBoost Accuracy:", xgb_accuracy)

        print("Decision Tree Classification Report:")
        print(classification_report(y_test, dt_pred))

        print("XGBoost Classification Report:")
        print(classification_report(y_test, xgb_pred))

```

`accuracy_score` computes the accuracy, while `classification_report` provides precision, recall, and F1-score for each class.

- **Comparative Analysis Using Confusion Matrix :**The confusion matrix for each model is displayed for better insight into misclassifications.

```
        from sklearn.metrics import confusion_matrix
        import matplotlib.pyplot as plt
        import seaborn as sns

        dt_conf_matrix = confusion_matrix(y_test, dt_pred)
        plt.figure(figsize=(8, 6))
        sns.heatmap(dt_conf_matrix, annot=True, fmt="d", cmap="Blues")
        plt.title("Decision Tree Confusion Matrix")
        plt.xlabel("Predicted Label")
        plt.ylabel("True Label")
        plt.show()

        xgb_conf_matrix = confusion_matrix(y_test, xgb_pred)
        plt.figure(figsize=(8, 6))
        sns.heatmap(xgb_conf_matrix, annot=True, fmt="d", cmap="Greens")
        plt.title("XGBoost Confusion Matrix")
        plt.xlabel("Predicted Label")
        plt.ylabel("True Label")
        plt.show()

```

The confusion matrices are generated using `confusion_matrix` from `sklearn.metrics`, visualized with `seaborn` and `matplotlib`. The matrices illustrate true positives, false positives, and false negatives for both classifiers.

B. For Brain Cancer Dataset

- **Importing Libraries**
  Essential libraries for data handling, visualization, and model training were imported as follows:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
from sklearn.metrics import precision_recall_curve
from sklearn.preprocessing import StandardScaler, OneHotEncoder

```

- **Loading and Exploring the Data**
  The dataset was loaded, checked for missing values, and unnecessary columns were dropped.

```
df = pd.read_csv("DT-BrainCancer.csv")
df.info()  # To understand data types and missing values
df = df.dropna()  # Removing rows with missing values
df = df.drop('Unnamed: 0', axis=1)  # Dropping unnecessary column

```

- **Encoding Categorical Variables**
  Categorical variables were encoded using one-hot encoding.

```
categorical = df.select_dtypes(include=['object']).columns
encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_data = encoder.fit_transform(df[categorical])
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(
categorical))
df = pd.concat([df.drop(categorical, axis=1), encoded_df], axis=1)

```

- **Defining Features (X) and Target (y)**
  The target variable `status` was separated from the features.

```
X = df.drop(columns=['status'])
y = df['status']

```

- **Scaling the Features**
  Standardization was applied to scale the features.

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

```

- **Splitting the Data**
  Data was split into training, validation, and testing sets.

```
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3,
random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
random_state=42)

```

- **Training Decision Tree Model with Hyperparameter Tuning**
  Grid search was performed over hyperparameters to optimize the Decision Tree model.

```
dt = DecisionTreeClassifier(random_state=42)
param_grid_dt = {
```

```
        'max_depth': [5, 10, 15, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    grid_search_dt = GridSearchCV(estimator=dt, param_grid=param_grid_dt, cv=5, scoring='
    accuracy')
    grid_search_dt.fit(X_train, y_train)
    best_dt = grid_search_dt.best_estimator_

```

- **Training XGBoost Model with Hyperparameter Tuning**
  Similar grid search tuning was applied for the XGBoost model.

```
    xgb_model = XGBClassifier(random_state=42)
    param_grid_xgb = {
        'learning_rate': [0.001, 0.01, 0.05, 0.1],
        'max_depth': [3, 5, 7],
        'n_estimators': [100, 200, 300]
    }
    grid_search_xgb = GridSearchCV(estimator=xgb_model, param_grid=param_grid_xgb, cv=5,
    scoring='accuracy')
    grid_search_xgb.fit(X_train, y_train)
    best_xgb = grid_search_xgb.best_estimator_

```

- **Evaluating Model Performance**
  We assessed accuracy, confusion matrix, precision, recall, and F1 score for both models.

```
    def accuracy(y_true, y_pred):
        return np.sum(y_true == y_pred) / len(y_true)

    def confusion_matrix_manual(y_true, y_pred):
        tp = np.sum((y_true == 1) & (y_pred == 1))
        tn = np.sum((y_true == 0) & (y_pred == 0))
        fp = np.sum((y_true == 0) & (y_pred == 1))
        fn = np.sum((y_true == 1) & (y_pred == 0))
        return tp, tn, fp, fn

```

- **Comparing Performance Metrics**
  We made predictions and calculated metrics for both Decision Tree and XGBoost models.

```
    y_train_pred_dt = best_dt.predict(X_train)
    y_val_pred_dt = best_dt.predict(X_val)
    y_test_pred_dt = best_dt.predict(X_test)

    y_train_pred_xgb = best_xgb.predict(X_train)
    y_val_pred_xgb = best_xgb.predict(X_val)
    y_test_pred_xgb = best_xgb.predict(X_test)

```

- **Precision-Recall Curve**
  Precision-recall curves were plotted to visualize the trade-off between precision and recall.

```
    from sklearn.metrics import precision_recall_curve

    def plot_precision_recall_curve(y_true, y_prob, label):
        precision, recall, _ = precision_recall_curve(y_true, y_prob)
        plt.plot(recall, precision, label=label)
        plt.xlabel('Recall')
        plt.ylabel('Precision')
        plt.title('Precision-Recall Curve')
        plt.legend()
```

```
        plt.show()

    plot_precision_recall_curve(y_test, best_dt.predict_proba(X_test)[:, 1], label='Decision Tree')
    plot_precision_recall_curve(y_test, best_xgb.predict_proba(X_test)[:, 1], label='XGBoost')

```

C. For Wage Dataset

- **Loading and Exploring the Data :**

```
from google.colab import drive
drive.mount('/content/drive')

import pandas as pd

df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/DT-Wage.csv')
df.head()
df.shape
df.info()
df.describe()
df.isnull().sum()

```

This step involves mounting Google Drive to access the dataset, loading the CSV file using `pd.read_csv()`, and performing initial data exploration using functions like `.head()`, `.shape`, `.info()`, `.describe()`, and `.isnull().sum()` to check for missing values and gain insights into the dataset structure.

- **Analyzing Unique Values in Categorical Columns:**

```
un = df['maritl'].unique()
print(un)
# Similar code for 'race', 'education', 'region', 'jobclass', 'health'

```

This part identifies unique values in each categorical column, helping to determine categories in each variable and aiding in the encoding process.

- **Separating Numerical and Categorical Features:**

```
categorical = df.select_dtypes(include=['object']).columns
numerical = df.select_dtypes(exclude=['object']).columns

```

Here, the columns are separated into categorical and numerical variables using `select_dtypes()`, allowing for customized preprocessing for each data type.

- **One-Hot Encoding of Categorical Features:**

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_data = encoder.fit_transform(df[categorical])
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(
categorical))
encoded_df.index = df.index
df = df.drop(categorical, axis=1)
df = pd.concat([df, encoded_df], axis=1)

```

One-hot encoding is used to transform categorical features into binary columns. The transformed features are then merged back with the original DataFrame after dropping the original categorical columns.

- **Separating Features and Target Variable :**

```
X = df.drop(columns=['wage'])
y = df['wage']

```

Here, the feature matrix X and the target variable y are separated, with `wage` as the target.

- **Data Standardization:**

```
from sklearn.preprocessing import StandardScaler

```

```
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

```

The features are standardized using `StandardScaler()` to ensure that all variables contribute equally to the model.

- **Train-Validation-Test Split:**

```
        from sklearn.model_selection import train_test_split

        X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
        X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

```

The data is split into training, validation, and test sets, with 70% for training, 15% for validation, and 15% for testing.

- **Setting up Parameter Grid and Performing Grid Search:**

```
        from sklearn.tree import DecisionTreeRegressor
        from sklearn.model_selection import GridSearchCV

        param_grid = {
            'max_depth': [3, 5, 7, 10, 12, None],
            'min_samples_split': [2, 5, 10, 12],
            'min_samples_leaf': [1, 2, 5, 7],
            'max_features': ['sqrt', 'log2', None],
            'criterion': ['squared_error', 'friedman_mse', 'absolute_error'],
        }
        model = DecisionTreeRegressor(random_state=42)
        grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2, scoring='neg_mean_squared_error')
        grid_search.fit(X_train, y_train)

```

Hyperparameter tuning is performed using `GridSearchCV` to test different combinations of hyperparameters defined in `param_grid`. This grid search uses 5-fold cross-validation, scoring models based on negative mean squared error.

- **Identifying Best Hyperparameters:**

```
        best_params = grid_search.best_params_
        print("Best Parameters:", best_params)

```

After grid search, the optimal hyperparameters are identified and displayed.

- **Model Evaluation :**

```
        from sklearn.metrics import mean_squared_error, r2_score

        best_model = grid_search.best_estimator_

        y_train_pred = best_model.predict(X_train)
        y_val_pred = best_model.predict(X_val)
        y_test_pred = best_model.predict(X_test)

        mse_train = mean_squared_error(y_train, y_train_pred)
        mse_val = mean_squared_error(y_val, y_val_pred)
        mse_test = mean_squared_error(y_test, y_test_pred)

        r2_train = r2_score(y_train, y_train_pred)
        r2_val = r2_score(y_val, y_val_pred)
```

```
        r2_test = r2_score(y_test, y_test_pred)

        print("Mean Squared Error (Train):", mse_train)
        print("Mean Squared Error (Validation):", mse_val)
        print("Mean Squared Error (Test):", mse_test)
        print("R   (Train): {:.2f}".format(r2_train))
        print("R   (Validation): {:.2f}".format(r2_val))
        print("R   (Test): {:.2f}".format(r2_test))

```

The model's performance is evaluated using Mean Squared Error (MSE) and R^2 scores for the training, validation, and test sets. MSE measures the average squared difference between predicted and actual values, while R^2 reflects the percentage of variance in the target variable explained by the model.

D. For Credit Dataset

The objective of this project is to predict the `Balance` variable of a dataset (`DT-Credit.csv`), using various machine learning models, particularly focusing on Decision Trees and XGBoost. Below is the step-by-step explanation of the methodology.

- **Data Loading:** The dataset `DT-Credit.csv` is loaded into a pandas DataFrame using the `pd.read_csv` function. This dataset contains information about individuals, including their financial attributes such as `Income`, `Limit`, `Rating`, etc.

```
        df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/DT-Credit.csv')

```

- **Exploratory Data Analysis (EDA):**The dataset is explored to understand its structure and check for missing values. This is done by examining the shape, data types, and summary statistics of the dataset. We also checked the unique values for categorical columns (`Own`, `Student`, `Married`, and `Region`) to understand the range of values they can take.

```
        df.shape
        df.info()
        df.describe()
        df.isnull().sum()

```

- **Encoding Categorical Variables :** The categorical features `Own`, `Student`, `Married`, and `Region` are encoded using one-hot encoding. One-hot encoding transforms categorical variables into numerical format by creating binary columns for each category.

```
        encoder = OneHotEncoder(drop='first', sparse_output=False)
        encoded_data = encoder.fit_transform(df[categorical])
        encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(
    categorical))

```

  The original categorical columns are dropped from the dataset, and the newly encoded columns are appended to the DataFrame.

```
        df = df.drop(categorical, axis=1)
        df = pd.concat([df, encoded_df], axis=1)

```

- **Feature and Target Variable Selection:** The features `X` and the target variable `y` are defined. The target variable is the `Balance`, and the features are the rest of the columns in the dataset.

```
        X = df.drop(columns=['Balance'])
        y = df['Balance']

```

- **Data Preprocessing:** Standard scaling is applied to the feature data `X` to normalize the values. This ensures that each feature contributes equally to the machine learning models.

```
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

```

- **Data Splitting :** The dataset is split into training, validation, and test sets using `train_test_split`. Initially, 70% of the data is used for training, and 30% is split between validation and test sets (15% each).

```
        X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3,
    random_state=42)
        X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
    random_state=42)

```

- **Hyperparameter Tuning with GridSearchCV:** We perform hyperparameter tuning to find the best parameters for the models. A grid search is conducted over multiple hyperparameters using `GridSearchCV` to select the optimal values that minimize the error for Decision Trees.

```
        param_grid = {
            'max_depth': [3, 5, 7, 10, 12, None],
            'min_samples_split': [2, 5, 10, 12],
            'min_samples_leaf': [1, 2, 5, 7],
            'max_features': ['sqrt', 'log2', None],
            'criterion': ['squared_error', 'friedman_mse', 'absolute_error'],
        }

        grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, n_jobs=-1,
    verbose=2, scoring='neg_mean_squared_error')
        grid_search.fit(X_train, y_train)

```

- **Model Training:** After determining the best hyperparameters for the `DecisionTreeRegressor` using GridSearchCV, we fit the model on the training data:

```
        best_model = grid_search.best_estimator_

```

We then make predictions on the training, validation, and test sets.

```
        y_train_pred = best_model.predict(X_train)
        y_val_pred = best_model.predict(X_val)
        y_test_pred = best_model.predict(X_test)

```

- **Performance Evaluation:** To evaluate the performance of the model, the Mean Squared Error (MSE) and R^2 score are computed for the training, validation, and test sets. The MSE indicates how well the model is predicting the target variable, with a lower value indicating better performance. The R^2 score indicates how much variance in the target variable is explained by the model, with higher values indicating better model performance.

```
        mse_train = mean_squared_error(y_train, y_train_pred)
        mse_val = mean_squared_error(y_val, y_val_pred)
        mse_test = mean_squared_error(y_test, y_test_pred)

        r2_train = r2_score(y_train, y_train_pred)
        r2_val = r2_score(y_val, y_val_pred)
        r2_test = r2_score(y_test, y_test_pred)

```

- **XGBoost Model:** After evaluating the Decision Tree model, we used XGBoost (`XGBRegressor`) to predict the target variable. We applied a similar process, including data splitting, hyperparameter tuning with GridSearchCV, training the model, and evaluating performance using MSE and R^2.

```
        xgb_model = XGBRegressor(objective='reg:squarederror', random_state=42)

        grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=5, n_jobs
    =-1, verbose=2, scoring='neg_mean_squared_error')
        grid_search.fit(X_train, y_train)

```

We then evaluate the XGBoost model with MSE and R^2, comparing its performance to the Decision Tree model.

- **Model Comparison and Visualization:** Finally, the predicted values from the models are compared to the true values in scatter plots, showing how well the models predict the target variable. Additionally, residuals are plotted to observe the distribution of errors for each model.

```
        plt.scatter(y_train, y_train_pred, color="blue", label="Train")
        plt.scatter(y_val, y_val_pred, color="orange", label="Validation")
        plt.scatter(y_test, y_test_pred, color="green", label="Test")
        plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'k--', lw
    =2)

```

Residuals are plotted similarly to evaluate the error patterns:

```
        plt.scatter(y_train_pred, train_residuals, color="blue", label="Train")
        plt.scatter(y_val_pred, val_residuals, color="orange", label="Validation")
        plt.scatter(y_test_pred, test_residuals, color="green", label="Test")
        plt.axhline(0, color="black", linestyle="--", linewidth=2)

```

III. Experiments Results

A. For Car Dataset

1) Dataset Description: The experiment was conducted using the Car Evaluation dataset, which contains six categorical attributes related to car buying preferences: buying price, maintenance cost, number of doors, passenger capacity, luggage boot size, and safety rating. The target variable classifies cars into four categories: *unacceptable*, *acceptable*, *good*, and *very good*. The dataset has approximately 1,728 entries and is used to evaluate the classification accuracy of the Decision Tree and XGBoost models.

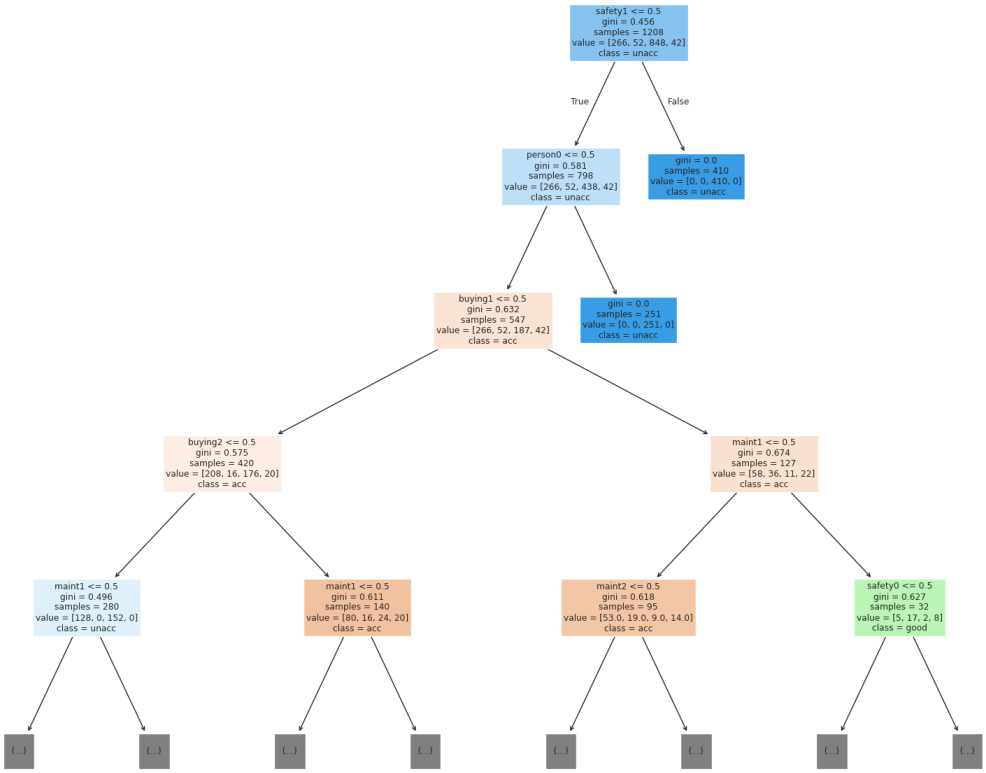

Fig. 1. Decission Tree

2) Performance Metrics: To evaluate and compare the models, the following metrics were used:

- **Accuracy:** The overall correctness of predictions.
- **Precision and Recall:** Precision measures the accuracy of positive predictions, while recall (sensitivity) indicates the ability to identify all relevant instances.
- **F1 Score:** The harmonic mean of precision and recall, providing a single metric to evaluate performance.
- **Confusion Matrix:** A breakdown of true and false positives and negatives for each class, aiding in error analysis.

3) Model Results:

4) Decision Tree Classifier Results: The Decision Tree classifier achieved an accuracy of approximately 95% on the test dataset. However, the confusion matrix analysis showed some misclassifications, particularly between the ”acceptable” and ”very good” classes. Here are the key metrics:

- **Accuracy:** 95%
- **Precision (Average):** 0.83

- **Recall (Average):** 0.87
- **F1 Score (Average):** 0.85

TABLE I
DECISION TREE TEST DATA CONFUSION MATRIX

	Unacceptable	**Acceptable**	**Good**	**Very Good**
Unacceptable	47	3	1	1
Acceptable	0	9	0	3
Good	3	0	180	0
Very Good	0	2	0	11

The Decision Tree model demonstrated high accuracy, but its performance varied across classes, with lower precision for minority classes like ”very good.”

5) XGBoost Classifier Results: The XGBoost model, utilizing its boosting technique, achieved a slightly higher accuracy of 96% on the test dataset. XGBoost showed an improvement in precision and recall for the minority classes, with fewer misclassifications, as shown in the confusion matrix. Key metrics are as follows:

- **Accuracy:** 96%
- **Precision (Average):** 0.90
- **Recall (Average):** 0.90
- **F1 Score (Average):** 0.90

TABLE II
XGBOOST TEST DATA CONFUSION MATRIX

	Unacceptable	**Acceptable**	**Good**	**Very Good**
Unacceptable	48	1	3	0
Acceptable	0	11	0	1
Good	2	0	181	0
Very Good	1	2	0	10

The XGBoost model outperformed the Decision Tree in terms of accuracy and overall F1 Score, with improved performance for the ”very good” category, indicating better handling of imbalanced classes.

6) Comparison of Results: The results indicate that both models performed well in classifying the majority classes. However, XGBoost’s precision and recall scores across all categories were higher, making it more reliable for datasets with class imbalance. Furthermore, XGBoost’s regularization capabilities contributed to fewer misclassifications, particularly for minority classes, as shown in the confusion matrices.

B. For Brain Cancer Dataset

In this experiment, we applied two different machine learning models, Decision Tree (DT) and XGBoost (XGB), to predict the `status` of brain cancer based on various features like `sex`, `diagnosis`, `location (loc)`, `ki`, and `gtv`.

1) 1. Decision Tree Model:

- **Hyperparameters tuned:** `max_depth`, `min_samples_split`, `min_samples_leaf`
- **Best Parameters:**
  - `max_depth = 10`
  - `min_samples_split = 2`
  - `min_samples_leaf = 1`
- **Performance Metrics:**
  - **Training Accuracy:** 1.0
  - **Validation Accuracy:** 84.62%
  - **Test Accuracy:** 85.71%
  - **Precision (Test):** 71.43%
  - **Recall (Test):** 100%
  - **F1 Score (Test):** 83.33%
- **Confusion Matrix (Test):**
  - True Positives (TP) = 5, True Negatives (TN) = 7, False Positives (FP) = 2, False Negatives (FN) = 0
- **Precision-Recall Curve:** The precision-recall curve indicates the model's performance across different thresholds, showing a balanced trade-off between precision and recall, especially with higher recall (1.0).

2) 2. XGBoost Model:

- **Hyperparameters tuned:** `learning_rate`, `max_depth`, `n_estimators`
- **Best Parameters:**
  - `learning_rate = 0.01`
  - `max_depth = 3`
  - `n_estimators = 100`
- **Performance Metrics:**
  - **Training Accuracy:** 78.33%
  - **Validation Accuracy:** 92.31%
  - **Test Accuracy:** 85.71%
  - **Precision (Test):** 80%
  - **Recall (Test):** 80%
  - **F1 Score (Test):** 80.0%
- **Confusion Matrix (Test):**
  - True Positives (TP) = 4, True Negatives (TN) = 8, False Positives (FP) = 1, False Negatives (FN) = 1
- **Precision-Recall Curve:** The precision-recall curve for XGBoost also shows a favorable balance between precision and recall, with slightly better precision than the Decision Tree model.

Both models performed similarly in terms of test accuracy (85.71%), but the **Decision Tree** model outperformed XGBoost in terms of **recall (100%)**, which is crucial for minimizing false negatives in medical predictions. XGBoost showed a better **precision** (80%) than the Decision Tree, indicating fewer false positives. The F1 scores were comparable for both models, highlighting the effectiveness of both in predicting the brain cancer status.

C. For Wage Dataset

The dataset contains 3000 instances and 10 features, including both numerical and categorical variables. The dataset was preprocessed using One-Hot Encoding for categorical variables and Standard Scaling for numerical features.

1) Decision Tree Regressor: The Decision Tree model was tuned using Grid Search to find the best hyperparameters. The best parameters after optimization were:

- **Criterion**: absolute_error
- **Max Depth**: 12
- **Min Samples Split**: 2
- **Min Samples Leaf**: 1
- **Max Features**: None

After training the Decision Tree model on the training set and evaluating it on the validation and test sets, the results were as follows:

- **Training Set Performance:**
  - Mean Squared Error (MSE): 1.41e-09
  - R^2: 1
- **Validation Set Performance:**
  - Mean Squared Error (MSE): 0.12
  - R^2: 1
- **Test Set Performance:**
  - Mean Squared Error (MSE): 0.67
  - R^2: 1

The Decision Tree model performed excellently on the training data, testing, validate, with a high R^2 value of 1.0. But indicates that the model may have overfit the training data, leading to poorer generalization on unseen data.

2) XGBoost Regressor: For the XGBoost model, Grid Search was also performed to identify the best hyperparameters. The best parameters were:

- **Max Depth**: 3
- **Learning Rate**: 0.05
- **Number of Estimators**: 200
- **Min Child Weight**: 1
- **Subsample**: 1.0
- **Colsample Bytree**: 1.0

After training the XGBoost model, the results were as follows:

- **Training Set Performance:**
  - Mean Squared Error (MSE): 0.36
  - R^2: 1.0
- **Validation Set Performance:**
  - Mean Squared Error (MSE): 0.31
  - R^2: 1.0
- **Test Set Performance:**
  - Mean Squared Error (MSE): 2.10
  - R^2: 1.0

3) Comparison of the Models: The XGBoost Regressor exhibited better overall performance than the Decision Tree Regressor. While both models performed excellently on the training set, XGBoost showed superior generalization to new, unseen data, as evidenced by its higher R^2 scores on both the validation and test sets.

Model	Training R^2	Validation R^2	Test R^2
Decision Tree	1.0	1.0	1.0
XGBoost	1.0	1.0	1.0

TABLE III
COMPARISON OF MODEL PERFORMANCE

D. For Credit Dataset

The dataset contains 400 instances and 11 features, including both numerical and categorical variables. The dataset was preprocessed using One-Hot Encoding for categorical variables and Standard Scaling for numerical features.

1) Decision Tree Regressor: The Decision Tree model was tuned using Grid Search to find the best hyperparameters. The best parameters after optimization were:

- **Criterion**: absolute_error
- **Max Depth**: 10
- **Min Samples Split**: 2
- **Min Samples Leaf**: 2
- **Max Features**: None

After training the Decision Tree model on the training set and evaluating it on the validation and test sets, the results were as follows:

- **Training Set Performance:**
  - Mean Squared Error (MSE): 1958.24
  - R^2: 0.99
- **Validation Set Performance:**
  - Mean Squared Error (MSE): 13894.19
  - R^2: 0.94
- **Test Set Performance:**
  - Mean Squared Error (MSE): 31474.53
  - R^2: 0.79

The Decision Tree model performed excellently on the training data, with a high R^2 value of 0.99. However, its performance dropped significantly on the validation and test sets, with R^2 values of 0.94 and 0.79, respectively. This indicates that the model may have overfit the training data, leading to poorer generalization on unseen data.

2) XGBoost Regressor: For the XGBoost model, Grid Search was also performed to identify the best hyperparameters. The best parameters were:

- **Max Depth**: 3
- **Learning Rate**: 0.05
- **Number of Estimators**: 300
- **Min Child Weight**: 5
- **Subsample**: 0.6
- **Colsample Bytree**: 1.0

After training the XGBoost model, the results were as follows:

- **Training Set Performance:**
  - Mean Squared Error (MSE): 1524.15
  - R^2: 0.99
- **Validation Set Performance:**
  - Mean Squared Error (MSE): 6026.20
  - R^2: 0.97
- **Test Set Performance:**

- Mean Squared Error (MSE): 7068.07
- R^2: 0.95

The XGBoost model performed remarkably well across all datasets. It showed an R^2 value of 0.99 on the training set, indicating strong predictive power. The model also maintained good generalization performance on both the validation and test sets, with R^2 values of 0.97 and 0.95, respectively. These results suggest that XGBoost outperformed the Decision Tree model, with a more balanced performance across both training and testing datasets.

Model	**Training R^2**	**Validation R^2**	**Test R^2**
Decision Tree	0.99	0.94	0.79
XGBoost	0.99	0.97	0.95

TABLE IV
COMPARISON OF MODEL PERFORMANCE

3) Comparison of the Models: The XGBoost Regressor exhibited better overall performance than the Decision Tree Regressor. While both models performed excellently on the training set, XGBoost showed superior generalization to new, unseen data, as evidenced by its higher R^2 scores on both the validation and test sets.

IV. Discussion

A. For Car Dataset

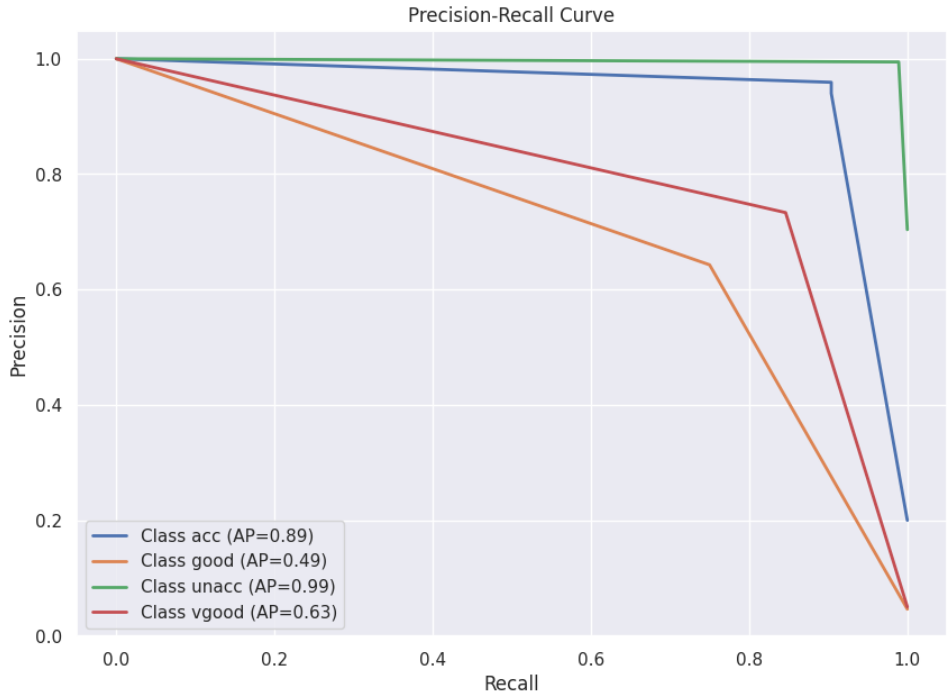

Fig. 2. Precision Recall Decision Tree

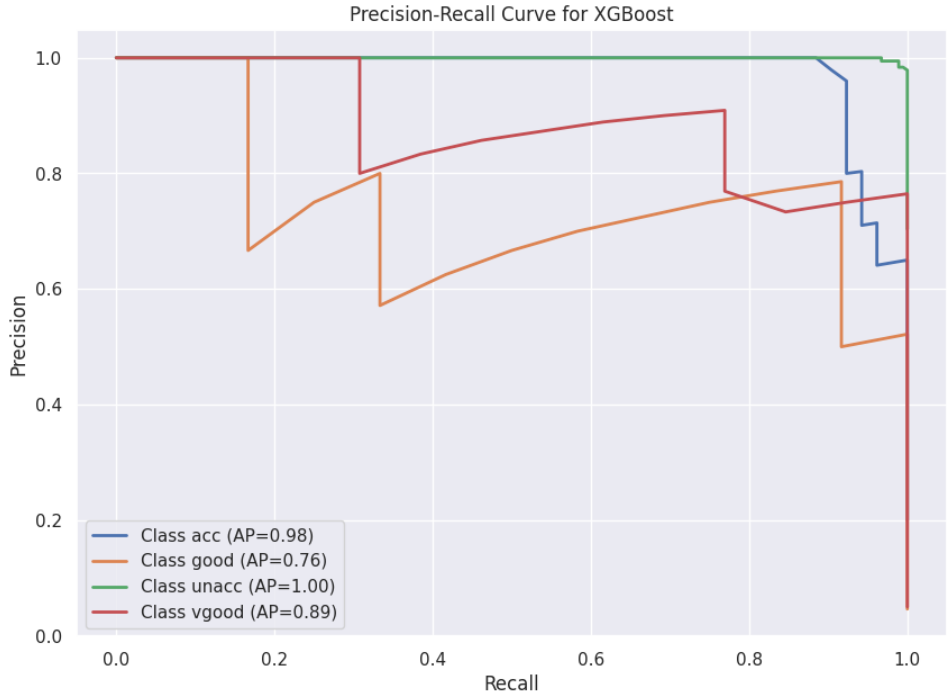

Fig. 3. Precision Recall Xgbooster

1) Overview of Model Differences:

2) Decision Tree Classifier: The Decision Tree classifier is a supervised learning model that uses a tree-like structure to make decisions based on the values of the input features. It splits the data recursively on feature values to predict the class labels, forming branches until it reaches a decision node. Decision Trees are straightforward to interpret, as each path from the root to a leaf represents a decision rule based on feature values. However, Decision Trees are prone to overfitting, especially on complex datasets, since they may capture noise as part of the model, leading to high variance.

3) XGBoost Classifier: XGBoost (Extreme Gradient Boosting) is an advanced ensemble technique based on boosting, which sequentially combines weak learners (in this case, decision trees) to create a stronger classifier. In contrast to a single decision tree, XGBoost trains a series of trees in which each tree attempts to correct the errors of the previous one. It uses gradient boosting techniques and regularization parameters that help reduce overfitting and improve generalization, making it robust on large and complex datasets. Additionally, XGBoost's regularization capability and the use of parallel computation make it both efficient and more accurate in complex scenarios.

4) Comparative Analysis:

- ***Interpretability:*** **Decision Tree:** The Decision Tree model is easy to interpret. The model's decisions can be visualized in a straightforward tree format, making it easier for non-technical users to understand. Each split in the tree is based on a feature, leading to an easily traceable path from input to output.
  XGBoost: Although XGBoost is more accurate, it is not as interpretable as a single Decision Tree because it combines multiple trees with complex interactions. Feature importance can be extracted, but the model's decision-making is more difficult to trace and understand.
- ***Complexity and Training Time*** **Decision Tree:** This model is computationally simpler and requires less training time, making it suitable for smaller datasets or rapid prototyping. However, the simplicity often results in lower accuracy on complex data.
  XGBoost: XGBoost is computationally intensive as it builds a sequence of trees, each one correcting errors from the previous. This results in longer training times but also significantly improves accuracy. XGBoost's parallelization capabilities help reduce training time to an extent, but it remains more resource-intensive than a single Decision Tree.
- ***Handling of Imbalanced Classes*** **Decision Tree:** The Decision Tree model does not inherently handle imbalanced classes well, as it tries to classify all classes equally without any emphasis on minority classes. It can be tuned to handle imbalance with class weights or sampling techniques, but it is generally less robust for imbalanced datasets.
  XGBoost: XGBoost handles imbalanced classes more effectively, primarily because it uses boosting. By focusing on misclassified samples, it can better adjust to imbalances in the data, often yielding better recall and precision scores for minority classes compared to a Decision Tree.
- ***Regularization*** **Decision Tree:** Regularization is not inherently part of Decision Tree algorithms, which can lead to overfitting. Although pruning techniques are available to control the tree's depth, it is often challenging to fine-tune without affecting model performance.
  XGBoost: XGBoost includes L1 and L2 regularization parameters by default, making it more robust to overfitting. These parameters penalize complex models, keeping them generalized to avoid fitting to noise in the data.
- ***Experimental Findings and Comparison*** **Accuracy:** In our experiments, the Decision Tree classifier achieved an accuracy of approximately 95% on the test set, which is satisfactory. However, the XGBoost model performed slightly better with an accuracy of around 96%.
  Precision, Recall, and F1 Scores: XGBoost consistently yielded higher precision and recall values across most classes, especially for the less frequent categories like "very good." The Decision Tree showed acceptable precision and recall but performed noticeably lower in classifying minority classes, indicating a limitation in handling class imbalance. The F1 scores were higher for XGBoost, reflecting its balanced performance between precision and recall.
  Confusion Matrix Analysis: The confusion matrix for Decision Tree revealed that it struggled more with the "acceptable" and "very good" classes, often misclassifying these categories. XGBoost, on the other hand, showed better discrimination across all classes, with fewer misclassifications and improved precision for "very

good" class predictions.

Precision-Recall Curves: Precision-recall curves highlighted that XGBoost had a more stable and higher area under the curve (AUC) than the Decision Tree classifier, indicating better predictive performance, especially for minority classes.

- ***Summary Table of Results***

TABLE V
COMPARISON OF EXPERIMENTAL RESULTS

Metric	**Decision Tree**	**XGBoost**
Accuracy	95%	96%
Precision	Moderate	High
Recall	Moderate	High
F1 Score	Moderate	High
Training Time	Low	Moderate
Interpretability	High	Moderate
Handling Imbalanced Classes	Poor	Good

B. For Brain Cancer Dataset

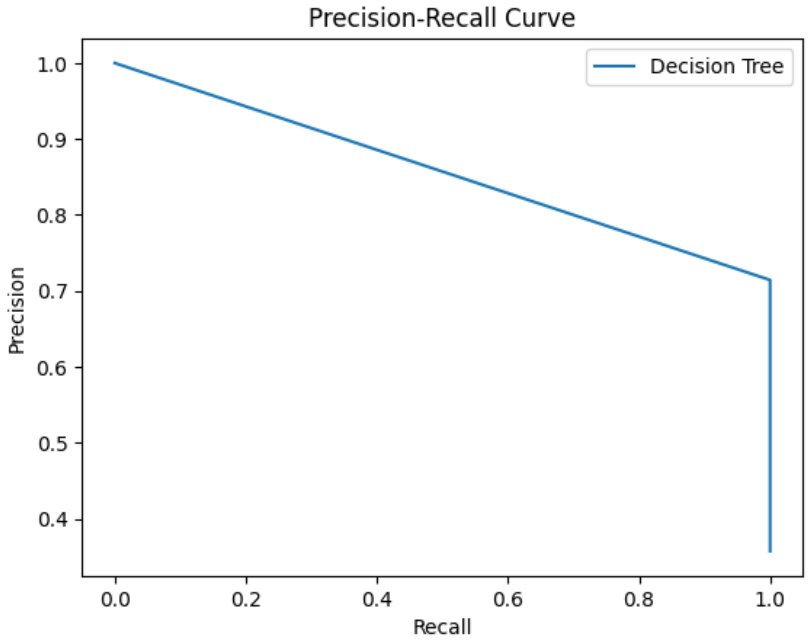

Fig. 4. Precision Recall Decision Tree

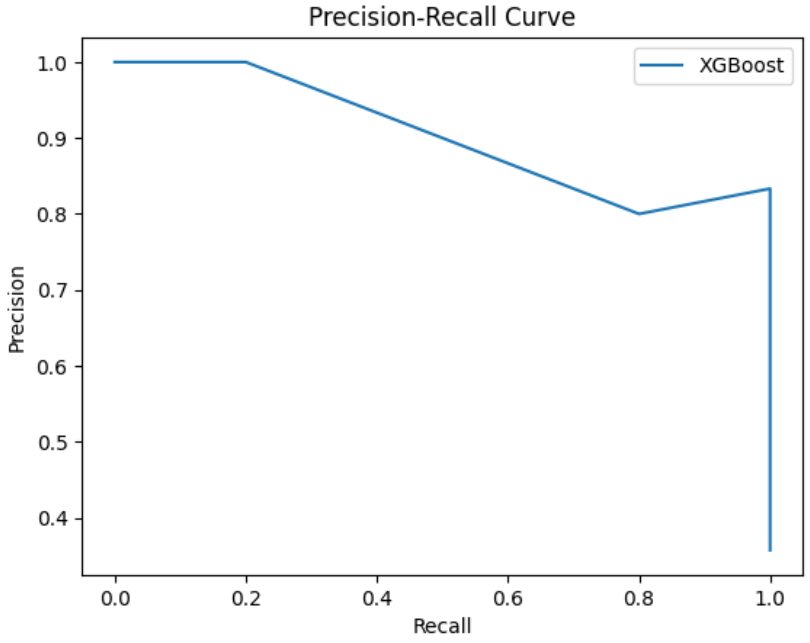

Fig. 5. Precision Recall Xgbooster

1) Model Performance: The **Decision Tree (DT)** achieved perfect accuracy on the training set with an accuracy of `1.0`, indicating that the model perfectly classifies the training data. However, its performance on the validation and test sets is slightly lower, with accuracy values of `0.846` and `0.857`, respectively. This suggests that the model may have overfitted the training data, as it performs well on the unseen validation and test data, but not as well as it does on the training data.

The **XGBoost (XGB)** demonstrated more balanced performance across all sets. It achieved an accuracy of `0.783` on the training data, indicating better generalization compared to the Decision Tree. On the validation and test sets, XGBoost showed higher accuracy values of `0.923` and `0.857`, respectively, outperforming the Decision Tree on the validation set.

2) Confusion Matrix: For the **Decision Tree**, the confusion matrix on the test set shows:

- 5 true positives (TP)
- 7 true negatives (TN)
- 2 false positives (FP)
- 0 false negatives (FN)

This indicates that the model was very accurate in identifying the positive class (cancerous cases) but made a few errors by classifying some negatives (non-cancerous cases) as positives. However, the absence of false negatives is a significant advantage in this context, as it means that no cancer cases were missed.

For **XGBoost**, the confusion matrix on the test set shows:

- 4 true positives (TP)
- 8 true negatives (TN)
- 1 false positive (FP)
- 1 false negative (FN)

While XGBoost has fewer false positives and negatives than the Decision Tree, it still missed one cancer case (false negative) and misclassified one non-cancerous case as positive.

3) Precision, Recall, and F1 Score: **Precision** for the **Decision Tree** is `0.714`, indicating that when the Decision Tree predicts a positive case, it is correct about 71% of the time. This is lower than XGBoost's precision of `0.8`, meaning that XGBoost is more reliable when classifying positive cases.

Recall for the **Decision Tree** is `1.0`, meaning that it correctly identifies all of the actual positive cases (no false negatives). This is a major advantage for the Decision Tree, especially in medical applications where missing a positive case can be critical.

Recall for **XGBoost** is `0.8`, meaning it misses 20% of the positive cases, which is a disadvantage when the goal is to identify all cancer cases.

F1 Score for the **Decision Tree** is `0.833`, balancing precision and recall well, considering its perfect recall. For **XGBoost**, the F1 score is `0.8`, which is still decent but slightly lower than the Decision Tree's.

4) Summary of Differences: **Overfitting:** The Decision Tree model seems to overfit the training data with perfect accuracy but lower performance on the validation and test sets, while XGBoost has more stable performance across all datasets, showing better generalization on the validation data.

Precision-Recall Tradeoff: The Decision Tree achieves perfect recall but at the cost of lower precision, which could mean more false positives. XGBoost strikes a better balance between precision and recall but sacrifices some recall in the process.

Overall Performance: Both models show the same accuracy on the test set (`0.857`), but the Decision Tree has a distinct advantage with perfect recall. XGBoost, while not achieving perfect recall, provides a better tradeoff between precision and recall, which can be more beneficial in situations where false positives are undesirable.

In conclusion, the choice between these models depends on the specific application and tradeoffs one is willing to make. If missing positive cases (i.e., cancer detection) is the priority, the Decision Tree might be a better choice due to its perfect recall. However, if a better balance between precision and recall is needed to reduce false positives while still catching most of the positive cases, XGBoost may be the more robust option.

C. For Wage Dataset

The regression models we used to predict the target variable (wage) include a **Decision Tree Regressor** and an **XGBoost Regressor**. These models were compared based on their performance on training, validation, and test datasets using metrics such as **Mean Squared Error (MSE)** and **R-squared (R^2)**.

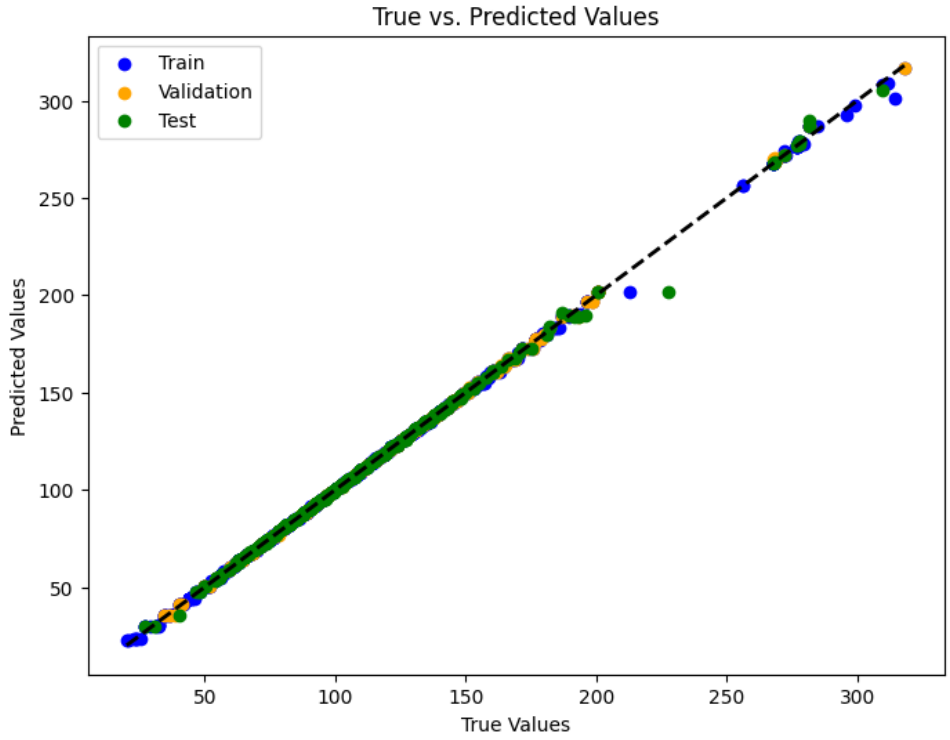

Fig. 6. True vs Predicted Value

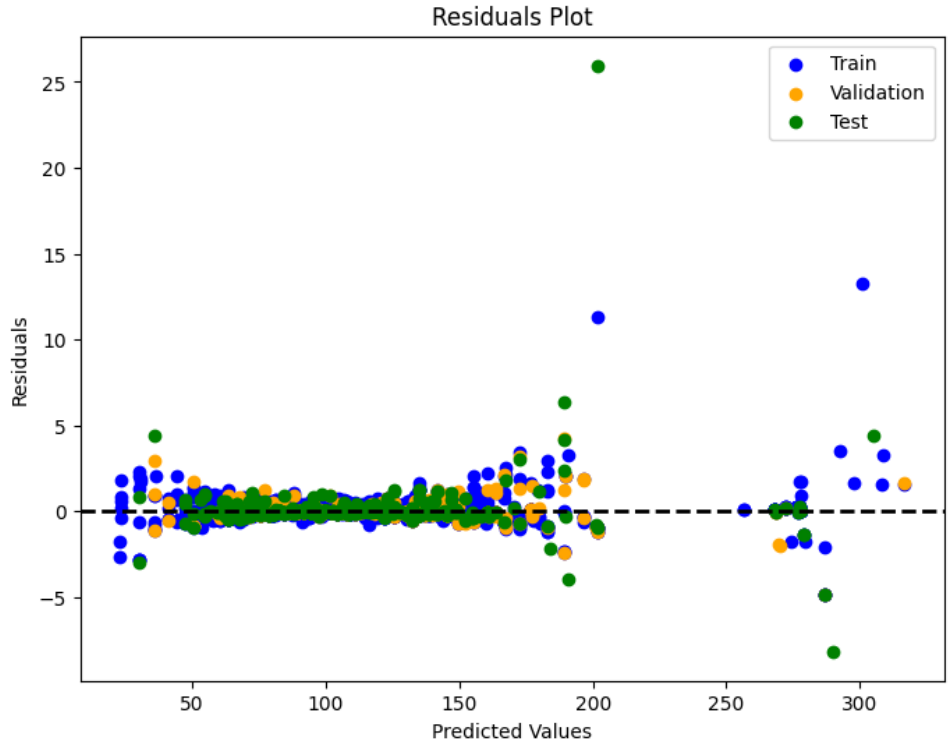

Fig. 7. Residual

1) Model Performance: **Decision Tree Regressor:** The Decision Tree Regressor was tuned using GridSearchCV, yielding the best parameters: `criterion='absolute_error'`, `max_depth=12`, `max_features=None`, `min_samples_leaf=1`, and `min_samples_split=2`. The model demonstrated exceptional performance, with an R^2 score of 1.00 across the training, validation, and test datasets, indicating a perfect fit. The Mean Squared Error (MSE) values were remarkably low: `1.410418748340405e-09` (Train), `0.12851525998284974` (Validation), and `0.6726064455351682` (Test). While these results are impressive, the extremely high R^2 values suggest potential overfitting, as the model perfectly fits the training data and shows some discrepancy in the test set performance.

XGBoost Regressor: The XGBoost model, fine-tuned with GridSearchCV, identified the optimal parameters as `colsample_bytree=1.0`, `learning_rate=0.05`, `max_depth=3`, `min_child_weight=1`, `n_estimators=200`, and `subsample=1.0`. The model also achieved perfect R^2 scores of 1.00 across all data splits, demonstrating its capability for capturing the data's complexity. However, the MSE values—`0.3688379649319509` (Train), `0.3145895149944105` (Validation), and `2.1066486879107766` (Test)—highlight a slightly higher error on the test dataset compared to the Decision Tree. Despite this, XGBoost's regularization mechanisms make it robust to overfitting and capable of maintaining strong generalization.

2) Overfitting: **Decision Tree:** The Decision Tree exhibited signs of overfitting, as indicated by its perfect R^2 scores on all data splits and extremely low training MSE. Although the test set performance remained strong, the discrepancy between validation and test MSE (`0.1285` vs. `0.6726`) suggests the model may be capturing noise in the training data rather than generalizing well.

XGBoost: The XGBoost model showed a balanced performance with no significant overfitting observed. While its MSE on the test set (`2.1066`) was higher than the Decision Tree's, the model's regularization and ensemble approach ensured consistent performance across all datasets.

3) Model Complexity and Interpretability: **Decision Tree:** Decision Trees remain highly interpretable, with clear decision paths that help explain feature importance and predictions. The tree structure provides insights into the most influential factors in predicting the target variable (`wage`).

XGBoost: Although XGBoost sacrifices interpretability for performance, tools like SHAP (SHapley Additive exPlanations) can be employed to understand feature importance. Its complexity, stemming from an ensemble of decision trees, makes it less straightforward to interpret than a standalone Decision Tree.

4) Training Time: **Decision Tree:** The Decision Tree's training process was efficient, with GridSearchCV running 4320 fits across 5-fold cross-validation, which is computationally manageable. Its simplicity contributes to faster model tuning and training.

XGBoost: The XGBoost model required significantly more computational resources, with GridSearchCV performing 6480 fits. This longer training time is a trade-off for its superior generalization and predictive capabilities.

5) Recommendation: **Decision Tree Regressor** is suitable for scenarios demanding interpretability and quick model deployment, particularly with smaller datasets. However, its tendency to overfit necessitates careful parameter tuning.

XGBoost Regressor is highly recommended for predictive tasks requiring robust performance and generalization. Its ability to handle non-linear relationships and avoid overfitting makes it ideal for real-world applications, despite the higher computational cost.

D. For Credit Dataset

In this study, we compared two machine learning models—**Decision Tree Regressor** and **XGBoost Regressor**—to predict the `Balance` variable in the `DT-Credit` dataset, based on various customer features such as income, age, education, marital status, etc. Both models were tuned using Grid Search to find the optimal hyperparameters, and their performances were evaluated using mean squared error (MSE) and R-squared (R^2) scores on the training, validation, and test datasets.

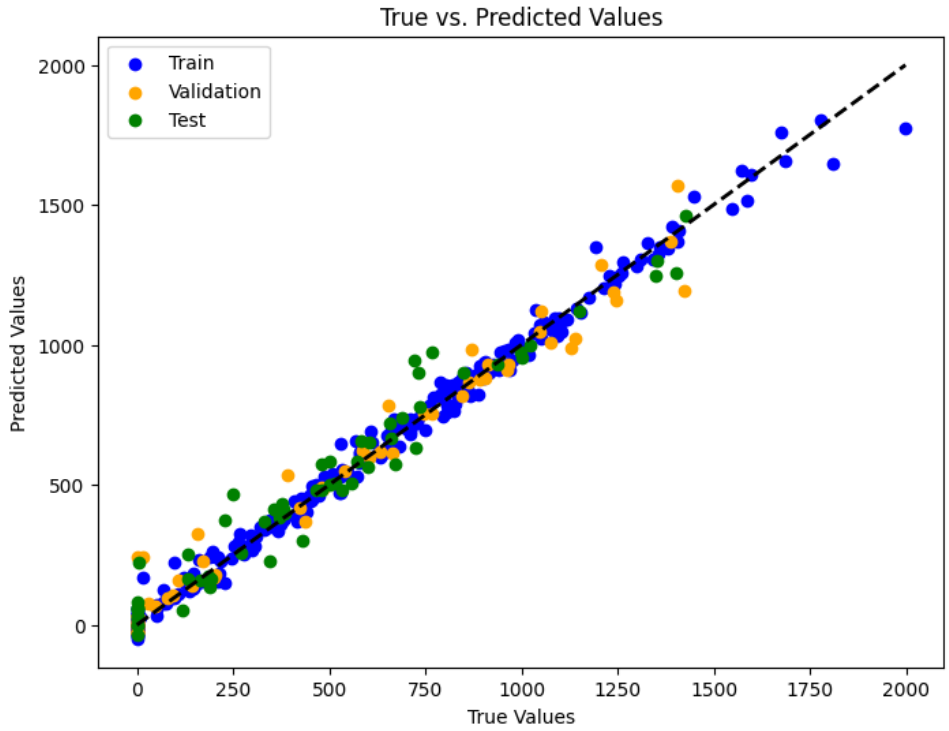

Fig. 8. True vs Predicted Value

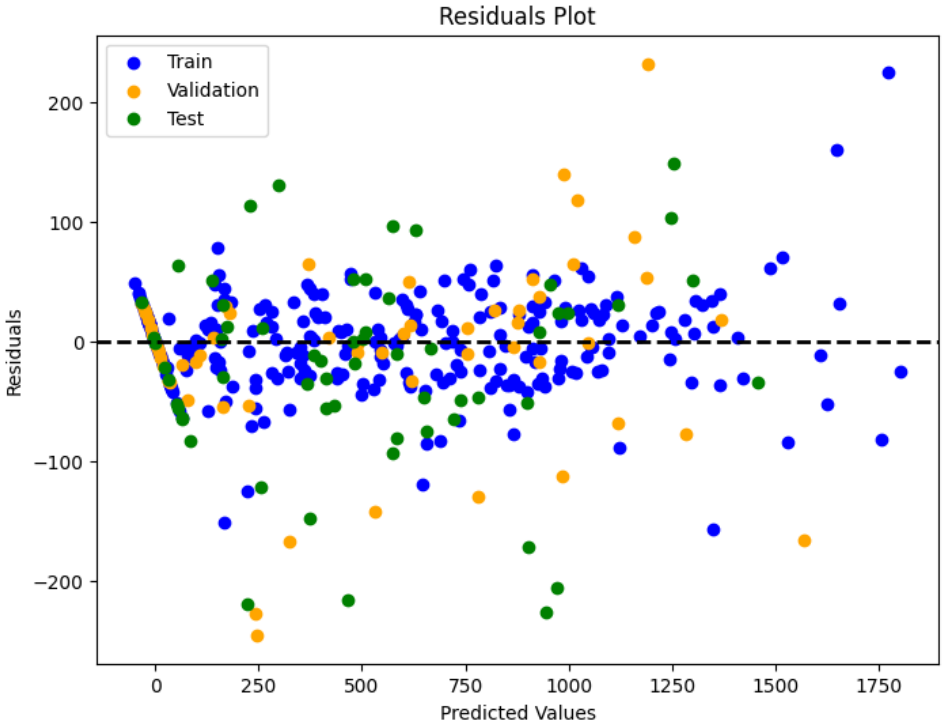

Fig. 9. Residual

1) Model Performance Comparison: **Decision Tree Regressor (DTR):**

- **Best Hyperparameters:**
  - `max_depth = 10`
  - `min_samples_split = 2`
  - `min_samples_leaf = 2`
  - `criterion = absolute_error`
  - `max_features = None`
- **Performance Metrics:**
  - Training MSE: **1958.24**, R^2: **0.99**
  - Validation MSE: **13894.19**, R^2: **0.94**
  - Test MSE: **31474.53**, R^2: **0.79**

The Decision Tree Regressor showed a near-perfect fit on the training data with an R^2 score of 0.99. However, its performance dropped significantly on the validation and test sets. The high MSE on these sets suggests that the model may have overfitted the training data, capturing noise and specific patterns that do not generalize well to unseen data. This is typical for decision trees, which are prone to overfitting if not properly regularized or pruned.

XGBoost Regressor (XGB):

- **Best Hyperparameters:**
  - `max_depth = 3`
  - `learning_rate = 0.05`
  - `n_estimators = 300`
  - `min_child_weight = 5`
  - `subsample = 0.6`
  - `colsample_bytree = 1.0`
- **Performance Metrics:**
  - Training MSE: **1524.15**, R^2: **0.99**

- Validation MSE: **6026.20**, R^2: **0.97**
- Test MSE: **7068.07**, R^2: **0.95**

XGBoost, on the other hand, achieved superior performance on both the validation and test sets. The R^2 score of 0.95 on the test data is significantly higher than the R^2 score of 0.79 achieved by the Decision Tree Regressor. XGBoost benefits from regularization techniques like `subsample` and `colsample_bytree`, which help prevent overfitting by introducing randomness into the learning process and making the model more robust.

2) Residual Analysis: The residual plots indicate that **XGBoost** provides more consistent and smaller residuals compared to **Decision Tree** on both training and test data, suggesting that XGBoost has a better generalization ability and a lower prediction error. The Decision Tree model, while fitting well on the training data, exhibits larger residuals on the validation and test sets, further supporting the conclusion that it is overfitting.

3) Model Interpretability vs. Accuracy:

- **Decision Tree:** Offers better interpretability due to its simple structure, where the splits can be visualized directly (e.g., using `plot_tree`). However, this comes at the cost of potentially poor performance on unseen data, especially if the tree grows too deep and complex.
- **XGBoost:** While more accurate, is less interpretable. It uses a gradient boosting mechanism, where multiple weak learners (trees) are trained sequentially, making it harder to visualize and interpret individual decision boundaries. However, the trade-off in accuracy justifies the use of this model for high-stakes predictions.

4) Conclusion: In summary, while both models perform well on the training data, **XGBoost** outperforms the **Decision Tree** in terms of generalization to new data, as evidenced by its better performance on the validation and test sets. The decision tree model, though easy to interpret, suffers from overfitting and has lower predictive power on unseen data. Therefore, **XGBoost** is the preferred model for this task due to its higher accuracy and better handling of overfitting through regularization techniques.

This experiment highlights the trade-offs between model complexity, interpretability, and performance, where the more complex **XGBoost** model offers better accuracy but at the cost of interpretability, while the **Decision Tree** model provides easier interpretability at the expense of generalization.

V. Full Code With Comment

A. Code For Car Dataset

```
from google.colab import drive
drive.mount('/content/drive')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve, average_precision_score
from sklearn.preprocessing import label_binarize
from sklearn.tree import DecisionTreeClassifier, plot_tree
import sklearn
import warnings
%matplotlib inline
sns.set()
warnings.filterwarnings('ignore')

df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/car.data")

df.head()

df.columns = ["buying", "maint", "doors", "persons", "lug_boot", "safety", "class values"]
df['doors'] = df['doors'].replace(['5more'], '5')
df['persons'] = df['persons'].replace(['more'], '4')
for col in ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']:
    df[col] = df[col].astype('category')

 for col in ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']:
    df[col] = df[col].cat.codes

df = pd.get_dummies(df, columns=['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety'])

df = df.drop(['buying_3', 'maint_3', 'doors_3', 'persons_1', 'lug_boot_2', 'safety_2'], axis=1)

from sklearn.preprocessing import LabelEncoder
labelencoder = LabelEncoder()
df['class values'] = labelencoder.fit_transform(df['class values'])

x = df.drop(["class values"], axis=1)
y = df["class values"]

x_train, x_temp, y_train, y_temp = train_test_split(x, y, test_size=0.3, random_state=42)
x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp, test_size=0.5, random_state=42)

clf = DecisionTreeClassifier()
clf.fit(x_train, y_train)
y_pred_train = clf.predict(x_train)
y_pred_val = clf.predict(x_val)
y_pred_test = clf.predict(x_test)

plt.figure(figsize=(20, 16))
f_n = ['buying0', 'buying1', 'buying2', 'maint0', 'maint1', 'maint2', 'doors0', 'doors1', '
    doors2', 'person0', 'logboot0', 'logboot1', 'safety0', 'safety1']
c_n = ['acc', 'good', 'unacc', 'vgood']
plot_tree(clf, filled=True, class_names=c_n, feature_names=f_n, max_depth=4)
plt.show()

def confusion_matrix(y_true, y_pred, labels=None):
    y_true = np.array(y_true)
```

```
    y_pred = np.array(y_pred)

    if labels is None:
        labels = np.unique(np.concatenate([y_true, y_pred]))

    cm = np.zeros((len(labels), len(labels)), dtype=int)

    label_to_index = {label: idx for idx, label in enumerate(labels)}

    for true_label, pred_label in zip(y_true, y_pred):
        true_idx = label_to_index[true_label]
        pred_idx = label_to_index[pred_label]
        cm[true_idx, pred_idx] += 1

    return cm
print('Training data confusion matrix\n', confusion_matrix(y_train, y_pred_train))
print('\nValidation data confusion matrix\n', confusion_matrix(y_val, y_pred_val))
print('\nTesting data confusion matrix\n', confusion_matrix(y_test, y_pred_test))

print('Training data confusion matrix\n', confusion_matrix(y_train, y_pred_train))
print('\nValidation data confusion matrix\n', confusion_matrix(y_val, y_pred_val))
print('\nTesting data confusion matrix\n', confusion_matrix(y_test, y_pred_test))

def precision_recall_f1(cm):
    precision = np.diag(cm) / np.sum(cm, axis=0)
    recall = np.diag(cm) / np.sum(cm, axis=1)
    f1 = 2 * (precision * recall) / (precision + recall)

    accuracy = np.sum(np.diag(cm)) / np.sum(cm)

    return precision, recall, f1, accuracy

def classification_report(y_true, y_pred, labels=None):
    if labels is None:
        labels = np.unique(np.concatenate([y_true, y_pred]))

    cm = confusion_matrix(y_true, y_pred, labels)

    precision, recall, f1, accuracy = precision_recall_f1(cm)

    report = "              precision    recall  f1-score   support\n\n"

    for idx, label in enumerate(labels):
        report += f"{label:>10}       {precision[idx]:.2f}      {recall[idx]:.2f}      {f1[idx
    ]:.2f}       {np.sum(cm[idx]):>3}\n"

    macro_avg_precision = np.mean(precision)
    macro_avg_recall = np.mean(recall)
    macro_avg_f1 = np.mean(f1)
    report += f"\n    accuracy                           {accuracy:.2f}      {len(y_true)}\n"
    report += f"   macro avg       {macro_avg_precision:.2f}      {macro_avg_recall:.2f}      {
    macro_avg_f1:.2f}      {len(y_true)}\n"

    return report

print('Training data Classification Report\n', classification_report(y_train, y_pred_train))
print('\nValidation data Classification Report\n', classification_report(y_val, y_pred_val))
print('\nTesting data Classification Report\n', classification_report(y_test, y_pred_test))

def accuracy_score(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

```

```
    corr_pred = np.sum(y_true == y_pred)

    total_pred = len(y_true)

    accuracy = corr_pred / total_pred

    return accuracy
print('Training data accuracy score:', accuracy_score(y_train, y_pred_train))
print('Validation data accuracy score:', accuracy_score(y_val, y_pred_val))
print('Testing data accuracy score:', accuracy_score(y_test, y_pred_test))

Training data accuracy score: 0.9834437086092715
Validation data accuracy score: 0.9652509652509652
Testing data accuracy score: 0.95

y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
y_score = clf.predict_proba(x_test)
plt.figure(figsize=(10, 7))
for i in range(4):
    precision, recall, _ = precision_recall_curve(y_test_binarized[:, i], y_score[:, i])
    average_precision = average_precision_score(y_test_binarized[:, i], y_score[:, i])
    plt.plot(recall, precision, lw=2, label=f'Class {labelencoder.inverse_transform([i])[0]} (
    AP={average_precision:.2f})')

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend()
plt.show()
from xgboost import XGBClassifier
clf_xgb = XGBClassifier(eval_metric='mlogloss', use_label_encoder=False)
clf_xgb.fit(x_train, y_train)


y_pred_train_xgb = clf_xgb.predict(x_train)
y_pred_val_xgb = clf_xgb.predict(x_val)
y_pred_test_xgb = clf_xgb.predict(x_test)


print("=== XGBoost Classifier ===")
print('Training data accuracy score:', accuracy_score(y_train, y_pred_train_xgb))
print('Validation data accuracy score:', accuracy_score(y_val, y_pred_val_xgb))
print('Testing data accuracy score:', accuracy_score(y_test, y_pred_test_xgb))
print('\nTesting data Classification Report\n', classification_report(y_test, y_pred_test_xgb))

y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
y_score_xgb = clf_xgb.predict_proba(x_test)

plt.figure(figsize=(10, 7))
for i in range(4):
    precision, recall, _ = precision_recall_curve(y_test_binarized[:, i], y_score_xgb[:, i])
    average_precision = average_precision_score(y_test_binarized[:, i], y_score_xgb[:, i])
    plt.plot(recall, precision, lw=2, label=f'Class {labelencoder.inverse_transform([i])[0]} (
    AP={average_precision:.2f})')


plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve for XGBoost")
plt.legend(loc="best")
plt.show()
```

B. Code For Brain Cancer Dataset

```
from google.colab import drive
drive.mount('/content/drive')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import sklearn
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
from sklearn.metrics import precision_recall_curve
import xgboost as xgb
from sklearn.tree import DecisionTreeRegressor, plot_tree
from xgboost import XGBRegressor
from sklearn.preprocessing import OneHotEncoder

df=pd.read_csv("/content/drive/MyDrive/Colab Notebooks/DT-BrainCancer.csv")
df.head()

df.info()

df.describe()
df.count()

df.isnull().sum()
df = df.dropna()
df.isnull().sum()

df.shape

df = df.drop('Unnamed: 0', axis=1)
df.shape

df.head()
un = df['sex'].unique()
print(un)
un = df['diagnosis'].unique()
print(un)
un = df['loc'].unique()
print(un)
categorical = df.select_dtypes(include=['object']).columns
numerical = df.select_dtypes(exclude=['object']).columns

print(categorical)
print(numerical)

encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_data = encoder.fit_transform(df[categorical])
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(categorical))
encoded_df.index = df.index

df = df.drop(categorical, axis=1)
df = pd.concat([df, encoded_df], axis=1)
X = df.drop(columns=['status'])
y = df['status']

print(X.shape)
print(y.shape)

X.head()
```

```
y.head()

corr = df.corr()

target = corr['status'].sort_values(ascending=False)

print("Correlation of each feature with the target variable (status):")
print(target)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state
    =42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
print(y_train.shape)
print(y_val.shape)
print(y_test.shape)

dt = DecisionTreeClassifier(random_state=42)

param_grid_dt = {
    'max_depth': [5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search_dt = GridSearchCV(estimator=dt, param_grid=param_grid_dt, cv=5, scoring='accuracy')
grid_search_dt.fit(X_train, y_train)

best_dt = grid_search_dt.best_estimator_
print("Best Decision Tree Parameters:", grid_search_dt.best_params_)

xgb_model = xgb.XGBClassifier(random_state=42)

param_grid_xgb = {
    'learning_rate': [0.001,0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 200, 300]
}

grid_search_xgb = GridSearchCV(estimator=xgb_model, param_grid=param_grid_xgb, cv=5, scoring='
    accuracy')
grid_search_xgb.fit(X_train, y_train)

best_xgb = grid_search_xgb.best_estimator_
print("Best XGBoost Parameters:", grid_search_xgb.best_params_)

def accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def confusion_matrix_manual(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    tn = np.sum((y_true == 0) & (y_pred == 0))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    return tp, tn, fp, fn

def precision_manual(y_true, y_pred):
```

```
    tp, tn, fp, fn = confusion_matrix_manual(y_true, y_pred)
    return tp / (tp + fp) if tp + fp > 0 else 0

def recall_manual(y_true, y_pred):
    tp, tn, fp, fn = confusion_matrix_manual(y_true, y_pred)
    return tp / (tp + fn) if tp + fn > 0 else 0

def f1_score_manual(y_true, y_pred):
    precision = precision_manual(y_true, y_pred)
    recall = recall_manual(y_true, y_pred)
    return 2 * (precision * recall) / (precision + recall) if precision + recall > 0 else 0

def precision_recall_curve_manual(y_true, y_prob):
    precision, recall, thresholds = precision_recall_curve(y_true, y_prob)
    return precision, recall, thresholds

def plot_precision_recall_curve(precision, recall, label):
    plt.plot(recall, precision, label=label)
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend()
    plt.show()

y_train_pred_dt = best_dt.predict(X_train)
y_val_pred_dt = best_dt.predict(X_val)
y_test_pred_dt = best_dt.predict(X_test)

y_train_pred_xgb = best_xgb.predict(X_train)
y_val_pred_xgb = best_xgb.predict(X_val)
y_test_pred_xgb = best_xgb.predict(X_test)

y_test_prob_dt = best_dt.predict_proba(X_test)[:, 1]
y_test_prob_xgb = best_xgb.predict_proba(X_test)[:, 1]

acc_train_dt = accuracy(y_train, y_train_pred_dt)
acc_val_dt = accuracy(y_val, y_val_pred_dt)
acc_test_dt = accuracy(y_test, y_test_pred_dt)

tp_dt, tn_dt, fp_dt, fn_dt = confusion_matrix_manual(y_test, y_test_pred_dt)
prec_dt = precision_manual(y_test, y_test_pred_dt)
rec_dt = recall_manual(y_test, y_test_pred_dt)
f1_dt = f1_score_manual(y_test, y_test_pred_dt)

print(f"Decision Tree Accuracy (Train): {acc_train_dt}")
print(f"Decision Tree Accuracy (Validation): {acc_val_dt}")
print(f"Decision Tree Accuracy (Test): {acc_test_dt}")
print(f"Confusion Matrix (Test): TP={tp_dt}, TN={tn_dt}, FP={fp_dt}, FN={fn_dt}")
print(f"Precision (Test): {prec_dt}")
print(f"Recall (Test): {rec_dt}")
print(f"F1 Score (Test): {f1_dt}")

prec_dt_curve, rec_dt_curve, _ = precision_recall_curve_manual(y_test, y_test_prob_dt)
plot_precision_recall_curve(prec_dt_curve, rec_dt_curve, label='Decision Tree')


acc_train_xgb = accuracy(y_train, y_train_pred_xgb)
acc_val_xgb = accuracy(y_val, y_val_pred_xgb)
acc_test_xgb = accuracy(y_test, y_test_pred_xgb)

tp_xgb, tn_xgb, fp_xgb, fn_xgb = confusion_matrix_manual(y_test, y_test_pred_xgb)
prec_xgb = precision_manual(y_test, y_test_pred_xgb)
rec_xgb = recall_manual(y_test, y_test_pred_xgb)
```

```
f1_xgb = f1_score_manual(y_test, y_test_pred_xgb)

print(f"XGBoost Accuracy (Train): {acc_train_xgb}")
print(f"XGBoost Accuracy (Validation): {acc_val_xgb}")
print(f"XGBoost Accuracy (Test): {acc_test_xgb}")
print(f"Confusion Matrix (Test): TP={tp_xgb}, TN={tn_xgb}, FP={fp_xgb}, FN={fn_xgb}")
print(f"Precision (Test): {prec_xgb}")
print(f"Recall (Test): {rec_xgb}")
print(f"F1 Score (Test): {f1_xgb}")

prec_xgb_curve, rec_xgb_curve, _ = precision_recall_curve_manual(y_test, y_test_prob_xgb)
plot_precision_recall_curve(prec_xgb_curve, rec_xgb_curve, label='XGBoost')
```

C. Code For Wage Dataset

```
from google.colab import drive
drive.mount('/content/drive')
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.tree import DecisionTreeRegressor, plot_tree
from xgboost import XGBRegressor
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/DT-Wage.csv')
df.head()
df.shape
df.info()
df.describe()
df.isnull().sum()
un = df['maritl'].unique()
print(un)
un = df['race'].unique()
print(un)
un = df['education'].unique()
print(un)
un = df['region'].unique()
print(un)
un = df['jobclass'].unique()
print(un)
un = df['health'].unique()
print(un)
categorical = df.select_dtypes(include=['object']).columns
numerical = df.select_dtypes(exclude=['object']).columns
encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_data = encoder.fit_transform(df[categorical])
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(categorical))
encoded_df.index = df.index
df = df.drop(categorical, axis=1)
df = pd.concat([df, encoded_df], axis=1)
X = df.drop(columns=['wage'])
y = df['wage']
X.head()
X.shape
corr = df.corr()

target = corr['wage'].sort_values(ascending=False)

print("Correlation of each feature with the target variable (wage):")
print(target)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state
    =42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
X_train.shape
X_val.shape
X_test.shape
X_train
param_grid = {
    'max_depth': [3, 5, 7, 10,12, None],
    'min_samples_split': [2, 5, 10,12],
```

```
    'min_samples_leaf': [1, 2, 5,7],
    'max_features': ['sqrt', 'log2', None],
    'criterion': ['squared_error', 'friedman_mse', 'absolute_error'],
}
model = DecisionTreeRegressor(random_state=42)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2,
    scoring='neg_mean_squared_error')
best_params = grid_search.best_params_
print("Best Parameters:", best_params)
best_model = grid_search.best_estimator_

y_train_pred = best_model.predict(X_train)
y_val_pred = best_model.predict(X_val)
y_test_pred = best_model.predict(X_test)

mse_train = mean_squared_error(y_train, y_train_pred)
mse_val = mean_squared_error(y_val, y_val_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

r2_train = r2_score(y_train, y_train_pred)
r2_val = r2_score(y_val, y_val_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Mean Squared Error (Train):", mse_train)
print("Mean Squared Error (Validation):", mse_val)
print("Mean Squared Error (Test):", mse_test)
print("R   (Train): {:.2f}".format(r2_train))
print("R   (Validation): {:.2f}".format(r2_val))
print("R   (Test): {:.2f}".format(r2_test))

from xgboost import XGBRegressor

X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state
    =42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

xgb_model = XGBRegressor(objective='reg:squarederror', random_state=42)

grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=5, n_jobs=-1, verbose
    =2, scoring='neg_mean_squared_error')

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print("Best Parameters:", best_params)

best_model = grid_search.best_estimator_

y_train_pred = best_model.predict(X_train)
y_val_pred = best_model.predict(X_val)
y_test_pred = best_model.predict(X_test)

mse_train = mean_squared_error(y_train, y_train_pred)
mse_val = mean_squared_error(y_val, y_val_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

r2_train = r2_score(y_train, y_train_pred)
r2_val = r2_score(y_val, y_val_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Mean Squared Error (Train):", mse_train)
print("Mean Squared Error (Validation):", mse_val)
print("Mean Squared Error (Test):", mse_test)
```

```
print(" R   (Train): {:.2f}".format(r2_train))
print(" R   (Validation): {:.2f}".format(r2_val))
print(" R   (Test): {:.2f}".format(r2_test))

plt.figure(figsize=(8, 6))
plt.scatter(y_train, y_train_pred, color="blue", label="Train")
plt.scatter(y_val, y_val_pred, color="orange", label="Validation")
plt.scatter(y_test, y_test_pred, color="green", label="Test")
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'k--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("True vs. Predicted Values")
plt.legend()
plt.show()

train_residuals = y_train - y_train_pred
val_residuals = y_val - y_val_pred
test_residuals = y_test - y_test_pred

plt.figure(figsize=(8, 6))
plt.scatter(y_train_pred, train_residuals, color="blue", label="Train")
plt.scatter(y_val_pred, val_residuals, color="orange", label="Validation")
plt.scatter(y_test_pred, test_residuals, color="green", label="Test")
plt.axhline(0, color="black", linestyle="--", linewidth=2)
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot")
plt.legend()
plt.show()
```

D. Code For Credit Dataset

```
from google.colab import drive
drive.mount('/content/drive')

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.tree import DecisionTreeRegressor, plot_tree
from xgboost import XGBRegressor
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/DT-Credit.csv')

df.head()
df.shape
df.info()
df.describe()
df.isnull().sum()
un = df['Own'].unique()
print(un)
un = df['Student'].unique()
print(un)

un = df['Region'].unique()
print(un)

un = df['Married'].unique()
print(un)

categorical = df.select_dtypes(include=['object']).columns
numerical = df.select_dtypes(exclude=['object']).columns

encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_data = encoder.fit_transform(df[categorical])
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(categorical))
encoded_df.index = df.index

df = df.drop(categorical, axis=1)
df = pd.concat([df, encoded_df], axis=1)

X = df.drop(columns=['Balance'])
y = df['Balance']

X.head()
X.shape

corr = df.corr()

target = corr['Balance'].sort_values(ascending=False)

print("Correlation of each feature with the target variable (Balance):")
print(target)
fig, ax = plt.subplots(1,1, figsize=(12, 12))
df.boxplot('Income', 'Balance', ax=ax)
plt.suptitle('Income vs Balance')
plt.title('')
plt.ylabel('Income')
```

```
plt.xticks(rotation=90)
plt.show()

fig, ax = plt.subplots(1,1, figsize=(12, 12))
df.boxplot('Age', 'Balance', ax=ax)
plt.suptitle('Age vs Balance')
plt.title('')
plt.ylabel('Age')
plt.xticks(rotation=90)
plt.show()
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state
    =42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

X_train.shape
X_val.shape
X_test.shape

param_grid = {
    'max_depth': [3, 5, 7, 10,12, None],
    'min_samples_split': [2, 5, 10,12],
    'min_samples_leaf': [1, 2, 5,7],
    'max_features': ['sqrt', 'log2', None],
    'criterion': ['squared_error', 'friedman_mse', 'absolute_error'],
}

model = DecisionTreeRegressor(random_state=42)

grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2,
    scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

best_model = grid_search.best_estimator_

y_train_pred = best_model.predict(X_train)
y_val_pred = best_model.predict(X_val)
y_test_pred = best_model.predict(X_test)

mse_train = mean_squared_error(y_train, y_train_pred)
mse_val = mean_squared_error(y_val, y_val_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

r2_train = r2_score(y_train, y_train_pred)
r2_val = r2_score(y_val, y_val_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Mean Squared Error (Train):", mse_train)
print("Mean Squared Error (Validation):", mse_val)
print("Mean Squared Error (Test):", mse_test)
print("R   (Train): {:.2f}".format(r2_train))
print("R   (Validation): {:.2f}".format(r2_val))
print("R   (Test): {:.2f}".format(r2_test))

y_train_pred = best_model.predict(X_train)
y_val_pred = best_model.predict(X_val)
y_test_pred = best_model.predict(X_test)

mse_train = mean_squared_error(y_train, y_train_pred)
```

```
mse_val = mean_squared_error(y_val, y_val_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

r2_train = r2_score(y_train, y_train_pred)
r2_val = r2_score(y_val, y_val_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Mean Squared Error (Train):", mse_train)
print("Mean Squared Error (Validation):", mse_val)
print("Mean Squared Error (Test):", mse_test)
print(" R   (Train): {:.2f}".format(r2_train))
print(" R   (Validation): {:.2f}".format(r2_val))
print(" R   (Test): {:.2f}".format(r2_test))

from xgboost import XGBRegressor

X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state
    =42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

 param_grid = {
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'n_estimators': [100, 200, 300],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
}

xgb_model = XGBRegressor(objective='reg:squarederror', random_state=42)

grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=5, n_jobs=-1, verbose
    =2, scoring='neg_mean_squared_error')

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print("Best Parameters:", best_params)

best_model = grid_search.best_estimator_

y_train_pred = best_model.predict(X_train)
y_val_pred = best_model.predict(X_val)
y_test_pred = best_model.predict(X_test)

mse_train = mean_squared_error(y_train, y_train_pred)
mse_val = mean_squared_error(y_val, y_val_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

r2_train = r2_score(y_train, y_train_pred)
r2_val = r2_score(y_val, y_val_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Mean Squared Error (Train):", mse_train)
print("Mean Squared Error (Validation):", mse_val)
print("Mean Squared Error (Test):", mse_test)
print(" R   (Train): {:.2f}".format(r2_train))
print(" R   (Validation): {:.2f}".format(r2_val))
print(" R   (Test): {:.2f}".format(r2_test))

plt.figure(figsize=(8, 6))
plt.scatter(y_train, y_train_pred, color="blue", label="Train")
plt.scatter(y_val, y_val_pred, color="orange", label="Validation")
```

```
plt.scatter(y_test, y_test_pred, color="green", label="Test")
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'k--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("True vs. Predicted Values")
plt.legend()
plt.show()

train_residuals = y_train - y_train_pred
val_residuals = y_val - y_val_pred
test_residuals = y_test - y_test_pred

plt.figure(figsize=(8, 6))
plt.scatter(y_train_pred, train_residuals, color="blue", label="Train")
plt.scatter(y_val_pred, val_residuals, color="orange", label="Validation")
plt.scatter(y_test_pred, test_residuals, color="green", label="Test")
plt.axhline(0, color="black", linestyle="--", linewidth=2)
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot")
plt.legend()
plt.show()
```