DATA 5322 Statistical Machine Learning II

Spring Quarter 2024

Vatsal Dalal

April 14, 2024

Exploring Youth Drug Use Patterns: A Decision Tree Analysis

Abstract:

In this study, I use decision tree analysis to examine the subject of teenage drug use using data from the National Survey on Drug Use and Health. With a focus on regression tasks and binary and multi-class classification, my goal is to understand the variables influencing teenage substance use. Through my investigation, I can determine if people have ever used marijuana, classify how often they have used it in the previous year, and determine when people first started smoking cigarettes.

For predicting teenage marijuana usage history (binary classification), the random forest model performed best with 86.3% accuracy. Important features were frequency of marijuana use and standard marijuana frequency.

For predicting number of days marijuana was used in the past year (multiclass classification), gradient boosting and pruned decision trees achieved 88.5% accuracy. Standard marijuana frequency and frequency of use were critical features.

For predicting age of starting smoking cigarettes (regression), the random forest model had the lowest test MSE of 5.45.

I can identify the pathways influencing teenagers related to drug behaviors by examining the branches of decision trees, which provides important information for treatments and legislation aimed at addressing teenage substance abuse. Come along with me as I take one branch at a time through the complex subject of teenage drug use.

Introduction:

In this investigation of youth substance use, I am utilizing tree-based algorithms to gain insight into different aspects of young people's interactions with drugs. The National Survey on Drug Use and Health (NSDUH) dataset serves as my tour guide, providing me with a detailed glimpse into teenage drug use.

The data I am working with contains a diverse array of variables, including substance use data, demographic information, and details about youths' experiences. The substance uses variables cover aspects such as alcohol, marijuana, and cigarette use, including indicators of whether individuals have used these substances, the frequency of use, and the age at which they initiated use. The demographic variables encompass factors like sex, race, health status, education level, family situation, income, and participation in government programs. Additionally, the dataset includes variables related to youths experiences, such as their feelings about school, participation in activities, and religious involvement.

Using a straightforward yes/no classification, I first examine whether teenagers have ever tried marijuana (mrjflag). Next, I examine how often people have used marijuana over the course of the previous year (mrjydays) and classify their usage into various ranges, from rare to frequent. Finally, I am investigating the initiation age of teen smoking and estimating the age of first use of cigarettes (IRCIGAGE).

By employing tree-based algorithms to address these questions, my goal is to identify the factors that motivate young people to engage in drug use. Also our goal is to provide information that will help direct efforts to help avoid substance abuse to make better decisions.

Theoretical Background:

Our analysis in this report is based on decision trees, which are the core tool in machine learning. Recursively dividing the data into subsets, these trees create hierarchical structures for classification and regression tasks by maximizing information gain or minimizing impurity at each node. Decision trees build reliable and useful predictive models by repeatedly segmenting the feature space according to attribute values.

We go beyond simple decision trees to include ensemble techniques like bagging, gradient boosting, and random forests. These methods improve model robustness and generalizability by combining predictions from multiple trees and leveraging the wisdom of crowds. In my ensemble model, I used bagging,

random forests, and gradient boosting. In bagging, key parameters include the number of estimators, sample sizes, and the subset of features to consider at each split. Random forests, an extension of bagging, introduce additional randomness by selecting a random subset of features at each split, reducing correlation among trees, and further enhancing robustness. Important parameters for random forests include the number of trees, the number of features at each split (max_features), tree depth, and the minimum number of samples required to split or be in a leaf node. Gradient boosting creates trees sequentially, with each tree learning from the errors of the previous ones. It uses a learning rate (shrinkage) to control the step size in correcting errors. Gradient boosting tends to achieve high accuracy but requires careful tuning to avoid overfitting. Key parameters for gradient boosting include the number of estimators, the learning rate, maximum tree depth, and regularization techniques such as subsampling.

Additionally, pruning methods help reduce overfitting and ensure that our models perform well when applied to new data. In regression, I've used a systematic approach to fine-tune parameters to minimize error. To automate the process of hyperparameter optimization and improve model performance, I employed the Optuna library, which streamlines the parameter tuning process and helps achieve better results.

Methodology:

- **Data Processing:**

First, we load the dataset into pandas After loading, we selected relevant columns for analysis, focusing on three key groups: substance-related columns, demographic information, and additional attributes related to youth behavior and experiences. The substance-related columns encompassed data about alcohol, marijuana, and tobacco use, while the demographic columns contained information on gender, race, education, parental roles, and more. After that, they are combined into a single data frame for easier analysis. To ensure the data was clean, we dealt with missing values and invalid entries by removing rows that contained certain codes indicating incomplete or incorrect information. we removed rows where imother or ifather had values 3 or 4, which indicate unclear information regarding the respondent's household composition—3 representing "Don't know if father is in the household" and 4 indicating that the respondent is older than 18 years, similarly for columns population density(PDEN10). Similarly, we addressed anomalies in the current grade level(EDUSCHGRD2) column, where values of 98 and 99 signified "legitimate skip" or "blank". Removing these rows was essential to maintain the accuracy of the data, ensuring it was ready for further analysis. After cleaning the data, we converted

categorical variables to the correct data type, distinguishing between unordered and ordered categorical variables. This step involved transforming unordered variables, like gender and race, into the 'category' data type, while ordered variables, such as education level and health status, were converted to ordered categorical data. After preprocessing, our dataset has 79 columns and 4032 rows.

- **Binary Classification:**

  Using the teen uses the marijuana or not (mrjflag)variable as our target, we concentrated on predicting whether teenagers have ever used marijuana for our binary classification task. Using a random state of 5 for reproducibility, the dataset was divided into training and testing sets, with 70% designated for training and 30% for testing. We used different models, such as bagging, gradient boosting, random forests, and decision trees, and we optimized each one using techniques for hyperparameter tuning. The model that performed the best at predicting marijuana use among teenagers was determined to be the one with the lowest test error on the testing set.

- **Multiclass Classification**

We first used a 70-30 split ratio to separate the data into training and testing sets in order to begin our multiclass classification approach to predicting the frequency of teenage marijuana use (mrjydays). We then trained several models and fine-tuned them for best results, including decision trees, random forests, and gradient boosting. Every model's accuracy was compared, and the top-performing model was chosen. The characteristics of this model were then examined in order to determine the key variables influencing the frequency of marijuana use among teenagers. Furthermore, to assess model performance, error comparison methods like Test Error and cross-validation were used. Finally, in order to identify the most pertinent predictors for the given task, variable selection techniques like feature_importances were applied.

- **Regression**

'IRCIGAGE', our target variable for regression analysis, was the age at which a person first smoked, with values ranging from 1 to 55 (never used, or 991). Using models such as DecisionTreeRegressor, RandomForestRegressor, our goal was to predict this variable. The Optuna library was used to optimize these models in order to determine the optimal parameters, and metrics like mean squared error (MSE) were used to assess their performance.

Computational Results:

- **Binary Classification:**

Initially, we trained a decision tree model, which classifies cases by learning decision rules from the data. Decision trees, however, frequently overfit the training set, which results in subpar generalization on new data. We used pruning strategies to reduce the decision tree's overfitting and keep it from being too complex. Pruning lowers variance while preserving bias, which enhances the model's capacity to generalize to new data.

In bagging, multiple decision trees are trained using bootstrap samples of the training data, and their predictions are aggregated to produce the final predictions. By using this method, the variance is decreased and the model's overall performance is enhanced.

By adding unpredictability to the feature selection process, random forests outperform bagging. Random forests reduce correlation between the trees and improve prediction performance by randomly selecting a subset of features for splitting at each node rather than taking into account all features.

By adding weak learners—usually decision trees—to the model one after the other, each tree improves the mistakes of the one before it, gradient boosting is an ensemble technique. To maximize the performance of the model, gradient boosting minimizes a loss function iteratively.

After training these models, we evaluated their performance based on two key metrics: Test Error and Accuracy. Here is the comparison table:

Model	Test Error	Accuracy
Decision Tree	0.137	86.28%
Pruned Decision Tree	0.100	89.92%

Model	Test Error	Accuracy
Bagging	0.096	90.41%
Random Forest	0.091	90.91%
Gradient Boosting	0.105	89.53%

Table 1: Binary Model Comparison

From table 1, we can observe that the random forest model achieved the lowest MSE and highest accuracy among all models, making it the best-performing model for predicting teenage marijuana use in this binary classification task.

- **Multi-class Classification:**

In our multiclass classification task, we aimed to predict the frequency of teenage marijuana use in the past month using the 'DAYS USED MARIJUANA IN PAST MONTH' variable. This variable categorizes marijuana use frequency into five distinct levels:

1-2 Days (IRMJFM=1-2)

3-5 Days (IRMJFM=3-5)

6-19 Days (IRMJFM=6-19)

20-30 Days (IRMJFM=20-30)

Non-User or No Past Month Use (IRMJFM=91,93)

I trained several models including Decision Trees, Bagging, Random Forests, and Gradient Boosting, focusing on test error and accuracy. An unpruned Decision Tree achieved an accuracy of approximately 83.28% on the test set. When pruned using GridSearchCV for hyperparameter tuning, the model's accuracy significantly improved, reducing the test error to around 11.46%. Bagging exhibited a higher test error (53.66%), suggesting that feature variability might impact accuracy. Random Forests, with an mtry value of 3, showed a test error of 71.63%. Gradient Boosting had a test error comparable to the pruned Decision Tree, indicating similar performance.

As we can see from the figure 1 below comparison plot between models pruned and gradient boosting the best model for multiclassification for our target variable.

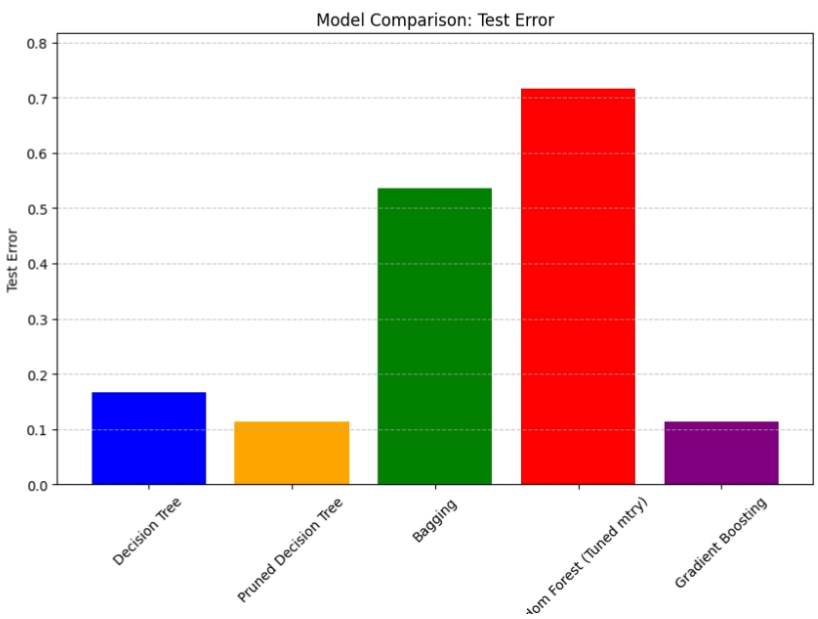

Figure 1: Multiclassification model comparison

- **Regressor:**

In our regression analysis, we sought to predict the age at which individuals first used cigarettes, leveraging the 'CIGARETTE AGE OF FIRST USE - IMPUTATION REVISED' variable, which encompasses age ranges from 1 to 55, with 991 representing those who never used cigarettes.

In the Decision Tree model, after employing Optuna for hyperparameter tuning, we identified an optimal tree depth of 4 with 8 leaf nodes. This configuration was determined to minimize the test mean squared error (MSE), resulting in a final MSE of 5.882, the Random Forest regressor with optimal parameters ({'n_estimators': 163, 'max_depth': 11, 'min_samples_split': 7, 'min_samples_leaf': 2}) achieved a test MSE of 5.473. Gradient Boosting model attained test MSEs of 5.748.

Based on figure 2 below test mean squared error (MSE), the Random Forest indeed achieved the lowest value of 5.473 among the regression models evaluated. This lower test error indicates that the Random Forest model performs better in predicting the age at which individuals first used cigarettes compared to the other models considered.

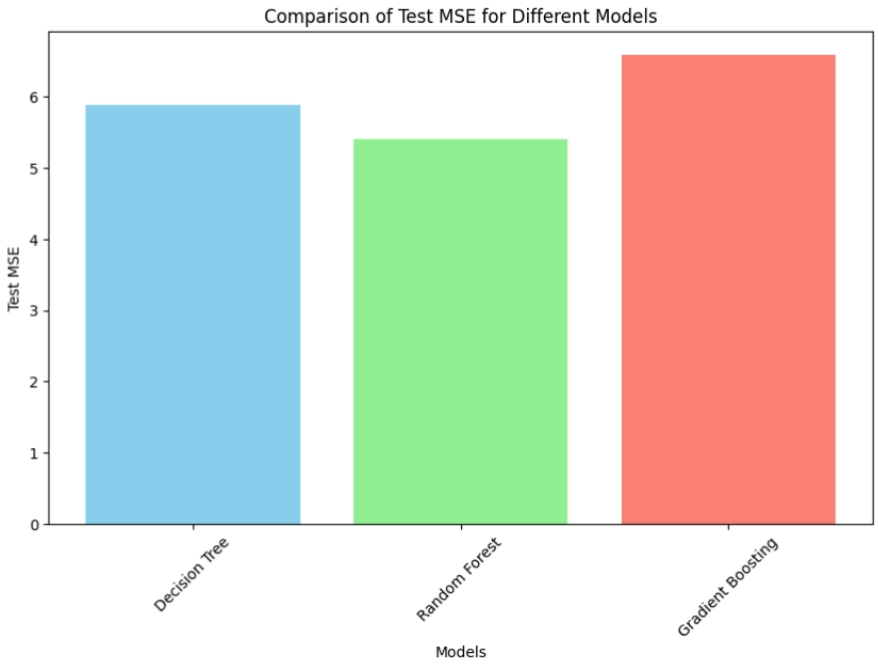

Figure 2: Regression model comparison

Discussion:

In this discussion, I reflect on the computational results from the binary classification, multi-class classification, and regression analyses. I examine why certain results are likely correct, identify key variables that significantly impact predictions, and emphasize the importance of choosing appropriate data types for different analyses.

- **Binary Classification:**

In our binary classification task, we predicted whether teenagers use marijuana. The target variable was categorical, representing a binary outcome (use or no use). The Random Forest model performed better than other models because it uses a group of decision trees instead of just one. This approach, called an ensemble, helps reduce errors by averaging the predictions from many trees, which makes the model more stable and reliable. The reason it works so well is that it reduces the chance of overfitting, a common problem in individual decision trees, where the model learns patterns that don't actually hold up with new data.

Figure 3 illustrates the most important features in the Random Forest model, providing insights into which variables have the most significant impact on prediction. The importance of social factors, such as "PEERS USING MARIJUANA MONTHLY" and "CLSE FRNDS FEEL ABT YTH USE MARIJUANA MON," highlights the influence of peer groups and close friends on teenage behavior. These predictors suggest that social dynamics play a significant role in whether teenagers choose to use marijuana, emphasizing the need for interventions targeting peer pressure and promoting positive behaviors among adolescents.

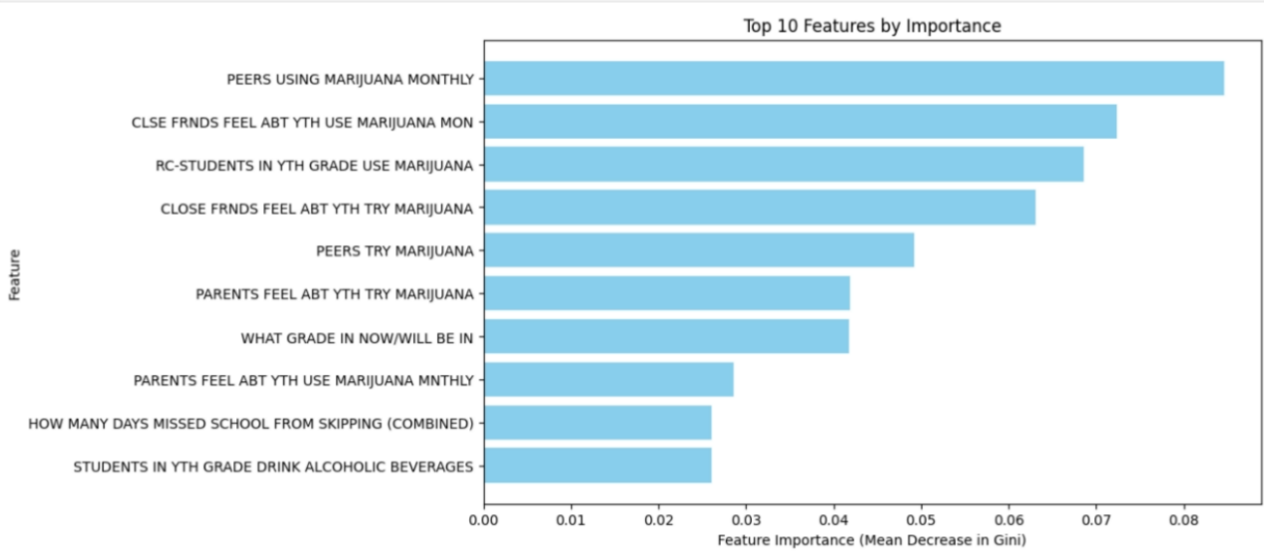

Figure 3: Important features of random forest model

In below figure 4, there is pruned binary tree to classify the teen use marijuana, which I have simplified than in my python code. In this I have removed other nodes

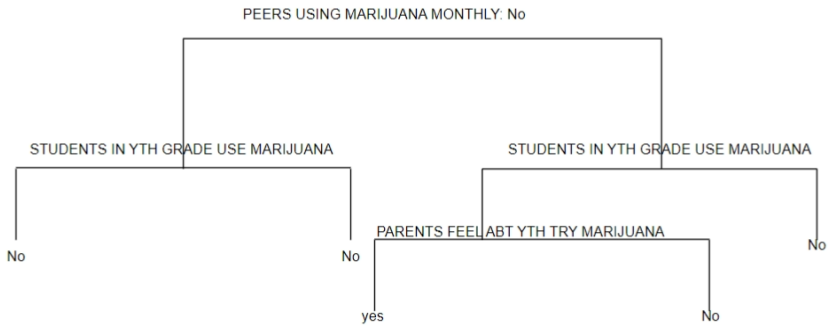

Figure 4: Simplify Pruned Binary Tree

The data is first divided into groups according to the feature "Peers using marijuana monthly" in the tree. The data is sent to the appropriate branch if peers of the students use marijuana on a monthly basis. If not, it proceeds to the left branch.

From the left branch, one arrives at a leaf node with the label "No," where peers refrain from monthly marijuana use. Put differently, the decision tree suggests that these students will not be monthly marijuana users.

The feature "Students in YTH grade use marijuana" causes the right branch, where peers use marijuana on a monthly basis, to split once more. The data is directed to the appropriate branch if students use marijuana themselves while in YTH grade. If not, it proceeds to the left branch.

A leaf node marked "No" is reached by the left branch, where YTH grade students who do not use marijuana are located. In other words, the decision tree indicates that these students won't use marijuana on a monthly basis.

Another leaf node with the label "PARENTS FEEL ABT YTH TRY MARIJUANA" is reached by the right branch, where marijuana is used by students in the YTH grade.

These results shows the importance of social influences and parental attitudes in predicting teenage marijuana use. By exploring these key features in

a simplified decision tree, we can better understand the factors contributing to teenage substance use and identify potential intervention points to reduce the risk of drug use among adolescents. This discussion highlights the critical role of simplifying complex models through pruning and the added stability provided by ensemble techniques like Random Forest, offering a more reliable and interpretable approach to predicting teenage behavior.

- **Multiclass Classification:**

For the multi-class classification task I predicted how often teenagers use marijuana. The target variable wasn't just a simple yes or no; it had several levels indicating different frequencies of use. the highest accuracy achieved by the Pruned Decision Tree and Gradient Boosting models suggests that these models can effectively capture distinct patterns in the data. The Pruned Decision Tree, with its simpler structure, indicating that avoiding overfitting is crucial for achieving reliable results. This aligns with the idea that simpler models often generalize better, especially when there is a risk of overfitting.

Important predictors for pruned decision tree, such as "Peer Monthly Marijuana Use" (yflmjmo) and " STUDENTS IN YTH GRADE USE MARIJUANA" (stndsmj), make logical sense in the context of teenage marijuana use. These variables represent social influences and peer behavior, which are known to play a significant role in adolescent substance use. Ethically, recognizing these factors can guide targeted interventions to address peer pressure and educate teenagers about the risks of marijuana use.

- **Regression:**

For the regression analysis, where the objective was to predict the age at which individuals first used cigarettes which is a continues variable, Random Forest and Gradient Boosting models demonstrated solid performance. The Random Forest model, with optimal hyperparameters, achieved a test mean squared error (MSE) of 5.473, while Gradient Boosting, with an MSE of 5.748, further supported the effectiveness of ensemble methods in regression tasks.

Influential features in the regression analysis included " WHAT GRADE IN NOW/WILL BE IN" (EDUSCHGRD2), "YTH PARTICIPATED IN YOUTH ACTIVITIES" (YTHACT2), and " RACE/HISPANICITY RECODE" (NEWRACE2). These variables indicate broader socio-economic and behavioral patterns that may affect when individuals first start using cigarettes. Education level could relate to access to information about the risks of smoking, while youth activities might reflect the social contexts in which cigarette use occurs. Recognizing

these factors can help public health officials design strategies to prevent early cigarette use and encourage healthier lifestyles among young people.

- **Data Types and Model Predictions:**

In my analysis, the choice of data type—binary, ordinal (categorical and ordered), or numerical—played a important role in shaping our predictions. Binary variables have two outcomes, such as "yes" or "no." I used them in the binary classification task to predict whether teenagers use marijuana. These variables are suitable for decision trees and ensemble models like Random Forest, allowing for clear and straightforward predictions. Ordinal variables have a defined order or hierarchy, as seen in " WHAT GRADE IN NOW/WILL BE IN" (EDUSCHGRD2). Which are useful in multi-class classification, where the target variable has a clear sequence of categories. This type of data allows models to capture relationships with an inherent order.

Conclusion:

In my analysis, I found that social factors, peer pressure, and education levels have a significant impact on teenage marijuana use and the age at which individuals start smoking cigarettes. The Random Forest model emerged as the best performer in binary classification, pointing to the crucial role of peer influence and educational settings in shaping substance use behaviors. Similarly, in multi-class classification, social and educational variables remained key predictors, reinforcing the idea that tailored interventions are crucial. In the regression analysis, I saw that education, youth activities, and race were instrumental in predicting the age at which individuals first used cigarettes. These results suggest that targeted interventions and educational programs are necessary to reduce substance use among adolescents. By focusing on key social determinants, we can foster healthier behaviors and reduce the risks associated with teenage substance use.

References:

1. https://medium.com/data-and-beyond/master-the-power-of-optuna-a-step-by-step-guide-ed43500e9b95
2. https://github.com/mendible/5322/blob/main/Homework%201/YouthParse.R
3. https://www.datafiles.samhsa.gov/sites/default/files/field-uploads-protected/studies/NSDUH-2020/NSDUH-2020-datasets/NSDUH-2020-DS0001/NSDUH-2020-DS0001-info/NSDUH-2020-DS0001-info-codebook.pdf
4. https://www.mathworks.com/campaigns/offers/next/choosing-the-best-machine-learning-classification-model-and-avoiding-overfitting.html

Appendix:

Data Processing:

```
import pandas as pd


# Load data

youth_data                                                                    =
pd.read_csv('https://media.githubusercontent.com/media/mendible/5322/m
ain/Homework%201/youth_data.csv')


# Substance columns

substance_cols = [

    'iralcfy', 'irmjfy', 'ircigfm', 'IRSMKLSS30N', 'iralcfm', 'irmjfm',

    'ircigage', 'irsmklsstry', 'iralcage', 'irmjage',

    'mrjflag', 'alcflag', 'tobflag',

    'alcydays',   'mrjydays',   'alcmdays',   'mrjmdays',   'cigmdays',
'smklsmdays'

]


# Demographic columns

demographic_cols = [
```

```
    'irsex',   'NEWRACE2',   'HEALTH2',   'eduschlgo',   'EDUSCHGRD2',
'eduskpcom',
    'imother', 'ifather', 'income', 'govtprog', 'POVERTY3', 'PDEN10',
'COUTYP4'
]


# Select columns
df_youth = youth_data.loc[:, 'schfelt':'rlgfrnd']
df_substance = youth_data[substance_cols]
df_demog = youth_data[demographic_cols]


# Combine dataframes
df = pd.concat([df_substance, df_youth, df_demog], axis=1)


# Convert to categorical variables
unordered_factor_cols = list(df_youth.columns) + ['mrjflag', 'alcflag',
'tobflag',  'irsex',  'NEWRACE2',  'eduschlgo',  'imother',  'ifather',
'govtprog', 'PDEN10', 'COUTYP4']
ordered_factor_cols = ['EDUSCHGRD2', 'HEALTH2', 'POVERTY3', 'income']


df[unordered_factor_cols]                                          =
df[unordered_factor_cols].astype('category')
df[ordered_factor_cols]   =   df[ordered_factor_cols].apply(lambda   x:
pd.Categorical(x, ordered=True))


# Cleaning the demographic variables
df = df[~df['EDUSCHGRD2'].isin([98, 99])]
df = df[~df['eduskpcom'].isin([94, 97, 98, 99])]
df = df[~df['imother'].isin([3, 4])]
df = df[~df['ifather'].isin([3, 4])]
```

```
df = df[df['PDEN10'] != 3]


# Reset index
df.reset_index(drop=True, inplace=True)


# Check dimensions
print(df.shape)
```

Method 1 : Binary Classification:

Decision Tree:

```
from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier,plot_tree

from sklearn.metrics import confusion_matrix, accuracy_score

import matplotlib.pyplot as plt


# Select relevant columns

df_demog = df[demographic_cols]

df_youth = df.loc[:, 'schfelt':'rlgfrnd']  # Assuming SCHFELT to RLGFRND are
the column names

MRJFLAG = df['mrjflag']


# Concatenate the selected data frames horizontally (by columns)

data = pd.concat([df_demog, df_youth, MRJFLAG], axis=1)


# Print the dimensions of the resulting dataset
```

```
print(data.shape)


# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.3, random_state=5)


# Build a decision tree model on the training data
X_train = train_data.drop(columns=['mrjflag'])
y_train = train_data['mrjflag']
decision_model = DecisionTreeClassifier(random_state=1)
decision_model
decision_model.fit(X_train, y_train)


# Predict on test data
X_test = test_data.drop(columns=['mrjflag'])
y_test = test_data['mrjflag']
predictions = decision_model.predict(X_test)


# Calculate accuracy
accuracy = accuracy_score(test_data['mrjflag'], predictions)
print("Accuracy:", accuracy)


# Generate Confusion Matrix
conf_matrix = confusion_matrix(test_data['mrjflag'], predictions)
print("Confusion Matrix:")
```

```
print(conf_matrix)
```

```
# Calculate Mean Squared Error (MSE)

decison_mse = ((predictions != test_data['mrjflag']) ** 2).mean()

print("Mean Squared Error (MSE):", decison_mse)
```

```
importances = decision_model.feature_importances_
```

```
# Create a DataFrame to display feature importances

feature_importance_df = pd.DataFrame({'Feature': X_train.columns,
'Importance': importances})

feature_importance_df = feature_importance_df.sort_values(by='Importance',
ascending=False).reset_index(drop=True)

print("Feature Importances:")

print(feature_importance_df.head(10))

# Plot decision tree

plt.figure(figsize=(50,50))

plot_tree(decision_model, feature_names=X_train.columns, class_names=['No',
'Yes'], filled=True,fontsize=12)

plt.show()
```

Pruning:

```
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.model_selection import GridSearchCV

from sklearn.metrics import accuracy_score,
confusion_matrix,mean_squared_error



# Define parameter grid

param_grid = {'max_leaf_nodes': range(2, 20)}



# Initialize DecisionTreeClassifier

decision_model = DecisionTreeClassifier(random_state=5)



# Initialize GridSearchCV

grid_search = GridSearchCV(estimator=decision_model, param_grid=param_grid)



# Fit the grid search to the data

grid_search.fit(X_train, y_train)



# Get the best estimator

best_decision_model = grid_search.best_estimator_



# Print the best parameters

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



# Predict on test data using best model

best_predictions = best_decision_model.predict(X_test)
```

```
# Calculate Mean Squared Error (MSE)
best_mse = ((best_predictions != test_data['mrjflag']) ** 2).mean()
print("MSE:", best_mse)

# Generate Confusion Matrix for best model
best_conf_matrix = confusion_matrix(test_data['mrjflag'], best_predictions)
print("Confusion Matrix:")
print(best_conf_matrix)

# Extract grid search results
results = grid_search.cv_results_
leaf_nodes = results['param_max_leaf_nodes']
neg_mse = -results['mean_test_score']

# Plot results
plt.figure(figsize=(8, 6))
plt.plot(leaf_nodes, neg_mse, marker='o', linestyle='-')
plt.xlabel('Max Leaf Nodes')
plt.ylabel('Mean Cross-Validated Negative MSE')
plt.title('Grid Search Results')
plt.grid(True)
plt.show()

feature_importances = best_decision_model.feature_importances_
```

```
# Create a DataFrame to store feature importances

importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance':
feature_importances})



# Sort the DataFrame by importance in descending order

importance_df = importance_df.sort_values(by='Importance', ascending=False,)

importance_df.reset_index(drop=True, inplace=True)

# Print the top 10 features

print("Top 5 Features:")

print(importance_df.head(5))

# Plot pruned decision tree

plt.figure(figsize=(20,10))

plot_tree(best_decision_model, feature_names=X_train.columns,
class_names=['No', 'Yes'], filled=True,fontsize=12)

plt.show()




from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import mean_squared_error, confusion_matrix

import matplotlib.pyplot as plt

import numpy as np



# Set the seed for reproducibility

np.random.seed(5)
```

```
# Perform bagging with random forest for different values of mtry

p = range(1, X_train.shape[1] + 1)

error = []



for i in p:

    # Fit the random forest model with current mtry value

    bagging_rf_model = RandomForestClassifier(n_estimators=500,
max_features=i, random_state=5)

    bagging_rf_model.fit(X_train, y_train)



    # Predict on test data using bagging model

    bagging_rf_predictions = bagging_rf_model.predict(X_test)



    # Calculate error rate

    error_rate = np.mean(bagging_rf_predictions != y_test)

    error.append(error_rate)



# Plot the error rate for different values of mtry

plt.plot(p, error, marker='o')

plt.xlabel('mtry')

plt.ylabel('Error Rate')

plt.title('Error Rate vs mtry')

index = np.argmin(np.round(error, 3)) # Index of the minimum error rate

Min_error = error[index]
```

```
min_p = p[index]

plt.axvline(x=min_p, color='red', linestyle='--')

plt.text(min_p, Min_error, "Min Error", color='red')

plt.show()
```

Random Forest:

```
best_mtry = min_p

print("Best value of mtry:", best_mtry, "has the error rate of:",
round(Min_error, 2))



# Fit the random forest model with the best mtry value

bagging_rf_model = RandomForestClassifier(n_estimators=500,
max_features=int(best_mtry), random_state=5)

bagging_rf_model.fit(X_train, y_train)



# Predict on test data using bagging model

bagging_rf_predictions = bagging_rf_model.predict(X_test)



# Calculate Mean Squared Error (MSE)

mse = mean_squared_error(y_test, bagging_rf_predictions)

print("Mean Squared Error (MSE):", mse)



# Generate Confusion Matrix

conf_matrix = confusion_matrix(y_test, bagging_rf_predictions)

print("Confusion Matrix:")

print(conf_matrix)
```

```
# Get feature importances

feature_importances = bagging_rf_model.feature_importances_


# Get the indices of the top 10 features

top_10_indices = np.argsort(feature_importances)[::-1][:10]


# Get the names and importances of the top 10 features

top_10_features = [X_train.columns[idx] for idx in top_10_indices]

top_10_importances = feature_importances[top_10_indices]

# Plot the top 10 features and their importances

plt.figure(figsize=(10, 6))

plt.barh(top_10_features, top_10_importances, color='skyblue')

plt.xlabel('Mean Decrease in Gini')

plt.ylabel('Feature')

plt.title('Top 10 Features by Importance')

plt.gca().invert_yaxis()  # Invert y-axis to display the most important
feature at the top

plt.show()
```

Bagging:

```
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score, mean_squared_error,
confusion_matrix

import pandas as pd
```

```
import matplotlib.pyplot as plt



# Set the seed for reproducibility

import numpy as np

np.random.seed(5)



# Define the number of variables to sample at each split

mtry_value = train_data.shape[1] - 1



# Perform bagging with random forest using mtry

bagging_model = RandomForestClassifier(n_estimators=500,
max_features=mtry_value, random_state=5)

bagging_model.fit(train_data.drop(columns=['mrjflag']),
train_data['mrjflag'])



# Predict on test data using bagging model

bagging_predictions =
bagging_model.predict(test_data.drop(columns=['mrjflag']))



# Calculate accuracy

bagging_accuracy = accuracy_score(test_data['mrjflag'], bagging_predictions)



# Calculate Mean Squared Error (MSE)

bagging_mse = mean_squared_error(test_data['mrjflag'], bagging_predictions)



# Print accuracy
```

```
print("Bagging Accuracy:", bagging_accuracy)



# Print MSE

print("Bagging MSE:", bagging_mse)



# Generate Confusion Matrix for bagging model

bagging_conf_matrix = confusion_matrix(test_data['mrjflag'],
bagging_predictions)

print("Confusion Matrix:")

print(bagging_conf_matrix)



# Extract feature importances from the bagging model

importances = bagging_model.feature_importances_

feature_names = train_data.drop(columns=['mrjflag']).columns



# Create a DataFrame to store feature importances

importance_df = pd.DataFrame({'Feature': feature_names, 'Importance':
importances})



# Sort the DataFrame by importance in descending order

importance_df = importance_df.sort_values(by='Importance',
ascending=False).reset_index(drop=True)



# Print top 10 feature importances

print("Top 10 Feature Importances:")

print(importance_df.head(10))
```

```
# Plot variable importance

plt.figure(figsize=(10, 6))

plt.barh(importance_df['Feature'][:20], importance_df['Importance'][:20],
color='purple')

plt.xlabel('Mean Decrease in Gini')

plt.ylabel('Variables')

plt.title('Variable Importance for Consumption of Marijuana')

plt.gca().invert_yaxis()

plt.show()
```

Gradient Boosting:

```
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.model_selection import GridSearchCV, cross_val_score

from sklearn.metrics import mean_squared_error, confusion_matrix

from sklearn.impute import SimpleImputer

import pandas as pd


# Remove missing values from train and test data

train_data_clean = train_data.dropna()

test_data_clean = test_data.dropna()


# Define a list of shrinkage values to try

shrinkage_values = [0.1, 0.21, 0.3,0.5]
```

```
# Initialize lists to store results

mse_scores = []

confusion_matrices = []



# Iterate over shrinkage values

for shrinkage in shrinkage_values:

    # Train Gradient Boosting model with current shrinkage value

    gb_model = GradientBoostingClassifier(learning_rate=shrinkage,
n_estimators=100, subsample=0.8, max_depth=3, random_state=123)



    # Perform cross-validation to evaluate the model

    cv_scores = cross_val_score(gb_model,
train_data_clean.drop(columns=['mrjflag']), train_data_clean['mrjflag'],
cv=5, scoring='neg_mean_squared_error')

    mse_scores.append((-1) * cv_scores.mean())  # Take the negative mean
squared error



    # Fit the model on the entire training data

    gb_model.fit(train_data_clean.drop(columns=['mrjflag']),
train_data_clean['mrjflag'])



    # Make predictions on the test data

    test_preds = gb_model.predict(test_data_clean.drop(columns=['mrjflag']))



    # Calculate confusion matrix
```

```
    confusion_matrices.append(confusion_matrix(test_data_clean['mrjflag'],
test_preds))



# Select the best shrinkage value based on MSE

best_shrinkage_idx = mse_scores.index(min(mse_scores))

best_shrinkage = shrinkage_values[best_shrinkage_idx]

print("Best shrinkage value:", best_shrinkage)



plt.figure(figsize=(8, 6))

plt.plot(shrinkage_values, mse_scores, marker='o', linestyle='-')

plt.title('Shrinkage vs. Mean Squared Error (MSE)')

plt.xlabel('Shrinkage')

plt.ylabel('Mean Squared Error (MSE)')

plt.xticks(shrinkage_values)

plt.grid(True)

plt.show()




# Train the Gradient Boosting model with the best shrinkage value

best_gb_model = GradientBoostingClassifier(learning_rate=best_shrinkage,
n_estimators=100, subsample=0.8, max_depth=3, random_state=123)

best_gb_model.fit(train_data_clean.drop(columns=['mrjflag']),
train_data_clean['mrjflag'])



# Make predictions on the test data

test_preds = best_gb_model.predict(test_data_clean.drop(columns=['mrjflag']))
```

```
# Calculate Mean Squared Error (MSE)

best_gb_mse = mean_squared_error(test_data_clean['mrjflag'], test_preds)

print("Best model MSE:", best_gb_mse)



# Generate Confusion Matrix for best model

best_conf_matrix = confusion_matrix(test_data_clean['mrjflag'], test_preds)

print("Confusion Matrix for best model:")

print(best_conf_matrix)



# Extract feature importance

feature_importance = pd.DataFrame({'Feature':
train_data_clean.drop(columns=['mrjflag']).columns,

                                   'Importance':
best_gb_model.feature_importances_})

top_10_features = feature_importance.sort_values(by='Importance',
ascending=False).reset_index(drop=True).head(10)

print("Top 10 Feature Importance:")

print(top_10_features)
```

Method 2: Multiclass Classification:

```
import pandas as pd



# Select relevant columns
```

```
df_demog = df[demographic_cols]

df_youth = df.loc[:, 'schfelt':'rlgfrnd']  # Assuming SCHFELT to RLGFRND are
the column names

mrjydays = df['mrjydays']



# Concatenate the selected data frames horizontally (by columns)

mrjydays_data = pd.concat([df_demog, df_youth, mrjydays], axis=1)

mrjydays_data= mrjydays_data.dropna()



# Print the dimensions of the resulting dataset

print(mrjydays_data.shape)

mrjydays_data



X = mrjydays_data.drop(columns=['mrjydays'])

y = mrjydays_data['mrjydays']



# Split data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=123)



# Train the DecisionTreeClassifier model

decision_model1 = DecisionTreeClassifier(random_state=5)

decision_model1.fit(X_train, y_train)



# Calculate feature importance
```

```
feature_importance = decision_model1.feature_importances_


# Predict on test data

predictions1 = decision_model1.predict(X_test)


accuracy = accuracy_score(y_test, predictions1)

print("Accuracy:", accuracy)


# Calculate test error

decision_test_error = 1 - accuracy

print("Test Error:", decision_test_error)


# Print feature importance

sorted_indices = np.argsort(feature_importance)[::-1]

top_10_features = X.columns[sorted_indices][:10]

print("Top 10 Features and their Importance Scores:")

for feature, importance in zip(top_10_features,
feature_importance[sorted_indices][:10]):

    print(feature, ":", importance)

# Plot decision tree

class_names = [str(label) for label in decision_model1.classes_]


# Plot the decision tree

plt.figure(figsize=(50, 50))
```

```
plot_tree(decision_model1, filled=True, feature_names=X.columns,
class_names=class_names,fontsize=12)

plt.show()
```

Pruning:

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import GridSearchCV

from sklearn.metrics import accuracy_score

import seaborn as sns



# Define the parameter grid for GridSearchCV

param_grid = {

    'max_leaf_nodes': range(2, 20)  # Define a range of values for
max_leaf_nodes

}



# Initialize the decision tree classifier

decision_model1 = DecisionTreeClassifier(random_state=5)



# Perform GridSearchCV for pruning

grid_search = GridSearchCV(estimator=decision_model1, param_grid=param_grid,
cv=5)

grid_search.fit(X_train, y_train)



# Extract the mean test scores from the GridSearchCV results

mean_test_scores = grid_search.cv_results_['mean_test_score']
```

```
# Reshape the mean test scores to match the shape of the parameter grid

mean_test_scores = mean_test_scores.reshape(-1,
len(param_grid['max_leaf_nodes']))



# Create a heatmap to visualize the mean test scores

plt.figure(figsize=(10, 6))

for fold, scores in enumerate(mean_test_scores):

    plt.plot(param_grid['max_leaf_nodes'], scores, marker='o', label=f'CV
Fold {fold + 1}')



plt.title("Grid Search Results")

plt.xlabel("max_leaf_nodes")

plt.ylabel("Mean Test Score")

plt.legend()

plt.grid(True)

plt.show()




# Get the best pruned model

best_pruned_model = grid_search.best_estimator_



# Predict on test data using the best pruned model

predictions1 = best_pruned_model.predict(X_test)
```

```
# Calculate accuracy

accuracy = accuracy_score(y_test, predictions1)

print("Accuracy:", accuracy)


# Calculate test error

pruned_test_error = 1 - accuracy

print("Test Error:", pruned_test_error)


# Get feature importance scores from the best pruned model

feature_importance = best_pruned_model.feature_importances_


# Sort feature importance scores

sorted_indices = np.argsort(feature_importance)[::-1]

top_10_features = X.columns[sorted_indices][:5]


# Print top 10 features and their importance scores

print("Top 10 Features:")

for feature, importance in zip(top_10_features,
feature_importance[sorted_indices][:10]):

    print(feature, ":", importance)

plt.figure(figsize=(20, 10))  # Set the figure size

plot_tree(best_pruned_model, filled=True, feature_names=X.columns,
class_names=class_names,fontsize=10)

plt.title("Pruned Decision Tree")

plt.show()
```

Bagging:

```
from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean_squared_error


# Set the seed for reproducibility

np.random.seed(5)


# Define the number of variables to sample at each split

mtry_value = len(train_data.columns) - 1


# Initialize the random forest regressor with 500 trees

bagging_model = RandomForestRegressor(n_estimators=500,
max_features=mtry_value, random_state=5)


# Fit the model to the training data

bagging_model.fit(X_train, y_train)


# Predict on test data using bagging model

bagging_predictions = bagging_model.predict(X_test)


y_test_numeric = y_test.astype(float)


# Calculate test error
```

```
bagging_test_error = np.mean(np.abs(bagging_predictions -
y_test_numeric))

print("Test Error:", bagging_test_error)



# Convert predicted values to integers for confusion matrix
calculation

bagging_predictions_int = np.round(bagging_predictions).astype(int)



# Generate Confusion Matrix

conf_matrix = confusion_matrix(y_test, bagging_predictions_int,
labels=np.unique(bagging_predictions_int))



plt.figure(figsize=(8, 6))

sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='g')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()
```

Random Forest:

```
from sklearn.ensemble import RandomForestRegressor

import numpy as np

import matplotlib.pyplot as plt



# Define a range of values for mtry

p = range(1, X_train.shape[1] + 1)
```

```
# Initialize an array to store test errors

errors = []



# Initialize an array to store feature importances

feature_importances = []



# Iterate over each value of mtry

for i in p:

    # Initialize random forest regressor with the current value of mtry

    rf_model = RandomForestRegressor(n_estimators=500, max_features=i,
random_state=5)



    # Train the model

    rf_model.fit(X_train, y_train)



    # Predict on test data

    rf_predictions = rf_model.predict(X_test)



    # Calculate test error (1 - R^2)

    test_error = 1 - rf_model.score(X_test, y_test)



    # Append the test error to the errors list

    errors.append(test_error)
```

```
    # Get feature importances
    feature_importances.append(rf_model.feature_importances_)

# Find the index of the minimum test error
min_error_index = np.argmin(errors)

# Plot the test errors for different values of mtry
plt.plot(p, errors, marker='o')
plt.xlabel('mtry')
plt.ylabel('Test Error')
plt.title('Test Error vs. mtry')
plt.axvline(x=min_error_index + 1, color='red', linestyle='--', label='Min
Test Error')
plt.legend()
plt.show()

# Print the best value of mtry and its corresponding test error
best_mtry = min_error_index + 1
print("Best value of mtry:", best_mtry)
print("Min Test Error:", errors[min_error_index])

# Calculate average feature importances
avg_feature_importances = np.mean(feature_importances, axis=0)

# Get the indices of the top 10 most important features
```

```
top_10_indices = np.argsort(avg_feature_importances)[::-1][:10]


# Get the names of the top 10 most important features

top_10_features = X.columns[top_10_indices]


# Get the importance scores of the top 10 features

top_10_importances = avg_feature_importances[top_10_indices]


# Print the top 10 most important features and their importance scores

print("\nTop 10 Most Important Features:")

for feature, importance in zip(top_10_features, top_10_importances):

    print(feature, ":", importance)


# Plot feature importances

plt.figure(figsize=(10, 6))

plt.bar(range(len(top_10_features)), top_10_importances,
tick_label=top_10_features)

plt.title('Top 10 Most Important Features')

plt.xlabel('Feature')

plt.ylabel('Importance')

plt.xticks(rotation=45, ha='right')

plt.show()

thresholds = [0, 1, 2, 3, 4, 5, np.inf]  # Define thresholds based on your
problem
```

```
# Discretize predicted values into classes

predicted_classes = np.digitize(rf_predictions, bins=thresholds) - 1  #
Subtract 1 to make classes start from 0



# Discretize true values into classes

true_classes = np.digitize(y_test, bins=thresholds) - 1  # Subtract 1 to make
classes start from 0



# Calculate confusion matrix

conf_matrix = confusion_matrix(true_classes, predicted_classes)



# Print confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d', cbar=False)

plt.xlabel('Predicted Class')

plt.ylabel('True Class')

plt.title('Confusion Matrix')

plt.show()
```

Gradient Boosting:

```
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.metrics import accuracy_score, confusion_matrix

import pandas as pd

import matplotlib.pyplot as plt



# Define a list of shrinkage values to try
```

```
shrinkage_values = np.arange(0.01, 0.21, 0.01).tolist()


# Initialize lists to store results

test_errors = []

confusion_matrices = []


# Iterate over shrinkage values

for shrinkage in shrinkage_values:

    # Train Gradient Boosting model with current shrinkage value

    gb_model = GradientBoostingClassifier(learning_rate=shrinkage,
n_estimators=100, subsample=0.8, max_depth=3, random_state=123)



    # Fit the model to the training data

    gb_model.fit(X_train, y_train)



    # Make predictions on the test data

    test_preds = gb_model.predict(X_test)



    # Calculate test error

    accuracy = accuracy_score(y_test, test_preds)

    test_error = 1 - accuracy

    test_errors.append(test_error)



    # Calculate confusion matrix

    confusion_matrices.append(confusion_matrix(y_test, test_preds))
```

```
# Select the best shrinkage value based on test error

best_shrinkage_idx = test_errors.index(min(test_errors))

best_shrinkage = shrinkage_values[best_shrinkage_idx]

print("Best shrinkage value:", best_shrinkage)


# Plot shrinkage vs. test error

plt.figure(figsize=(8, 6))

plt.plot(shrinkage_values, test_errors, marker='o', linestyle='-')

plt.title('Shrinkage vs. Test Error')

plt.xlabel('Shrinkage')

plt.ylabel('Test Error')

plt.xticks(shrinkage_values)

plt.grid(True)

plt.show()


# Train the Gradient Boosting model with the best shrinkage value

best_gb_model = GradientBoostingClassifier(learning_rate=best_shrinkage,
n_estimators=100, subsample=0.8, max_depth=3, random_state=123)

best_gb_model.fit(X_train, y_train)


# Make predictions on the test data

test_preds = best_gb_model.predict(X_test)


# Calculate test error
```

```
accuracy = accuracy_score(y_test, test_preds)

best_test_error = 1 - accuracy

print("Best model Test Error:", best_test_error)



conf_matrix = confusion_matrix(y_test, test_preds)


# Plot confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion Matrix for Best Model')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()



# Extract feature importance

feature_importance = pd.DataFrame({'Feature': X.columns, 'Importance':
best_gb_model.feature_importances_})

top_10_features = feature_importance.sort_values(by='Importance',
ascending=False).head(10)

print("Top 10 Feature Importance:")

print(top_10_features)
```

Multiclass Model comparison:

```
import matplotlib.pyplot as plt



# Test errors of each model

models = ['Decision Tree', 'Pruned Decision Tree', 'Bagging', 'Random Forest
(Tuned mtry)', 'Gradient Boosting']

test_errors = [decision_test_error, pruned_test_error, bagging_test_error,
errors[min_error_index], best_test_error]



# Plot model comparison

plt.figure(figsize=(10, 6))

plt.bar(models, test_errors, color=['blue', 'orange', 'green', 'red',
'purple'])

plt.title('Model Comparison: Test Error')

plt.xlabel('Model')

plt.ylabel('Test Error')

plt.ylim(0, max(test_errors) + 0.1)  # Adjust ylim for better visualization

plt.xticks(rotation=45)

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()
```

Method 3: Regressor

```
# Select relevant columns

df_demog = df[demographic_cols]

df_youth = df.loc[:, 'schfelt':'rlgfrnd']  # Assuming SCHFELT to RLGFRND are
the column names
```

```
ircigage = df[df['ircigage'] != 991]['ircigage']


#Concatenate the selected data frames horizontally (by columns)

ircigage_data = pd.concat([df_demog, df_youth, ircigage], axis=1)

ircigage_data= ircigage_data.dropna()


# Print the dimensions of the resulting dataset

print(ircigage_data.shape)

ircigage_data

X = ircigage_data.drop(columns=['ircigage'])

y = ircigage_data['ircigage']


# Split data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=123)
```

Decision Tree Regression:

```
import optuna

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.tree import DecisionTreeRegressor

from sklearn.metrics import mean_squared_error, confusion_matrix


# Split the data into train and test sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)



# Define the objective function for Decision Tree

def objective(trial):

    params = {

        'max_depth': trial.suggest_int('max_depth', 3, 10),

        'min_samples_split': trial.suggest_int('min_samples_split', 2, 20),

        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 10)

    }

    model = DecisionTreeRegressor(**params)

    return -cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error').mean()



# Create a study object and optimize the objective function

study = optuna.create_study(direction='minimize')

study.optimize(objective, n_trials=50)



# Get the best hyperparameters found by Optuna

best_params = study.best_params



# Train the model using the best hyperparameters

best_model = DecisionTreeRegressor(**best_params)

best_model.fit(X_train, y_train)
```

```
# Evaluate the model on the test set
test_predictions = best_model.predict(X_test)
deciison_test_mse = mean_squared_error(y_test, test_predictions)
print("Test MSE for Decision Tree:", deciison_test_mse)

importances = best_model.feature_importances_
indices = np.argsort(importances)[::-1]

# Extract the top 5 features and their importances
top_feature_indices = indices[:5]
top_feature_names = X.columns[top_feature_indices]
top_feature_importances = importances[top_feature_indices]

# Create a DataFrame to display the top features and their importances
top_features_df = pd.DataFrame({
    'Feature': top_feature_names,
    'Importance': top_feature_importances
})

print("Top 5 important features:")
print(top_features_df)
plt.figure(figsize=(20, 10))
plot_tree(best_model, filled=True, feature_names=X.columns)
plt.show()
```

Random Forest:

```
import optuna
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np


# Define objective function for Optuna
def objective(trial):
    # Define hyperparameters to optimize
    n_estimators = trial.suggest_int('n_estimators', 50, 200)
    max_depth = trial.suggest_int('max_depth', 3, 20)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 10)
    min_samples_leaf = trial.suggest_int('min_samples_leaf', 1, 4)

    # Train Random Forest model
    model = RandomForestRegressor(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        min_samples_leaf=min_samples_leaf,
        random_state=42
```

```
    )

    model.fit(X_train, y_train)



    # Evaluate model on validation set

    predictions = model.predict(X_valid)

    mse = mean_squared_error(y_valid, predictions)

    return mse



# Split data into train, validation, and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)



# Run Optuna optimization

study = optuna.create_study(direction='minimize')

study.optimize(objective, n_trials=50)



# Get best hyperparameters

best_params = study.best_params

print("Best hyperparameters:", best_params)



# Train Random Forest model with best hyperparameters

best_model = RandomForestRegressor(**best_params, random_state=42)

best_model.fit(X_train, y_train)
```

```
# Evaluate the best model on the test set
test_predictions = best_model.predict(X_test)
random_test_mse = mean_squared_error(y_test, test_predictions)
print("Test MSE:", random_test_mse)

# Get feature importances and sort them
importances = best_model.feature_importances_
indices = np.argsort(importances)[::-1]

# Extract the top 5 features and their importances
top_feature_indices = indices[:5]
top_feature_names = X.columns[top_feature_indices]
top_feature_importances = importances[top_feature_indices]

# Create a DataFrame to display the top features and their importances
top_features_df = pd.DataFrame({
    'Feature': top_feature_names,
    'Importance': top_feature_importances
})

print("Top 5 important features:")
print(top_features_df)
```

Gradient Boosting:

```
import optuna

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import GradientBoostingRegressor

from sklearn.metrics import mean_squared_error



# Split the data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)



# Define the objective function for Gradient Boosting

def objective(trial):

    params = {

        'n_estimators': trial.suggest_int('n_estimators', 50, 200),

        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.2),

        'max_depth': trial.suggest_int('max_depth', 3, 10),

        'min_samples_split': trial.suggest_int('min_samples_split', 2, 20),

        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 10)

    }

    model = GradientBoostingRegressor(**params)

    return -cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error').mean()
```

```
# Create a study object and optimize the objective function
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)

# Get the best hyperparameters found by Optuna
best_params = study.best_params

# Train the model using the best hyperparameters
best_model = GradientBoostingRegressor(**best_params)
best_model.fit(X_train, y_train)

# Evaluate the model on the test set
test_predictions = best_model.predict(X_test)
gradient_test_mse = mean_squared_error(y_test, test_predictions)
print("Test MSE for Gradient Boosting:", gradient_test_mse)
# Get feature importances from the best model
feature_importances_gb = best_model.feature_importances_

# Get indices of the top 5 features
top_feature_indices_gb = feature_importances_gb.argsort()[::-1][:5]

# Get the names and importance values of the top 5 features
top_features_gb = X.columns[top_feature_indices_gb]
top_feature_values_gb = feature_importances_gb[top_feature_indices_gb]
```

```
print("Top 5 features for Gradient Boosting:")

for feature, importance in zip(top_features_gb, top_feature_values_gb):

    print(f"{feature}: {importance}")
```

Svr:

```
import optuna

import pandas as pd

from sklearn.model_selection import train_test_split, cross_val_score

from sklearn.svm import SVR

from sklearn.metrics import mean_squared_error

from sklearn.inspection import permutation_importance



# Split the data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)



# Define the objective function for SVR

def objective(trial):

    params = {

        'C': trial.suggest_loguniform('C', 0.1, 10),

        'epsilon': trial.suggest_loguniform('epsilon', 0.001, 1),

        'kernel': trial.suggest_categorical('kernel', ['linear', 'rbf',
'poly'])

    }

    model = SVR(**params)
```

```
    return -cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error').mean()



# Create a study object and optimize the objective function

study = optuna.create_study(direction='minimize')

study.optimize(objective, n_trials=50)



# Get the best hyperparameters found by Optuna

best_params = study.best_params



# Train the model using the best hyperparameters

best_model = SVR(**best_params)

best_model.fit(X_train, y_train)



# Calculate permutation feature importance

perm_importance = permutation_importance(best_model, X_test, y_test,
n_repeats=30, random_state=42)



# Get the indices of the top 5 features

top_feature_indices_svr = perm_importance.importances_mean.argsort()[::-
1][:5]



# Get the names and importance values of the top 5 features

top_features_svr = X.columns[top_feature_indices_svr]

top_feature_values_svr =
perm_importance.importances_mean[top_feature_indices_svr]
```

```
print("Top 5 features for SVR:")

for feature, importance in zip(top_features_svr, top_feature_values_svr):

    print(f"{feature}: {importance}")


# Evaluate the model on the test set

test_predictions = best_model.predict(X_test)

svr_test_mse = mean_squared_error(y_test, test_predictions)

print("Test MSE for SVR:", svr_test_mse)
```

Knn:

```
import optuna

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error


# Split the data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)


# Define the objective function for KNN

def objective(trial):

    params = {

        'n_neighbors': trial.suggest_int('n_neighbors', 3, 15),
```

```
        'weights': trial.suggest_categorical('weights', ['uniform',
'distance']),

        'p': trial.suggest_int('p', 1, 2)  # Using 1 for Manhattan distance
and 2 for Euclidean distance

    }

    model = KNeighborsRegressor(**params)

    return -cross_val_score(model, X_train, y_train, cv=5,
scoring='neg_mean_squared_error').mean()



# Create a study object and optimize the objective function

study = optuna.create_study(direction='minimize')

study.optimize(objective, n_trials=50)



# Get the best hyperparameters found by Optuna

best_params = study.best_params



# Train the model using the best hyperparameters

best_model = KNeighborsRegressor(**best_params)

best_model.fit(X_train, y_train)



# Evaluate the model on the test set

test_predictions = best_model.predict(X_test)

knn_test_mse = mean_squared_error(y_test, test_predictions)

print("Test MSE for KNN:", knn_test_mse)
```

Regression Model Comparison:

```
import matplotlib.pyplot as plt



# Define models and their corresponding MSE

models = ['Decision Tree', 'Random Forest', 'Gradient Boosting', 'SVR',
'KNN']

mse_scores = [deciison_test_mse, random_test_mse,
gradient_test_mse,svr_test_mse, knn_test_mse]  # Replace with actual MSE
values



colors = ['skyblue', 'lightgreen', 'salmon', 'lightcoral', 'gold']



# Plot comparison

plt.figure(figsize=(10, 6))

plt.bar(models, mse_scores, color=colors)

plt.title('Comparison of Test MSE for Different Models')

plt.xlabel('Models')

plt.ylabel('Test MSE')

plt.xticks(rotation=45)

plt.show()
```