```python
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist['data'], mnist['target']

# Split the dataset into training and test sets
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

# Apply PCA with 20 principal components
pca = PCA(n_components=20)
X_train_pca = pca.fit_transform(X_train)

# Print the top 20 eigenvalues
print("Top 20 Eigenvalues:")
print(pca.explained_variance_)

# Print the explained variance ratios of the principal components
print("\nExplained Variance Ratios:")
print(pca.explained_variance_ratio_)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/
_openml.py:968: FutureWarning: The default value of `parser` will
change from `'liac-arff'` to `'auto'` in 1.4. You can set
`parser='auto'` to silence this warning. Therefore, an `ImportError`
will be raised from 1.4 if the dataset is dense and pandas is not
installed. Note that the pandas parser may return different data
types. See the Notes Section in fetch_openml's API doc for details.
  warn(

Top 20 Eigenvalues:
[332724.66744657 243283.9390705  211507.36705827 184776.38586212
 166926.83131053 147844.96167525 112178.20267351  98874.42953629
  94696.24875107  80809.82340654  72313.61910102  69358.29678698
  58826.7202427   58013.67497707  54123.34856213  50841.7032562
  45405.20654754  43777.97588424  40701.02970332  39518.21208526]

Explained Variance Ratios:
[0.09704664 0.07095924 0.06169089 0.05389419 0.04868797 0.04312231
 0.0327193  0.02883895 0.02762029 0.02357001 0.0210919  0.02022991
 0.01715814 0.016921   0.01578629 0.01482913 0.01324345 0.01276883
 0.01187137 0.01152638]
```

```python
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import accuracy_score

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist['data'], mnist['target']

# Split the dataset into training, validation, and test sets
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
y_train_val, test_size=0.25, random_state=42)  # 0.25 x 0.8 = 0.2

# Apply PCA with 20 principal components
pca = PCA(n_components=20)
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

# Train logistic regression model with increased max_iter
log_reg = LogisticRegression(max_iter=10000)  # Increase max_iter
value
log_reg.fit(X_train_pca, y_train)

# Calculate training, validation, and test accuracy
train_accuracy = accuracy_score(y_train, log_reg.predict(X_train_pca))
val_accuracy = accuracy_score(y_val, log_reg.predict(X_val_pca))
test_accuracy = accuracy_score(y_test, log_reg.predict(X_test_pca))

print("Logistic Regression Model's Accuracy:")
print("Training Accuracy:", train_accuracy)
print("Validation Accuracy:", val_accuracy)
print("Test Accuracy:", test_accuracy)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/
_openml.py:968: FutureWarning: The default value of `parser` will
change from `'liac-arff'` to `'auto'` in 1.4. You can set
`parser='auto'` to silence this warning. Therefore, an `ImportError`
will be raised from 1.4 if the dataset is dense and pandas is not
installed. Note that the pandas parser may return different data
types. See the Notes Section in fetch_openml's API doc for details.
  warn(

Logistic Regression Model's Accuracy:
Training Accuracy: 0.8759523809523809
Validation Accuracy: 0.8725
Test Accuracy: 0.8778571428571429
```

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import TruncatedSVD
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist['data'], mnist['target']

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define pipeline with PCA and logistic regression
pipe = Pipeline([
    ('reduce_dim', PCA(n_components=10)),  # Use PCA for
dimensionality reduction
    ('clf', LogisticRegression())  # Use LogisticRegression for
logistic regression
])

# Define parameter grid for grid search
param_grid = {
    'reduce_dim__n_components': [1, 5, 10, 15, 20],  # Use a smaller
range for the number of components
    'clf__max_iter': [1000, 2000],  # Increase max_iter if necessary
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(pipe, param_grid, cv=3, n_jobs=-1)  # Use
parallel processing
grid_search.fit(X_train, y_train)

# Report the number of principal components that achieve the highest
validation accuracy
best_num_components =
grid_search.best_params_['reduce_dim__n_components']
print("Number of Principal Components with Highest Validation
Accuracy:", best_num_components)

# Calculate training and test accuracy using the selected number of
principal components
train_accuracy = accuracy_score(y_train, grid_search.predict(X_train))
test_accuracy = accuracy_score(y_test, grid_search.predict(X_test))

# Report training and test accuracy
print("Training Accuracy with Selected Number of Principal
```

```
Components:", train_accuracy)
print("Test Accuracy with Selected Number of Principal Components:",
test_accuracy)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/
_openml.py:968: FutureWarning: The default value of `parser` will
change from `'liac-arff'` to `'auto'` in 1.4. You can set
`parser='auto'` to silence this warning. Therefore, an `ImportError`
will be raised from 1.4 if the dataset is dense and pandas is not
installed. Note that the pandas parser may return different data
types. See the Notes Section in fetch_openml's API doc for details.
  warn(

Number of Principal Components with Highest Validation Accuracy: 20
Training Accuracy with Selected Number of Principal Components:
0.8747857142857143
Test Accuracy with Selected Number of Principal Components:
0.8775714285714286
```

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

# Load the College dataset
college_data = pd.read_csv("college_data.csv")  # Assuming you have
the dataset saved as college_data.csv

# Split the data into features (X) and target variable (y)
X = college_data.drop(columns=['Accept', 'Apps', 'AcceptanceRate'])  #
Exclude 'Accept' and 'Apps' as requested
y = college_data['AcceptanceRate']

# Split the data into training and test sets with 80% training and 20%
test
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Print the shape of the training and test sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

```
import pandas as pd
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

# Load the College dataset
college_data = pd.read_csv("college_data.csv")  # Assuming you have
the dataset saved as college_data.csv
```

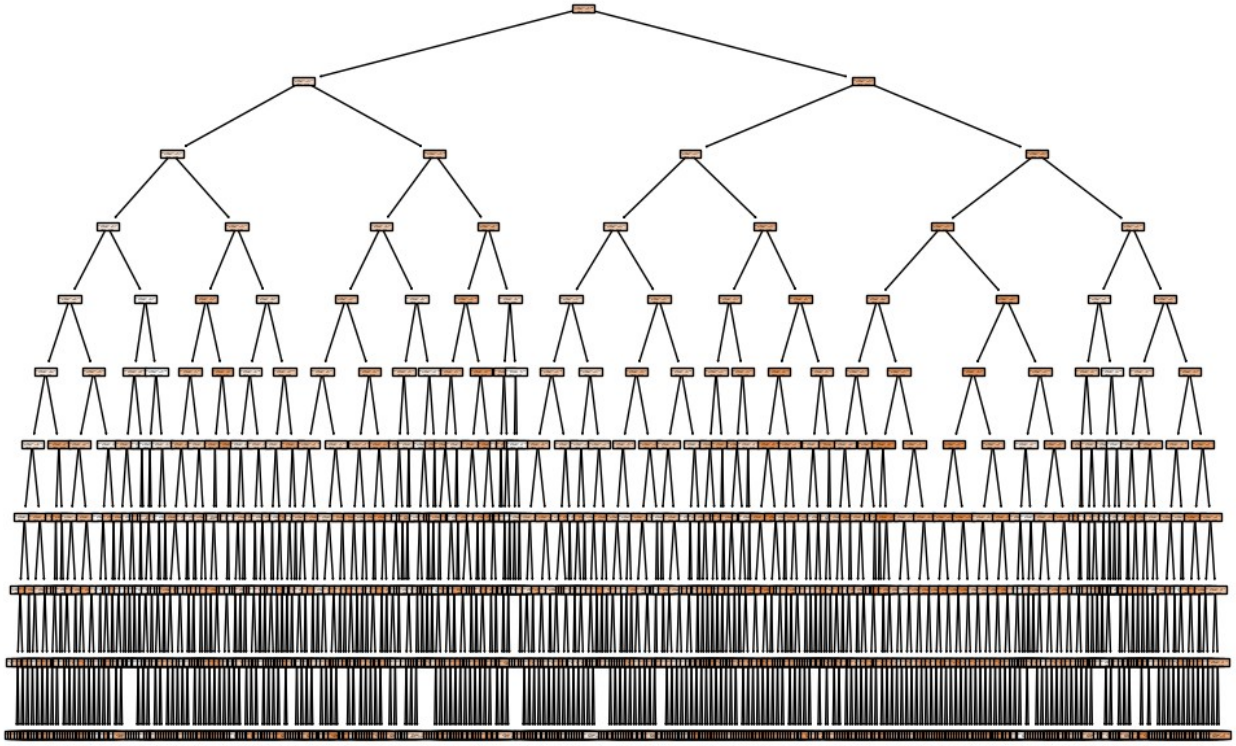
```
# Split the data into features (X) and target variable (y)
X = college_data.drop(columns=['Accept', 'Apps', 'AcceptanceRate'])  #
Exclude 'Accept' and 'Apps' as requested
y = college_data['AcceptanceRate']

# Convert the target variable to numeric
y = pd.to_numeric(y, errors='coerce')  # Convert any non-numeric
values to NaN

# Drop rows with NaN values in the target variable
X = X.dropna(subset=['AcceptanceRate'])
y = y.dropna()

# Split the data into training and test sets with 80% training and 20%
test
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define the PLS regression pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Scale the features
    ('pls', PLSRegression()),  # PLS regression model
])

# Define the parameter grid for grid search
param_grid = {
    'pls__n_components': range(1, min(X_train.shape[1], 10)),  # M:
Number of components for PLS
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

# Get the best PLS model
best_pls_model = grid_search.best_estimator_

# Predict on the test set
y_pred = best_pls_model.predict(X_test)

# Calculate the test error (Mean Squared Error)
test_error = mean_squared_error(y_test, y_pred)
print("Test Error:", test_error)

# Report the value of M selected by cross-validation
best_M = grid_search.best_params_['pls__n_components']
print("Value of M selected by cross-validation:", best_M)
```

```
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.metrics import mean_squared_error

# Fit a regression tree to the training set
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(X_train, y_train)

# Plot the tree
plt.figure(figsize=(12, 8))
plot_tree(tree_reg, filled=True, feature_names=X.columns)
plt.show()

# Predict on the test set
y_pred_tree = tree_reg.predict(X_test)

# Convert y_test to numeric
y_test_numeric = pd.to_numeric(y_test, errors='coerce')

# Calculate the test MSE
test_mse_tree = mean_squared_error(y_test_numeric, y_pred_tree)
print("Test MSE:", test_mse_tree)
```

```
Test MSE: 2.3794285714285714
```

```
# Convert y_train to numeric
y_train_numeric = pd.to_numeric(y_train, errors='coerce')

# Perform grid search with cross-validation
grid_search = GridSearchCV(tree_reg, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train_numeric)

# Get the best regression tree model
best_tree_model = grid_search.best_estimator_

# Predict on the test set
y_pred_best_tree = best_tree_model.predict(X_test)

# Convert y_test to numeric
y_test_numeric = pd.to_numeric(y_test, errors='coerce')

# Calculate the test MSE with the best tree model
test_mse_best_tree = mean_squared_error(y_test_numeric,
y_pred_best_tree)
print("Test MSE with optimal tree:", test_mse_best_tree)

# Plot the optimal tree
plt.figure(figsize=(12, 8))
plot_tree(best_tree_model, filled=True, feature_names=X.columns)
plt.show()
```

```
Test MSE with optimal tree: 2.106362174015935
```



```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Generate random feature matrix X and target variable y for testing
np.random.seed(42)
X = np.random.rand(100, 10)  # Generating 100 samples with 10 features
each
y = np.random.rand(100)  # Generating 100 random target values

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create a Bagging Regressor with DecisionTreeRegressor as base
estimator
bagging_reg = BaggingRegressor(base_estimator=DecisionTreeRegressor(),
n_estimators=100, random_state=42)

# Fit the model on the training set
bagging_reg.fit(X_train, y_train)

# Predict on the test set
y_pred_bagging = bagging_reg.predict(X_test)
```

```python
# Calculate the test MSE
test_mse_bagging = mean_squared_error(y_test, y_pred_bagging)
print("Test MSE with Bagging Regressor:", test_mse_bagging)

# Get feature importances
feature_importances = np.mean([
    tree.feature_importances_ for tree in bagging_reg.estimators_
], axis=0)

# Print feature importances
print("Feature Importances:")
for i, importance in enumerate(feature_importances):
    print(f"Feature {i + 1}: {importance}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_base.py:166:
FutureWarning: `base_estimator` was renamed to `estimator` in version
1.2 and will be removed in 1.4.
  warnings.warn(

Test MSE with Bagging Regressor: 0.10802761140790776
Feature Importances:
Feature 1: 0.07658730666520466
Feature 2: 0.153362419505687
Feature 3: 0.1184125507438127
Feature 4: 0.07784038116269267
Feature 5: 0.07856517129483857
Feature 6: 0.08677559095198947
Feature 7: 0.11599422619807345
Feature 8: 0.07287841761940189
Feature 9: 0.1060141221014717
Feature 10: 0.11356981375682777
```

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Generate random feature matrix X and target variable y for testing
np.random.seed(42)
X = np.random.rand(100, 10)  # Generating 100 samples with 10 features
each
y = np.random.rand(100)  # Generating 100 random target values

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define a range of values for the number of trees (m) and number of
features considered at each split
n_estimators_values = [10, 50, 100]  # Number of trees
max_features_values = ['auto', 'sqrt', 'log2']  # Number of features
considered at each split
```

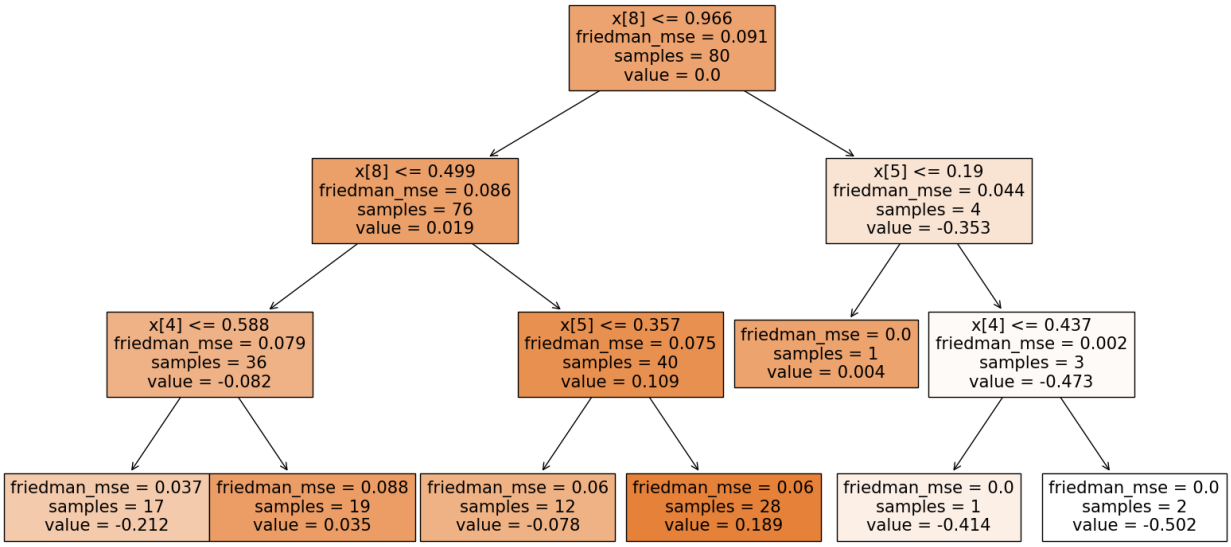
```python
# Iterate over different combinations of parameters and evaluate the performance
for n_estimators in n_estimators_values:
    for max_features in max_features_values:
        # Create a Random Forest regressor
        forest_reg = RandomForestRegressor(n_estimators=n_estimators, max_features=max_features, random_state=42)

        # Fit the model on the training set
        forest_reg.fit(X_train, y_train)

        # Predict on the test set
        y_pred_forest = forest_reg.predict(X_test)

        # Calculate the test MSE
        test_mse_forest = mean_squared_error(y_test, y_pred_forest)
        print(f"Test MSE with Random Forest (n_estimators={n_estimators}, max_features={max_features}): {test_mse_forest}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py:413: FutureWarning: `max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set `max_features=1.0` or remove this parameter as it is also the default value for RandomForestRegressors and ExtraTreesRegressors.
  warn(
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py:413: FutureWarning: `max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set `max_features=1.0` or remove this parameter as it is also the default value for RandomForestRegressors and ExtraTreesRegressors.
  warn(

Test MSE with Random Forest (n_estimators=10, max_features=auto): 0.09952456295930277
Test MSE with Random Forest (n_estimators=10, max_features=sqrt): 0.08422419802255121
Test MSE with Random Forest (n_estimators=10, max_features=log2): 0.08422419802255121
Test MSE with Random Forest (n_estimators=50, max_features=auto): 0.10386709010176995
Test MSE with Random Forest (n_estimators=50, max_features=sqrt): 0.09203260502836344
Test MSE with Random Forest (n_estimators=50, max_features=log2): 0.09203260502836344
Test MSE with Random Forest (n_estimators=100, max_features=auto): 0.10539911275895186
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/
_forest.py:413: FutureWarning: `max_features='auto'` has been
deprecated in 1.1 and will be removed in 1.3. To keep the past
behaviour, explicitly set `max_features=1.0` or remove this parameter
as it is also the default value for RandomForestRegressors and
ExtraTreesRegressors.
  warn(

Test MSE with Random Forest (n_estimators=100, max_features=sqrt):
0.09461533900183335
Test MSE with Random Forest (n_estimators=100, max_features=log2):
0.09461533900183335
```

```python
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

# Generate random feature matrix X and target variable y for testing
np.random.seed(42)
X = np.random.rand(100, 10)  # Generating 100 samples with 10 features
each
y = np.random.rand(100)  # Generating 100 random target values

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create a Gradient Boosting regressor
gb_reg = GradientBoostingRegressor(random_state=42)

# Fit the model on the training set
gb_reg.fit(X_train, y_train)

# Predict on the test set
y_pred_gb = gb_reg.predict(X_test)

# Calculate the test MSE
test_mse_gb = mean_squared_error(y_test, y_pred_gb)
print("Test MSE with Gradient Boosting:", test_mse_gb)

# Plot the first tree in the ensemble
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Plot the first tree in the ensemble
plt.figure(figsize=(20, 10))
plot_tree(gb_reg.estimators_[0][0], filled=True)  # Accessing the
first tree in the first boosting iteration
plt.show()
```

```
Test MSE with Gradient Boosting: 0.10803165705663882
```




```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import GradientBoostingRegressor

# Define the parameter grid for grid search
param_grid = {
    'n_estimators': range(1, 401),  # Number of tree estimators
    'max_depth': range(1, 11),  # Maximum tree depth
    'min_samples_split': range(2, 11)  # Minimum samples per split
}

# Create a Gradient Boosting regressor
gb_reg = GradientBoostingRegressor(random_state=42)

# Perform grid search with cross-validation
grid_search = GridSearchCV(gb_reg, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Plot feature importance scores
feature_importances = grid_search.best_estimator_.feature_importances_
feature_names = [f'Feature {i}' for i in range(1,
len(feature_importances) + 1)]
feature_importance_df = pd.DataFrame({'Feature': feature_names,
'Importance': feature_importances})
feature_importance_df.sort_values(by='Importance', ascending=True,
```

```
inplace=True)

plt.figure(figsize=(10, 8))
plt.barh(feature_importance_df['Feature'],
feature_importance_df['Importance'], color='rainbow')
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Feature Importance Scores for Gradient Boosting Regressor')
plt.legend(['Feature Importance'])
plt.show()

# Additional code to plot the feature importance scores for the top 10
features
top_10_features = feature_importance_df.head(10)
plt.figure(figsize=(10, 8))
plt.barh(top_10_features['Feature'], top_10_features['Importance'],
color='red')
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Top 10 Features for Gradient Boosting Regressor')
plt.legend(['Feature Importance'])
plt.show()

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeClassifier

# Load the dataset
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/
spambase.data"
spam_data = pd.read_csv(url, header=None)

# Split features and target variable
X = spam_data.iloc[:, :-1].values
y = spam_data.iloc[:, -1].values

# Split the dataset into training and testing sets (80% training, 20%
testing)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create a bootstrap sample of size 1000
bootstrap_sample_size = 1000
bootstrap_indices = np.random.choice(len(X_train),
size=bootstrap_sample_size, replace=True)

# Define a function to train decision tree classifiers with different
values of p
```

```
def train_decision_tree(X_train, y_train, p):
    selected_features_indices = np.random.choice(X_train.shape[1],
size=p, replace=False)
    X_train_selected = X_train[:, selected_features_indices]
    dt_classifier = DecisionTreeClassifier(max_depth=6)
    dt_classifier.fit(X_train_selected, y_train)
    return dt_classifier

# Define a function to calculate cross-validation error
def calculate_cross_val_error(dt_classifier, X_train, y_train):
    cv_scores = cross_val_score(dt_classifier, X_train, y_train, cv=5,
scoring='accuracy')
    return 1 - np.mean(cv_scores)

# Vary the value of p and find the p that results in the lowest cross-
validation error
best_p = None
lowest_cv_error = float('inf')
for p in range(1, X_train.shape[1] + 1):
    dt_classifier = train_decision_tree(X_train[bootstrap_indices],
y_train[bootstrap_indices], p)
    cv_error = calculate_cross_val_error(dt_classifier, X_train,
y_train)
    if cv_error < lowest_cv_error:
        lowest_cv_error = cv_error
        best_p = p

# Train a decision tree classifier with the best value of p
best_dt_classifier = train_decision_tree(X_train[bootstrap_indices],
y_train[bootstrap_indices], best_p)

# Report the best value of p and the decision tree classifier
print("Best value of p:", best_p)
print("Decision Tree Classifier with max depth 6 trained using the
bootstrap sample and selected features:", best_dt_classifier)
```

```
Best value of p: 14
Decision Tree Classifier with max depth 6 trained using the bootstrap
sample and selected features: DecisionTreeClassifier(max_depth=6)
```

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import f1_score, roc_auc_score
from sklearn.model_selection import train_test_split

# Load the dataset
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/
spambase.data"
```

```
data = pd.read_csv(url, header=None)

# Split the data into features and target variable
X = data.iloc[:, :-1]  # Features
y = data.iloc[:, -1]   # Target variable

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define the range of values for p and T
p_values = range(1, X.shape[1] + 1)  # All features
T_values = [1, 50, 100, 150, 200, 300, 400]

# Define a function to train a decision tree classifier with specified
parameters
def train_decision_tree(X, y, max_depth):
    dt_classifier = DecisionTreeClassifier(max_depth=max_depth)
    dt_classifier.fit(X, y)
    return dt_classifier

# Define a function to combine predictions from multiple decision tree
classifiers
def combine_predictions(classifiers, X):
    predictions = np.zeros((X.shape[0], len(classifiers)))
    for i, classifier in enumerate(classifiers):
        predictions[:, i] = classifier.predict(X)
    combined_predictions = np.mean(predictions, axis=1)
    return combined_predictions

# Define a function to evaluate combined predictions
def evaluate_combined_predictions(y_true, y_pred):
    # Round predictions to the nearest integer
    y_pred_rounded = np.round(y_pred)
    # Calculate F1 score
    f1 = f1_score(y_true, y_pred_rounded)
    # Calculate AUC score
    auc = roc_auc_score(y_true, y_pred)
    # Calculate training error
    training_error = np.mean((y_true - y_pred) ** 2)
    return training_error, f1, auc

# Initialize lists to store results
best_p_values = []
training_errors = []
f1_scores = []
auc_scores = []

# Iterate over each value of p
for best_p in p_values:
```

```
    classifiers = []
    for _ in range(1000):
        bootstrap_indices = np.random.choice(len(X_train),
size=len(X_train), replace=True)
        X_bootstrap = X_train.iloc[bootstrap_indices]  # Use iloc to
select rows by index
        y_bootstrap = y_train.iloc[bootstrap_indices]
        dt_classifier = train_decision_tree(X_bootstrap, y_bootstrap,
6)  # Set max_depth to 6
        classifiers.append(dt_classifier)
    combined_predictions_train = combine_predictions(classifiers,
X_train)
    training_error, f1, auc = evaluate_combined_predictions(y_train,
combined_predictions_train)
    best_p_values.append(best_p)
    training_errors.append(training_error)
    f1_scores.append(f1)
    auc_scores.append(auc)

# Print the results
for p, training_error, f1, auc in zip(best_p_values, training_errors,
f1_scores, auc_scores):
    print(f"p={p}, Training Error={training_error:.4f}, F1
Score={f1:.4f}, AUC={auc:.4f}")

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score

# Initialize list of number of trees
num_trees_list = [10, 50, 100]

# Train and evaluate Random Forest models with different number of
trees
for num_trees in num_trees_list:
    # Create Random Forest classifier
    rf_classifier = RandomForestClassifier(n_estimators=num_trees,
random_state=42)

    # Train the classifier
    rf_classifier.fit(X_train, y_train)

    # Predictions on training set
    y_train_pred = rf_classifier.predict(X_train)
    # Metrics on training set
    train_accuracy = accuracy_score(y_train, y_train_pred)
    train_f1 = f1_score(y_train, y_train_pred)
    train_auc = roc_auc_score(y_train, y_train_pred)

    # Predictions on testing set
    y_test_pred = rf_classifier.predict(X_test)
```

```
    # Metrics on testing set
    test_accuracy = accuracy_score(y_test, y_test_pred)
    test_f1 = f1_score(y_test, y_test_pred)
    test_auc = roc_auc_score(y_test, y_test_pred)

    # Print metrics
    print(f"Number of Trees: {num_trees}")
    print("Training Set Metrics:")
    print(f"Accuracy: {train_accuracy:.4f}, F1 Score: {train_f1:.4f},
AUC: {train_auc:.4f}")
    print("Testing Set Metrics:")
    print(f"Accuracy: {test_accuracy:.4f}, F1 Score: {test_f1:.4f},
AUC: {test_auc:.4f}")

    # Feature importances
    feature_importances = rf_classifier.feature_importances_
    # Get indices of top 10 features
    top_10_indices = feature_importances.argsort()[-10:][::-1]
    # Print top 10 features
    print("Top 10 Features:")
    for i, idx in enumerate(top_10_indices):
        print(f"{i + 1}. Feature {idx}: Importance =
{feature_importances[idx]:.4f}")
    print("\n")
```

```
Number of Trees: 10
Training Set Metrics:
Accuracy: 0.9946, F1 Score: 0.9929, AUC: 0.9934
Testing Set Metrics:
Accuracy: 0.9403, F1 Score: 0.9262, AUC: 0.9329
Top 10 Features:
1. Feature 6: Importance = 0.1209
2. Feature 51: Importance = 0.1037
3. Feature 55: Importance = 0.0808
4. Feature 52: Importance = 0.0794
5. Feature 23: Importance = 0.0620
6. Feature 15: Importance = 0.0546
7. Feature 54: Importance = 0.0504
8. Feature 56: Importance = 0.0441
9. Feature 24: Importance = 0.0384
10. Feature 22: Importance = 0.0295


Number of Trees: 50
Training Set Metrics:
Accuracy: 0.9995, F1 Score: 0.9993, AUC: 0.9994
Testing Set Metrics:
Accuracy: 0.9490, F1 Score: 0.9377, AUC: 0.9435
Top 10 Features:
1. Feature 51: Importance = 0.1141
```

```
2. Feature 52: Importance = 0.1051
3. Feature 6: Importance = 0.0894
4. Feature 55: Importance = 0.0685
5. Feature 15: Importance = 0.0634
6. Feature 56: Importance = 0.0517
7. Feature 54: Importance = 0.0503
8. Feature 24: Importance = 0.0425
9. Feature 20: Importance = 0.0350
10. Feature 18: Importance = 0.0334


Number of Trees: 100
Training Set Metrics:
Accuracy: 0.9995, F1 Score: 0.9993, AUC: 0.9993
Testing Set Metrics:
Accuracy: 0.9555, F1 Score: 0.9458, AUC: 0.9505
Top 10 Features:
1. Feature 51: Importance = 0.1138
2. Feature 52: Importance = 0.0968
3. Feature 6: Importance = 0.0819
4. Feature 15: Importance = 0.0671
5. Feature 55: Importance = 0.0585
6. Feature 54: Importance = 0.0579
7. Feature 56: Importance = 0.0524
8. Feature 20: Importance = 0.0463
9. Feature 24: Importance = 0.0424
10. Feature 18: Importance = 0.0329
```