PS4

May 15, 2024

[62]:
```
# 1. (ISLP: Chapter 6, Question 9) In this exercise, we will predict the number of applications received using the other variables in the College data set.
```

[63]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
```

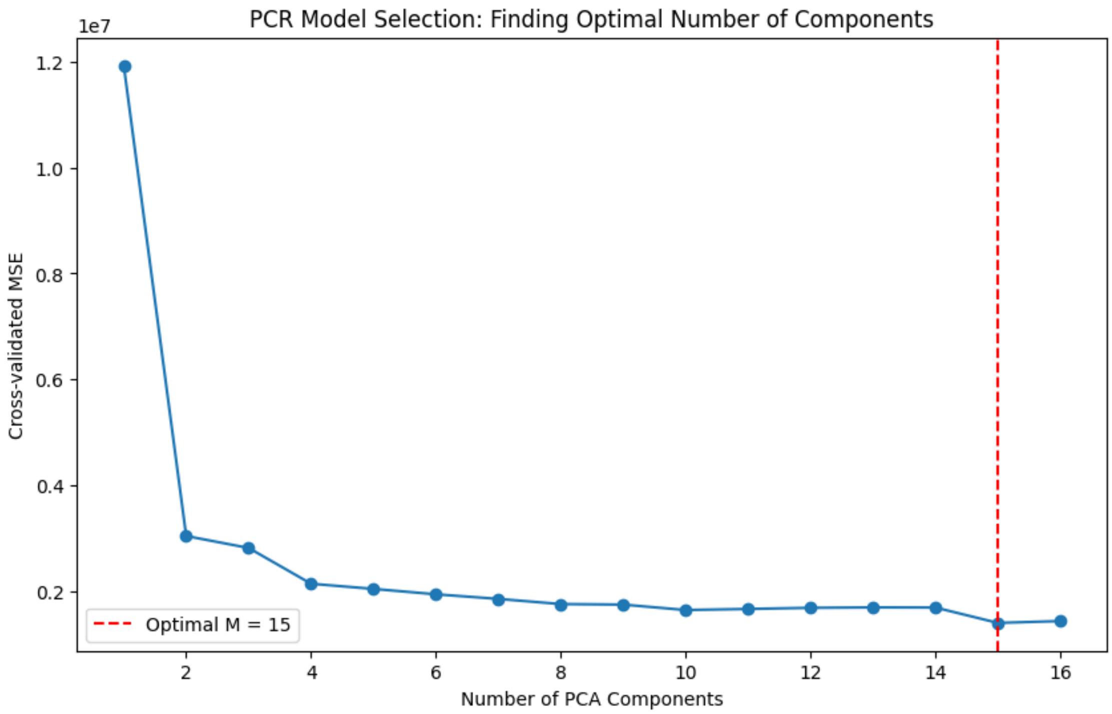
[64]:
```
# Part (a)
df = pd.read_csv('/Users/rouren/Desktop/24S ML/HW/ps4/Data-College.csv')

X = df.select_dtypes(include=[np.number]).drop(['Apps'], axis=1)
y = df['Apps']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=37)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

[65]:
```
# Part (b): Linear Regression
linear_model = LinearRegression()
linear_model.fit(X_train_scaled, y_train)
y_pred_linear = linear_model.predict(X_test_scaled)
mse_linear = mean_squared_error(y_test, y_pred_linear)
print(f'Linear Regression MSE: {mse_linear:.2f}')
```

```
Linear Regression MSE: 1236460.22
```

[66]:

```
# Part (c): Principal Components Regression with Cross-Validation

pca = PCA()
linear_pcr = LinearRegression()
pipeline_pcr = Pipeline([('pca', pca), ('linear', linear_pcr)])
kf = KFold(n_splits=10, shuffle=True, random_state=1)

mse_scores = []
components_range = range(1, X_train_scaled.shape[1] + 1)
for n_components in components_range:
    pipeline_pcr.set_params(pca__n_components=n_components)
    scores = -cross_val_score(pipeline_pcr, X_train_scaled, y_train, cv=kf,␣
↪scoring='neg_mean_squared_error')
    mse_scores.append(np.mean(scores))

optimal_components = np.argmin(mse_scores) + 1

plt.figure(figsize=(10, 6))
plt.plot(components_range, mse_scores, marker='o')
plt.xlabel('Number of PCA Components')
plt.ylabel('Cross-validated MSE')
plt.title('PCR Model Selection: Finding Optimal Number of Components')
plt.axvline(x=optimal_components, color='r', linestyle='--', label=f'Optimal M␣
↪= {optimal_components}')
plt.legend()
plt.show()

pipeline_pcr.set_params(pca__n_components=optimal_components)
pipeline_pcr.fit(X_train_scaled, y_train)
y_pred_pcr = pipeline_pcr.predict(X_test_scaled)
mse_pcr = mean_squared_error(y_test, y_pred_pcr)
print(f'PCR with Optimal Components ({optimal_components}) MSE: {mse_pcr:.2f}')
```




```
PCR with Optimal Components (15) MSE: 1517737.26
```

[67]:

```python
# Part (d): Partial Least Squares Regression (PLS)
mse_scores_pls = []
components_range_pls = range(1, min(15, X_train_scaled.shape[1] + 1))
for n_components in components_range_pls:
    pls = PLSRegression(n_components=n_components)
    scores = -cross_val_score(pls, X_train_scaled, y_train, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
    mse_scores_pls.append(np.mean(scores))

optimal_components_pls = np.argmin(mse_scores_pls) + 1
pls_final = PLSRegression(n_components=optimal_components_pls)
pls_final.fit(X_train_scaled, y_train)
y_pred_pls = pls_final.predict(X_test_scaled)
mse_pls = mean_squared_error(y_test, y_pred_pls)
print(f'PLS MSE: {mse_pls:.2f}, Optimal Components (PLS):␣
 ↪{optimal_components_pls}')
```

```
PLS MSE: 1335335.16, Optimal Components (PLS): 8
```

[68]:

```
# In comparing the performance of linear regression, Principal Components␣
↪Regression (PCR), and Partial Least Squares (PLS) regression on predicting␣
↪college applications, linear regression yielded the lowest Mean Squared␣
↪Error (MSE) of 1,236,460.22, indicating the most effective fit among the␣
↪models tested. PCR, despite using the maximum 15 components, performed the␣
↪worst with an MSE of 1,517,737.26, suggesting a potential loss of essential␣
↪predictive information during dimensionality reduction. PLS, with 8␣
↪components, performed better than PCR but still underperformed compared to␣
↪the baseline linear regression with an MSE of 1,335,335.16, implying that␣
↪while dimensionality reduction aimed to capture relevant information, it␣
↪still could not surpass the predictive power of using all available features␣
↪directly in linear regression. This outcome suggests that simpler models␣
↪without transformation of variables might be more effective for this␣
↪dataset, and dimensionality reduction techniques such as PCR and PLS may not␣
↪provide additional predictive benefit in this context.
```

[69]:
```
# 2. This question relates to the plots in the figure that follows (ISLP Figure␣
↪8.14)
```

[70]:
```
# 3. This question involves the OJ data set which is available on Canvas.
```

[71]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
from sklearn.metrics import confusion_matrix, accuracy_score
import numpy as np
from sklearn.model_selection import cross_val_score, KFold
```

[72]:
```
# Part (a)
oj_data = pd.read_csv('/Users/rouren/Desktop/24S ML/HW/ps4/Data-OJ.csv')

oj_data['Store7'] = oj_data['Store7'].map({'Yes': 1, 'No': 0})

train_set, test_set = train_test_split(oj_data, test_size=0.30, random_state=3)

print("Training set:")
print(train_set.head())
print("\nTest set:")
print(test_set.head())
```

```
Training set:
    Purchase  WeekofPurchase  StoreID  PriceCH  PriceMM  DiscCH  DiscMM  \
716       MM             267        3     1.99     2.09     0.1     0.0
386       MM             229        2     1.69     1.69     0.0     0.0
105       CH             245        2     1.89     2.09     0.0     0.0
```

```
163        CH              271         4     1.99     2.09     0.1     0.4
581        CH              258         7     1.86     2.18     0.0     0.0

     SpecialCH  SpecialMM   LoyalCH  SalePriceMM  SalePriceCH  PriceDiff  \
716          0          0  0.000104         2.09         1.89       0.20
386          0          0  0.165373         1.69         1.69       0.00
105          0          0  0.797050         2.09         1.89       0.20
163          1          0  0.985926         1.69         1.89      -0.20
581          0          0  0.680000         2.18         1.86       0.32

     Store7  PctDiscMM  PctDiscCH  ListPriceDiff  STORE
716       0   0.000000   0.050251           0.10      3
386       0   0.000000   0.000000           0.00      2
105       0   0.000000   0.000000           0.20      2
163       0   0.191388   0.050251           0.10      4
581       1   0.000000   0.000000           0.32      0

Test set:
    Purchase  WeekofPurchase  StoreID  PriceCH  PriceMM  DiscCH  DiscMM  \
354       MM             227        4     1.79     1.79     0.0    0.00
599       CH             260        7     1.86     2.13     0.0    0.24
478       CH             267        7     1.86     2.13     0.0    0.00
796       MM             263        1     1.76     1.99     0.0    0.40
955       CH             270        2     1.86     2.18     0.0    0.00

     SpecialCH  SpecialMM   LoyalCH  SalePriceMM  SalePriceCH  PriceDiff  \
354          0          1  0.500000         1.79         1.79       0.00
599          0          0  0.944165         1.89         1.86       0.03
478          1          0  0.692800         2.13         1.86       0.27
796          0          1  0.042950         1.59         1.76      -0.17
955          0          0  0.201954         2.18         1.86       0.32

     Store7  PctDiscMM  PctDiscCH  ListPriceDiff  STORE
354       0   0.000000        0.0           0.00      4
599       1   0.112676        0.0           0.27      0
478       1   0.000000        0.0           0.27      0
796       0   0.201005        0.0           0.23      1
955       0   0.000000        0.0           0.32      2
```

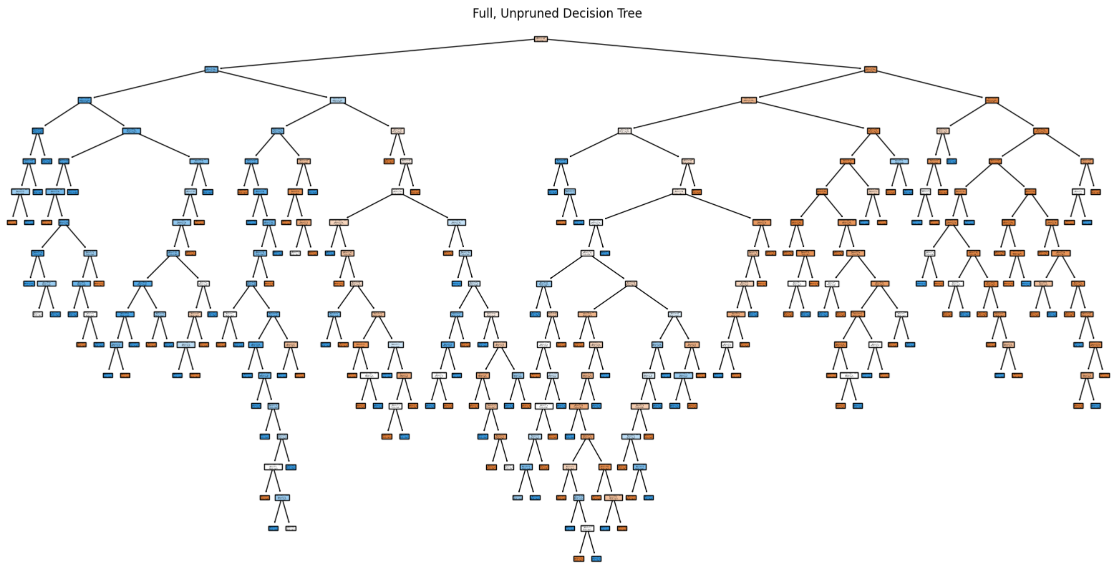
[73]:

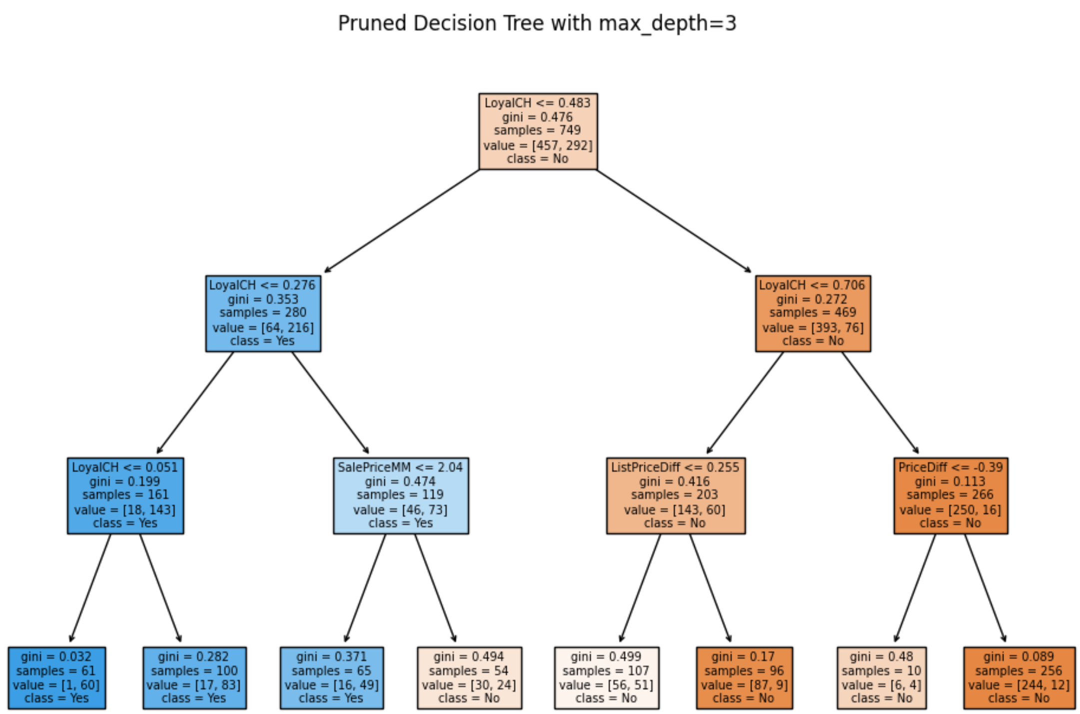
```
# Part (b)

X_train = train_set.drop(['Purchase'], axis=1)
y_train = train_set['Purchase']

tree_clf = DecisionTreeClassifier(random_state=2)

tree_clf.fit(X_train, y_train)
```

```python
# Predict on the training set
y_train_pred = tree_clf.predict(X_train)

training_accuracy = accuracy_score(y_train, y_train_pred)
training_error_rate = 1 - training_accuracy

print("Training Error Rate:", training_error_rate)
print("Number of Terminal Nodes (Leaves):", tree_clf.get_n_leaves())
```

```
Training Error Rate: 0.006675567423230944
Number of Terminal Nodes (Leaves): 154
```

[74]:
```python
# Part (c)

plt.figure(figsize=(20, 10))
plot_tree(tree_clf, filled=True, feature_names=X_train.columns,␣
 ↪class_names=['No', 'Yes'])
plt.title("Full, Unpruned Decision Tree")
plt.show()

pruned_tree_clf = DecisionTreeClassifier(random_state=2, max_depth=3)
pruned_tree_clf.fit(X_train, y_train)

plt.figure(figsize=(12, 8))
plot_tree(pruned_tree_clf, filled=True, feature_names=X_train.columns,␣
 ↪class_names=['No', 'Yes'])
plt.title("Pruned Decision Tree with max_depth=3")
plt.show()

num_terminal_nodes = pruned_tree_clf.get_n_leaves()
print("Number of Terminal Nodes:", num_terminal_nodes)
```






```
Number of Terminal Nodes: 8
```

[ ]:

```
# The initial leaf of the decision tree illustrates a data segment where
↪loyalty to the CH brand (LoyalCH) is 0.051 or lower. This node exhibits a
↪Gini impurity of 0.199, suggesting that the data within this node is fairly
↪uniform. A total of 161 samples fall into this category, with 18 classified
↪as CH and 143 as MM.
```

[75]:

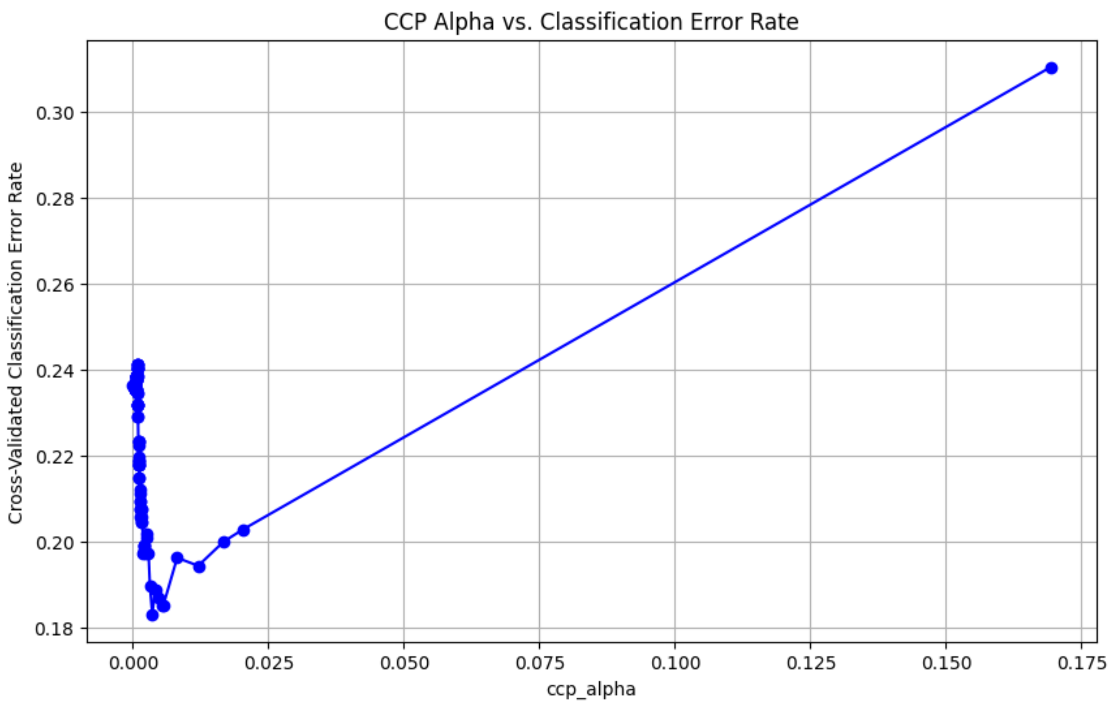
```python
# Part (d)

# Prepare test data
X_test = test_set.drop(['Purchase'], axis=1)
y_test = test_set['Purchase']

# Predict on the test set using the unpruned tree
y_test_pred = tree_clf.predict(X_test)

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_test_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Calculate test accuracy
test_accuracy = accuracy_score(y_test, y_test_pred)

# Calculate test error rate
test_error_rate = 1 - test_accuracy
print("Test Error Rate:", test_error_rate)
```

```
Confusion Matrix:
[[160  36]
 [ 40  85]]
Test Error Rate: 0.23676012461059193
```

[76]:

```python
# Part (e)

X = oj_data.drop(['Purchase'], axis=1)
y = oj_data['Purchase']

tree_clf = DecisionTreeClassifier(random_state=2)
tree_clf.fit(X, y)

path = tree_clf.cost_complexity_pruning_path(X, y)
ccp_alphas = path.ccp_alphas

kf = KFold(n_splits=5, shuffle=True, random_state=2)

cv_error_rates = []

for ccp_alpha in ccp_alphas:
```

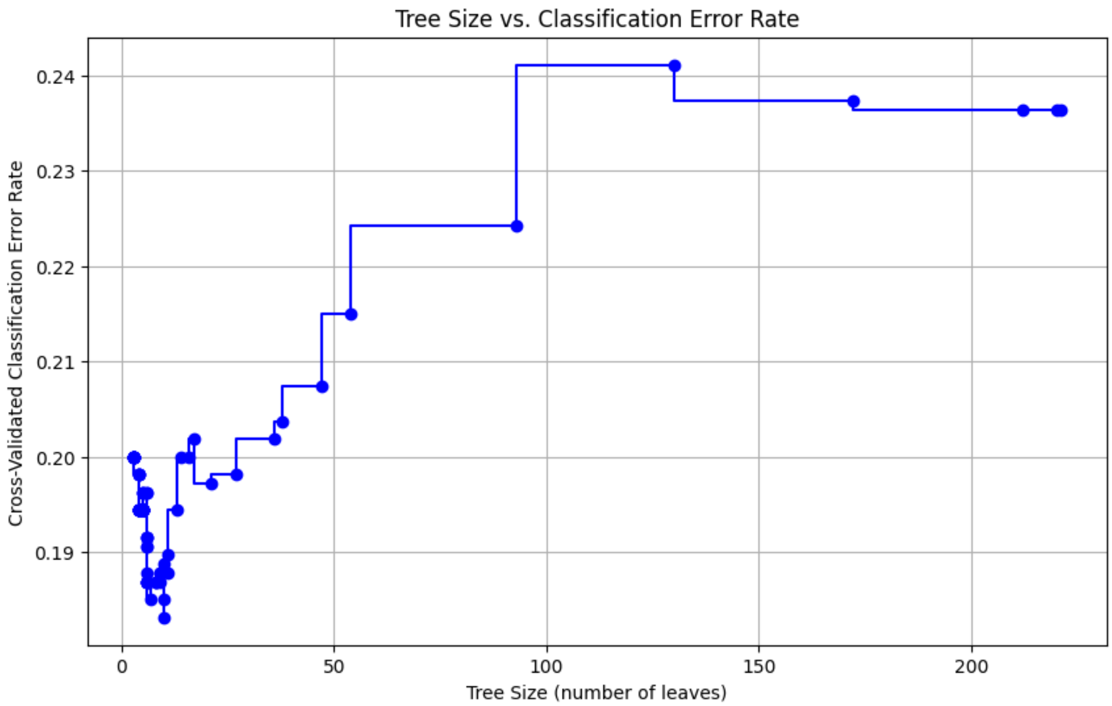
```
    clf = DecisionTreeClassifier(random_state=2, ccp_alpha=ccp_alpha)
    cv_scores = cross_val_score(clf, X, y, cv=kf, scoring='accuracy')
    cv_error_rates.append(1 - np.mean(cv_scores))

plt.figure(figsize=(10, 6))
plt.plot(ccp_alphas, cv_error_rates, marker='o', linestyle='-', color='blue')
plt.title('CCP Alpha vs. Classification Error Rate')
plt.xlabel('ccp_alpha')
plt.ylabel('Cross-Validated Classification Error Rate')
plt.grid(True)
plt.show()

optimal_alpha = ccp_alphas[np.argmin(cv_error_rates)]
print("Optimal ccp_alpha with the lowest error rate:", optimal_alpha)
```




```
Optimal ccp_alpha with the lowest error rate: 0.0035966593897716753
```

[77]:
```
# Part (f)

X = oj_data.drop(['Purchase'], axis=1)
y = oj_data['Purchase']

ccp_alphas = np.linspace(0, 0.02, 100)
```

```
kf = KFold(n_splits=5, shuffle=True, random_state=2)

cv_error_rates = []
tree_sizes = []

for ccp_alpha in ccp_alphas:
    tree_clf = DecisionTreeClassifier(random_state=2, ccp_alpha=ccp_alpha)
    scores = cross_val_score(tree_clf, X, y, cv=kf, scoring='accuracy')
    cv_error_rates.append(1 - np.mean(scores))


    tree_clf.fit(X, y)
    tree_sizes.append(tree_clf.get_n_leaves())

plt.figure(figsize=(10, 6))
plt.plot(tree_sizes, cv_error_rates, marker='o', linestyle='-',
 ↪drawstyle='steps-post', color='blue')
plt.title('Tree Size vs. Classification Error Rate')
plt.xlabel('Tree Size (number of leaves)')
plt.ylabel('Cross-Validated Classification Error Rate')
plt.grid(True)
plt.show()

optimal_index = np.argmin(cv_error_rates)
optimal_tree_size = tree_sizes[optimal_index]
optimal_ccp_alpha = ccp_alphas[optimal_index]

print("Optimal Tree Size with the lowest error rate:", optimal_tree_size)
print("Optimal ccp_alpha with the lowest error rate:", optimal_ccp_alpha)
```

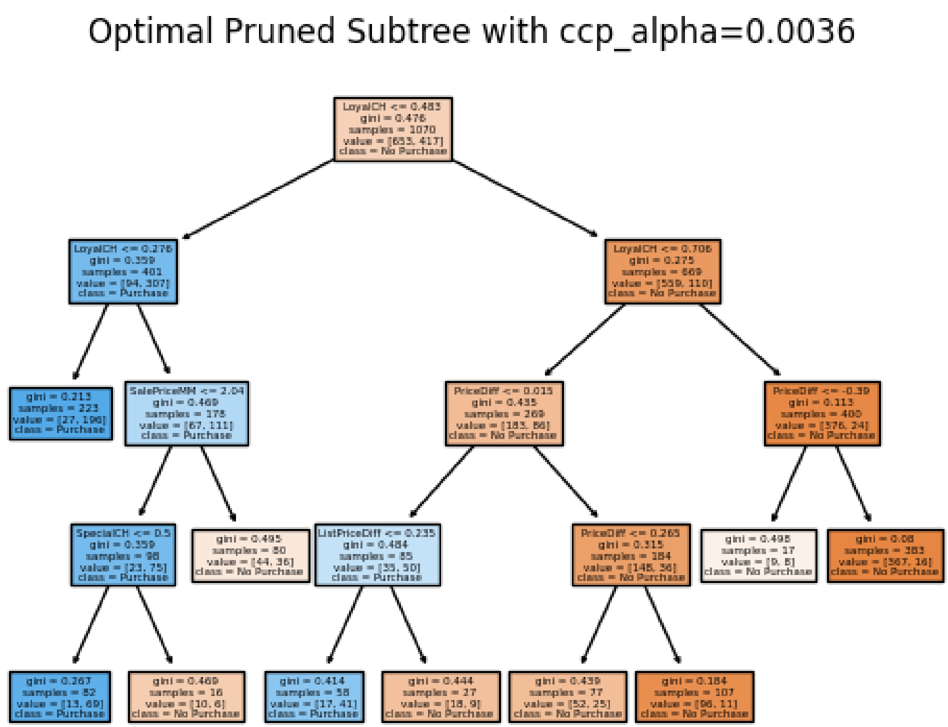



```
Optimal Tree Size with the lowest error rate: 10
Optimal ccp_alpha with the lowest error rate: 0.0036363636363636364
```

[ ]:
```
# The ccp_alpha parameter in decision tree pruning controls the trade-off
↪between tree complexity and model accuracy. A higher value of  leads to more
↪aggressive pruning, resulting in a smaller, simpler tree that may help
↪prevent overfitting but could underfit the data. Conversely, a lower  value
↪allows for a larger, more complex tree that captures detailed data patterns,
↪but risks fitting noise, potentially increasing the classification error on
↪new data.
```

[78]:
```
# Part (g)

ccp_alphas = np.linspace(0, 0.02, 100)

kf = KFold(n_splits=5, shuffle=True, random_state=2)

cv_error_rates = []
tree_sizes = []

for ccp_alpha in ccp_alphas:
    tree_clf = DecisionTreeClassifier(random_state=2, ccp_alpha=ccp_alpha)
    scores = cross_val_score(tree_clf, X, y, cv=kf, scoring='accuracy')
    cv_error_rates.append(1 - np.mean(scores))
```

```
    tree_clf.fit(X, y)
    tree_sizes.append(tree_clf.get_n_leaves())

optimal_index = np.argmin(cv_error_rates)
optimal_ccp_alpha = ccp_alphas[optimal_index]

tree_clf_optimal = DecisionTreeClassifier(random_state=2,␣
 ↪ccp_alpha=optimal_ccp_alpha)
tree_clf_optimal.fit(X, y)

plot_tree(tree_clf_optimal, filled=True, feature_names=X.columns,␣
 ↪class_names=['No Purchase', 'Purchase'])
plt.title(f'Optimal Pruned Subtree with ccp_alpha={optimal_ccp_alpha:.4f}')
plt.show()
```



[79]:
```
# Part (h)

unpruned_tree_clf = DecisionTreeClassifier(random_state=2)
unpruned_tree_clf.fit(X_train, y_train)
unpruned_predictions = unpruned_tree_clf.predict(X_train)
unpruned_error_rate = 1 - accuracy_score(y_train, unpruned_predictions)
```

```python
optimal_ccp_alpha = 0.01  # Assume this value was found via cross-validation
pruned_tree_clf = DecisionTreeClassifier(random_state=2,␣
 ↪ccp_alpha=optimal_ccp_alpha)
pruned_tree_clf.fit(X_train, y_train)
pruned_predictions = pruned_tree_clf.predict(X_train)
pruned_error_rate = 1 - accuracy_score(y_train, pruned_predictions)

print(f"Unpruned Tree Training Error Rate: {unpruned_error_rate}")
print(f"Pruned Tree Training Error Rate: {pruned_error_rate}")
```

```
Unpruned Tree Training Error Rate: 0.006675567423230944
Pruned Tree Training Error Rate: 0.1869158878504673
```

[ ]:
```python
# The unpruned tree's low training error rate of about 0.0067 suggests␣
 ↪overfitting by capturing noise, whereas the pruned tree's higher rate of 0.
 ↪1869 indicates better generalization by avoiding excessive complexity. This␣
 ↪exemplifies the classic machine learning trade-off between bias and␣
 ↪variance, favoring the pruned tree for its robustness in practical␣
 ↪applications.
```

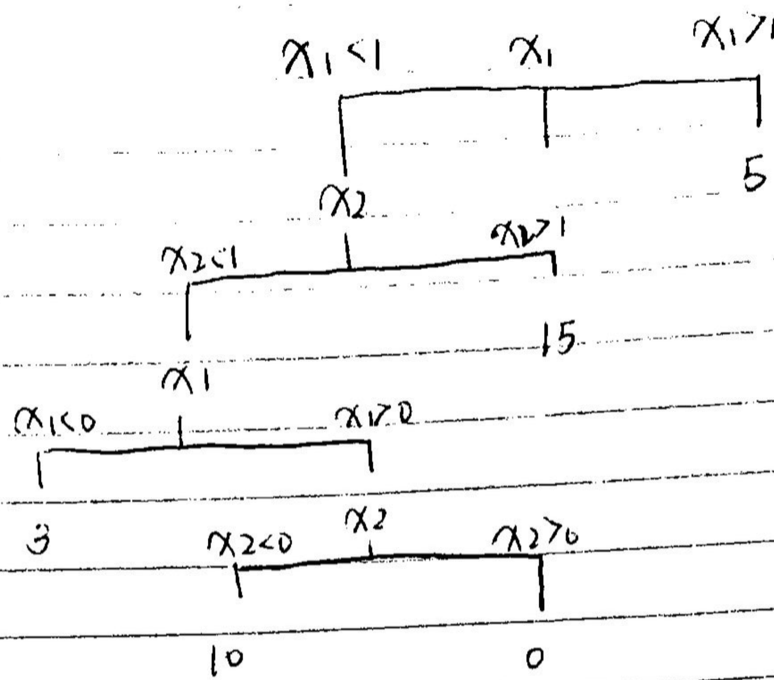
[80]:
```python
# Part (i)

unpruned_test_predictions = unpruned_tree_clf.predict(X_test)
unpruned_test_error_rate = 1 - accuracy_score(y_test, unpruned_test_predictions)

pruned_test_predictions = pruned_tree_clf.predict(X_test)
pruned_test_error_rate = 1 - accuracy_score(y_test, pruned_test_predictions)

print(f"Unpruned Tree Test Error Rate: {unpruned_test_error_rate}")
print(f"Pruned Tree Test Error Rate: {pruned_test_error_rate}")
```

```
Unpruned Tree Test Error Rate: 0.23676012461059193
Pruned Tree Test Error Rate: 0.19937694704049846
```

[ ]:
```python
# The test error rate for the unpruned tree is higher at approximately 0.2368␣
 ↪compared to the pruned tree's error rate of approximately 0.1994. This␣
 ↪indicates that the pruned tree, despite being simpler, generalizes better to␣
 ↪new data than the unpruned tree, which is likely overfitted to the training␣
 ↪data. This exemplifies the benefit of pruning, which reduces model␣
 ↪complexity to enhance performance on unseen data, preventing overfitting.
```

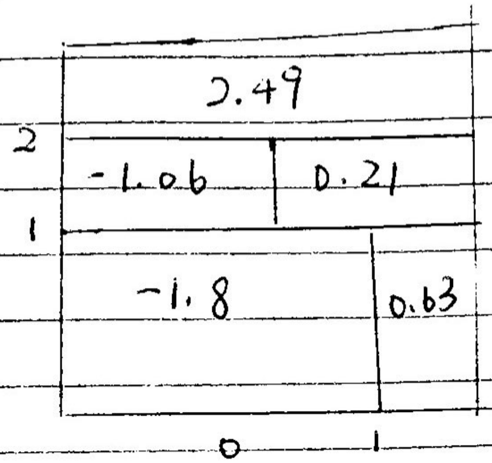
Q2.

(a)



$x_1<1$
x_1
$x_1>1$
5
x_2
$x_2<1$
$x_2>1$
15
x_1
$x_1<0$
$x_1>0$
3
x_2
$x_2<0$
$x_2>0$
10
0

(b)



2.49
2
-1.06
0.21
1
-1.8
0.63
0
1