Machine learning - Assignment 6 - Tree-based approaches

Author: Kemal Cikota

Course: Machine learning

Introduction

In this assignment, i will explore the use of **tree-based** machine learning algorithms like classification trees, regression trees and also explore techniques on how to improve the tree-based approaches like bagging and boosting in order to get better performance metrics.

I used this source when trying to understand what the different variables meant and what they did. This made it easier for me to understand the outputs that i got.

Conceptual Questions

1. explain the differences between bagging and boosting and explain how machine learning methods in general use theese strategies to build robust models using tree models?

bagging (bootstrap aggregation) is a technique that improves the accuracy and stability of machine learning models by reducing variance. It works by bootstraping multiple random subsets of training data and then doing independent training of each subset to train a separate model. All of theese different trees are then combined together in different ways depending on if its a regressive model or a classifier model. A regressive model will use the average of the predictions and a classifier model will use majority voting instead.

For example, random forest is a bagging based model that builds multiple decision trees and averages their predictions in to one prediction.

Boosting is another technique that instead of training independent trees. It will train them sequentially, where each model tries to correct the errors of the previous model. It will begin with a rather shallow decision tree and then fit the model to calculate the error residuals. In the next iteration a new model will be trained that tries to mitigate theese errors. This algorithm is then repeated multiple times untill some limit on number of iterations or some threshold has been met on the errors.

For example, gradient boosting is a boosting algorithm where each tree improves predictions by minimizing the loss function using the gradient descent which is the difference between the input and learning rate.

2. What are the main differences between Random Forest and AdaBoost regarding how the base learners are trained, how samples are weighted, how predictors are made and how robust are they regarding overfitting?

Random forest, as stated earlier, uses bagging where each tree is trained independently and uses very deep trees for high variance models. AdaBoost uses boosting where each tree depends on the previous trees and uses shallow trees.

In random forest, samples are weighted equally where each tree sees a random subset of the data and then in the final model averaging the results for regression. In AdaBoost, misclassified sampler get higher weights because its based on the error, making the next tree focus on them and in the final model the higher weights are assigned to the better trees.

in random forest, predictors are made from averaging and majority voting in order to smooth out predictions. In AdaBoost, more power is given to stronger weak learners.

Random Forest is more robust to overfitting, as averaging prevents extreme predictions while AdaBoost can overfit if the number of weak learners is high.

Practical

For the practical part of the assignment, i have to work with the carseats.csv dataset which contains sales of child car seats at 400 different stores. It shows data like how many units are sold, how much money is invested in advertising, different prices of carseats. And there is also a classifier feature that is rating of how good a shelving location is for a carseat and much more.

Load the data and get an overview of the data

In [ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats as stats
import statsmodels.api as sm
import numpy as np

from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import plot_tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# load carseats.csv
carseats = pd.read_csv('Carseats.csv')
```

```
# Set pandas option to display all columns
pd.set_option('display.max_columns', None)

print(carseats.describe(), end="\n\n")

# I count the categorical separate instances manually as i couldnt find a good f
categorical_columns = carseats.select_dtypes(include=['object', 'category']).col

for col in categorical_columns:
    print(f"Summary for {col}:")
    print(carseats[col].value_counts(), end="\n\n")
```

```
       Unnamed: 0       Sales   CompPrice      Income  Advertising  \
count  400.000000  400.000000  400.000000  400.000000   400.000000   
mean   200.500000    7.496325  124.975000   68.657500     6.635000   
std    115.614301    2.824115   15.334512   27.986037     6.650364   
min      1.000000    0.000000   77.000000   21.000000     0.000000   
25%    100.750000    5.390000  115.000000   42.750000     0.000000   
50%    200.500000    7.490000  125.000000   69.000000     5.000000   
75%    300.250000    9.320000  135.000000   91.000000    12.000000   
max    400.000000   16.270000  175.000000  120.000000    29.000000   

       Population       Price         Age   Education  
count  400.000000  400.000000  400.000000  400.000000  
mean   264.840000  115.795000   53.322500   13.900000  
std    147.376436   23.676664   16.200297    2.620528  
min     10.000000   24.000000   25.000000   10.000000  
25%    139.000000  100.000000   39.750000   12.000000  
50%    272.000000  117.000000   54.500000   14.000000  
75%    398.500000  131.000000   66.000000   16.000000  
max    509.000000  191.000000   80.000000   18.000000  

Summary for ShelveLoc:
ShelveLoc
Medium    219
Bad        96
Good       85
Name: count, dtype: int64

Summary for Urban:
Urban
Yes    282
No     118
Name: count, dtype: int64

Summary for US:
US
Yes    258
No     142
Name: count, dtype: int64
```

In [390…

```
print("total amount of datapoints: ", carseats.shape[0], end="\n\n")
```

```
total amount of datapoints:  400
```

In [391…

```
carseats
```

Out[391…

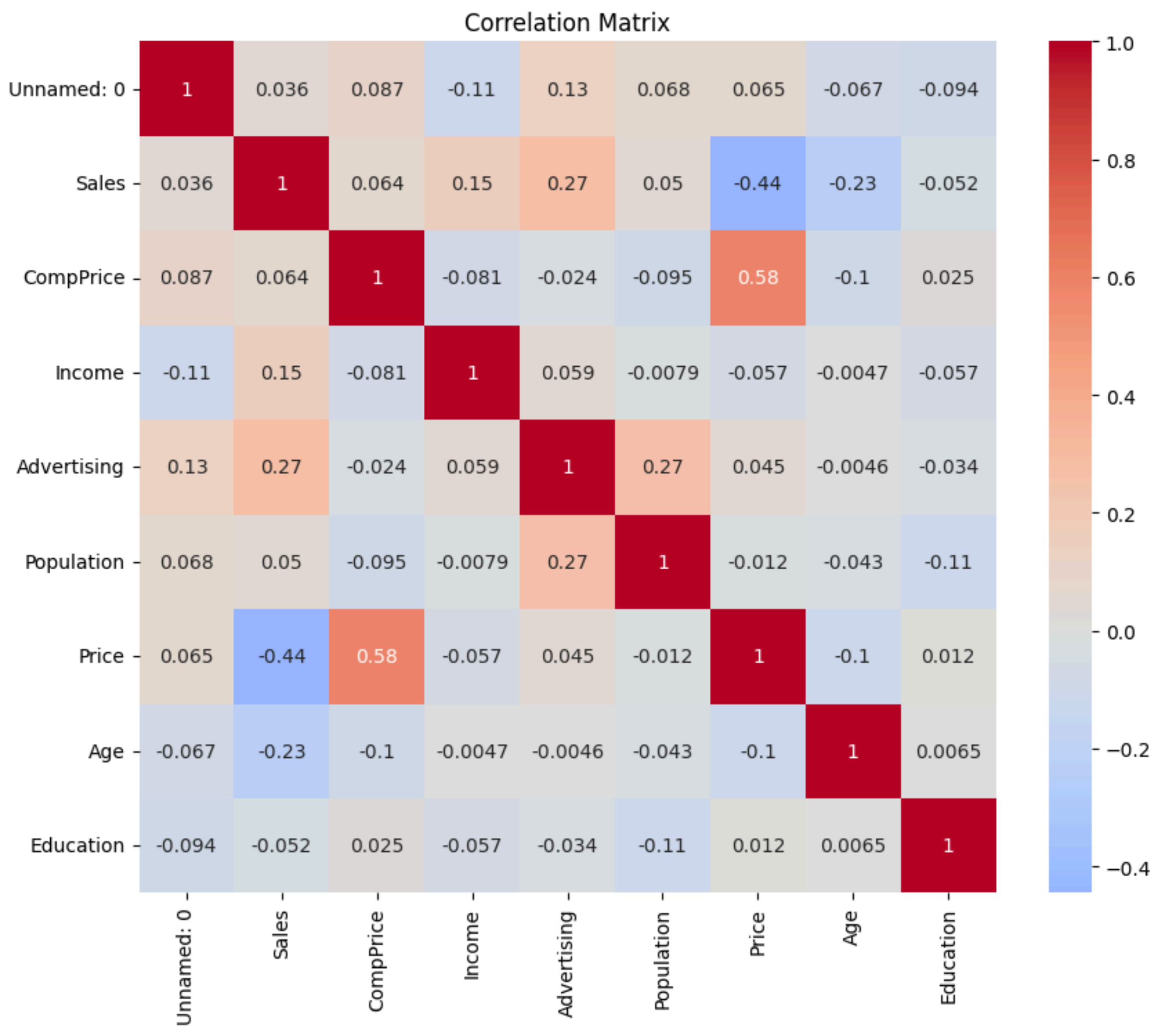
	Unnamed: 0	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc
0	1	9.50	138	73	11	276	120	Bad
1	2	11.22	111	48	16	260	83	Good
2	3	10.06	113	35	10	269	80	Medium
3	4	7.40	117	100	4	466	97	Medium
4	5	4.15	141	64	3	340	128	Bad
...	...	...	...	...	...	...	...	...
395	396	12.57	138	108	17	203	128	Good
396	397	6.14	139	23	3	37	120	Medium
397	398	7.41	162	26	12	368	159	Medium
398	399	5.94	100	79	7	284	95	Bad
399	400	9.71	134	37	0	27	120	Good

400 rows × 12 columns

I now plotted the pairwise correlation between the quantitative variables. This means that i had to drop the classifier features as was shown in the example.

In [392…

```python
# drop urban, US, shelveloc column
carseats = carseats.drop(columns=['Urban', 'US', 'ShelveLoc'])

correlation_matrix = carseats.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title("Correlation Matrix")
plt.show()
```

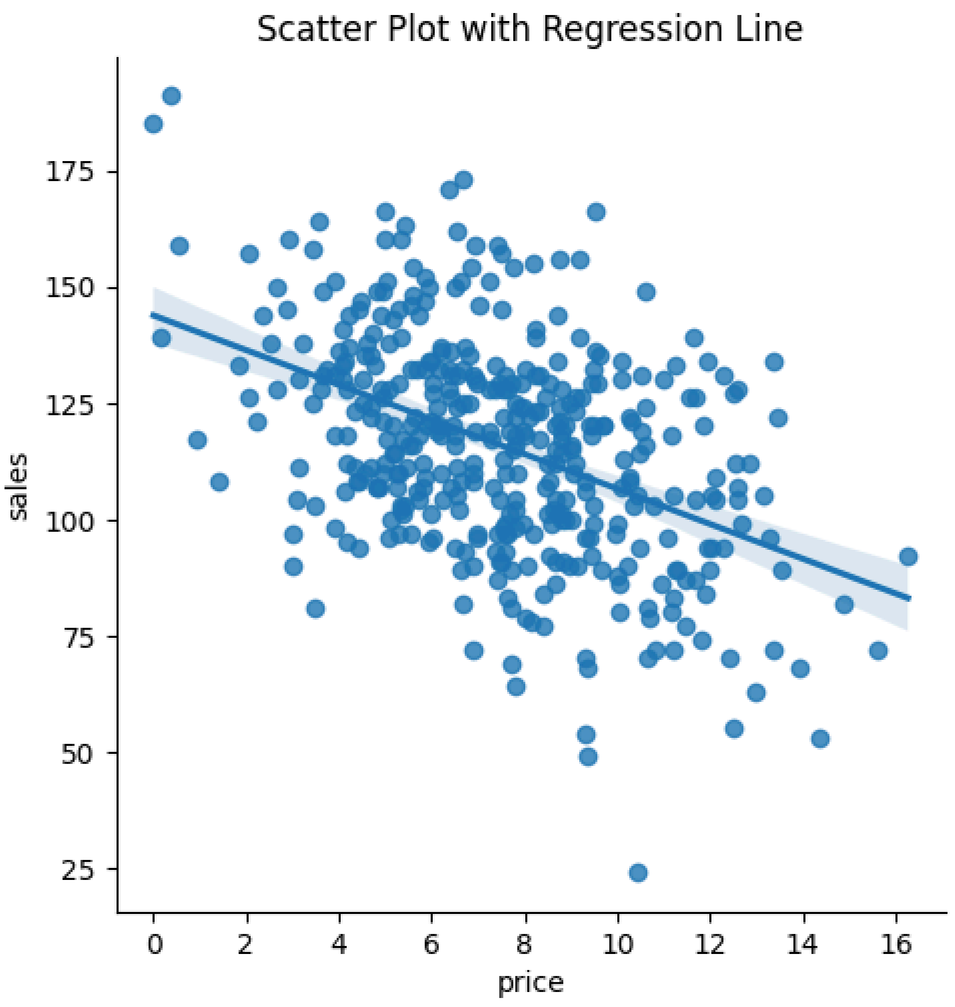


	Unnamed: 0	Sales	CompPrice	Income	Advertising	Population	Price	Age	Education
Unnamed: 0	1	0.036	0.087	-0.11	0.13	0.068	0.065	-0.067	-0.094
Sales	0.036	1	0.064	0.15	0.27	0.05	-0.44	-0.23	-0.052
CompPrice	0.087	0.064	1	-0.081	-0.024	-0.095	0.58	-0.1	0.025
Income	-0.11	0.15	-0.081	1	0.059	-0.0079	-0.057	-0.0047	-0.057
Advertising	0.13	0.27	-0.024	0.059	1	0.27	0.045	-0.0046	-0.034
Population	0.068	0.05	-0.095	-0.0079	0.27	1	-0.012	-0.043	-0.11
Price	0.065	-0.44	0.58	-0.057	0.045	-0.012	1	-0.1	0.012
Age	-0.067	-0.23	-0.1	-0.0047	-0.0046	-0.043	-0.1	1	0.0065
Education	-0.094	-0.052	0.025	-0.057	-0.034	-0.11	0.012	0.0065	1

In [393…

```
# Step 1 (plots)

corCoef, pValue = stats.pearsonr(carseats['Sales'], carseats['Price'])

print(f"Pearson Correlation Coefficient: {corCoef:.2f}, P-value: {pValue:.2e}")

sns.lmplot(data=carseats, x='Sales', y='Price', ci=95)

# Add labels to the plot
plt.xlabel("price")
plt.ylabel("sales")
plt.title(f"Scatter Plot with Regression Line")
plt.show() # Remember to make the window bigger to see the plot
```

```
Pearson Correlation Coefficient: -0.44, P-value: 7.62e-21
```



So i plotted a correlation plot and found that two pairs of variables had interesting correlations. The first one is Price-Compprice (0.58) and the other one is Price-Sales (-0.48). I chose the two variables Price and Sales and then wanted to print those two variables against each other on a scatter plot. In this scatter plot we also have a straight line. This is a linear regression line that shows the relationship between Sales and Price and as we can see when prices increase, sales decrease. And vice versa. This confirms that there is a negative relationship between sales and price from our scatterplot.

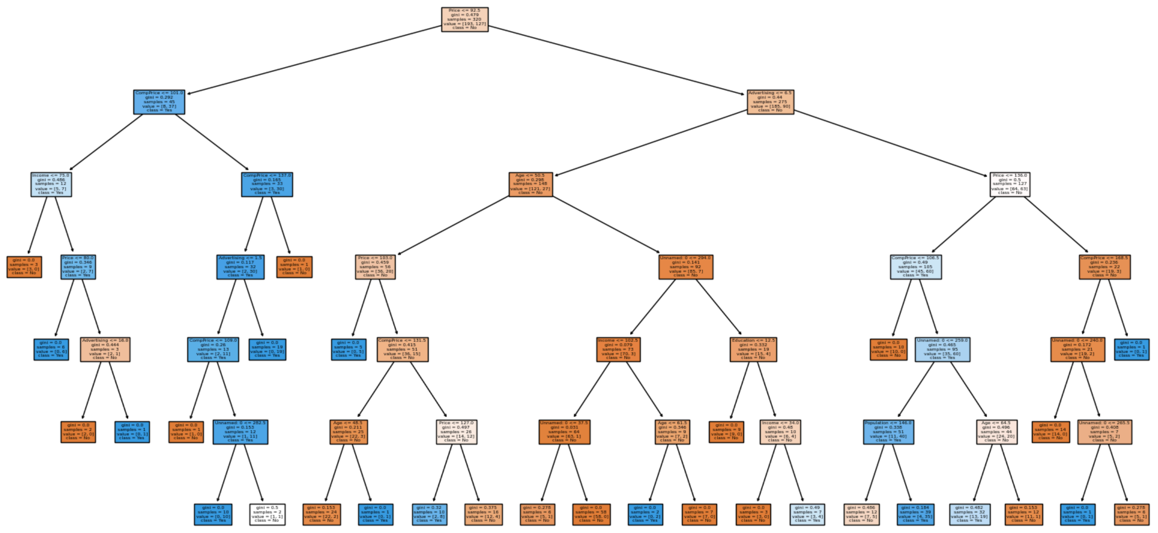
Learn ans assess classification trees

In [394…

```
# Create binary classification target variable 'High'
carseats['High'] = np.where(carseats['Sales'] <= 8, "No", "Yes")

# Drop 'Sales' since it should not be a predictor
carseats = carseats.drop(columns=['Sales'])

# Separate features (X) and target variable (y)
X = carseats.drop(columns=['High'])
y = carseats['High'].map({'No': 0, 'Yes': 1})
X = pd.get_dummies(X, drop_first=False)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

tree_model = DecisionTreeClassifier(random_state=42, max_depth=6) # set max_dept
```

```
tree_model.fit(X_train, y_train)
y_pred = tree_model.predict(X_test)

# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("model accuracy: ", accuracy)

# number of terminal nodes
print("number of nodes: ", tree_model.get_n_leaves())

# residual mean deviance
print("residual mean deviance: ", tree_model.score(X_test, y_test))
print(f"Residual mean deviance: {tree_model.score(X_test, y_test)}")

plt.figure(figsize=(20,10))
plot_tree(tree_model, filled=True, feature_names=X.columns.tolist(), class_names
plt.show()
```

```
model accuracy:  0.6125
number of nodes:  30
residual mean deviance:  0.6125
Residual mean deviance: 0.6125
```



For this first simple model i want to predict whether or not sales exceed 8 units or not so i set the target variable to either "high, yes" or "high, no" depending of whether or not i get more than 8 sales or not. I then converted all categorical variables to dummy variables as well.

Because sklearn by default prints very large trees i had to hardocde a "max_depth=6" argument when generating the tree. even in PDF-form, this tree is very hard to read so i recommend opening this in VScode and then pressing the "expand image" button thats on the upper right corner of the printed image because then the tree will be rendered properly so we can zoom in and move around in the image in order to see the contents of each node.

another thing about sklearn that is important to notice is that noticably, all of the trees generated in this assignments will be different then the output that was given in the assignment/example sheet. According to my research, sklearn uses **Gini impurity** when generating the decision trees and the "tree" library that is used in the examples probably has some other technique. There are also a lot of other factors that can cause this difference like how missing values are handled, how dummy variables are generated and how the RNG generates random values. I spent a lot of time trying to get the exact same tree but didnt manage to so instead of focusing on that too much, i will instead focus more on giving good interpretations and showing that improvements techniques gives better performance metrics and so on.

However, now that this has been cleared up i can explain the trees and metrics.

The model acheived a accuracy of 61.25% which is a measure of its ability to classify sales. The residual mean deviance score is a measurement of how well the model fits to the test data.

The root node splits the data based on the **price** feature which is a key determinantn in the sales classification. Other key variables appearing include **CompPrice**, **age** and **advertizing**, which means theese factors play significant roles in predicting the outcome of sales.

In [395…

```
# Create a confusion matrix as a crosstab
confusion_tab = pd.crosstab(index=y_pred, columns=y_test, rownames=['Predicted']

print("Confusion Table (pred, actual)):")
print(confusion_tab)

print("\n\n")

# F-score
precision = confusion_tab.iloc[1, 1] / (confusion_tab.iloc[1, 1] + confusion_tab
recall = confusion_tab.iloc[1, 1] / (confusion_tab.iloc[1, 1] + confusion_tab.il
f_score = 2 * (precision * recall) / (precision + recall)
print("precision: ", precision)
print("recall: ", recall)
print("f-score: ", f_score)
```

```
Confusion Table (pred, actual)):
Actual      0   1
Predicted
0          31  19
1          12  18



precision:  0.4864864864864865
recall:  0.6
f-score:  0.5373134328358209
```

Like we have done in previous assignments i also plotted a confusion matrix which gives us a feeling for how well the model predicts when taking false positives and false negatives in to account.

So as wel can see the first cell in the matrix is (0, 0) which can me read as (actually no, predicted no) which means "true negatives". the other cells can be read in a similar way.

From this, it is perfect to calculate the precision, recall and f-score that are all derived from this confusion matrix.

precision is a measure of positive predictive values:

$$Precision = \frac{TP}{TP + FP}$$

recall is a measure of how many actual cases were correctly predicted:

$$Recall = \frac{TP}{TP + FN}$$

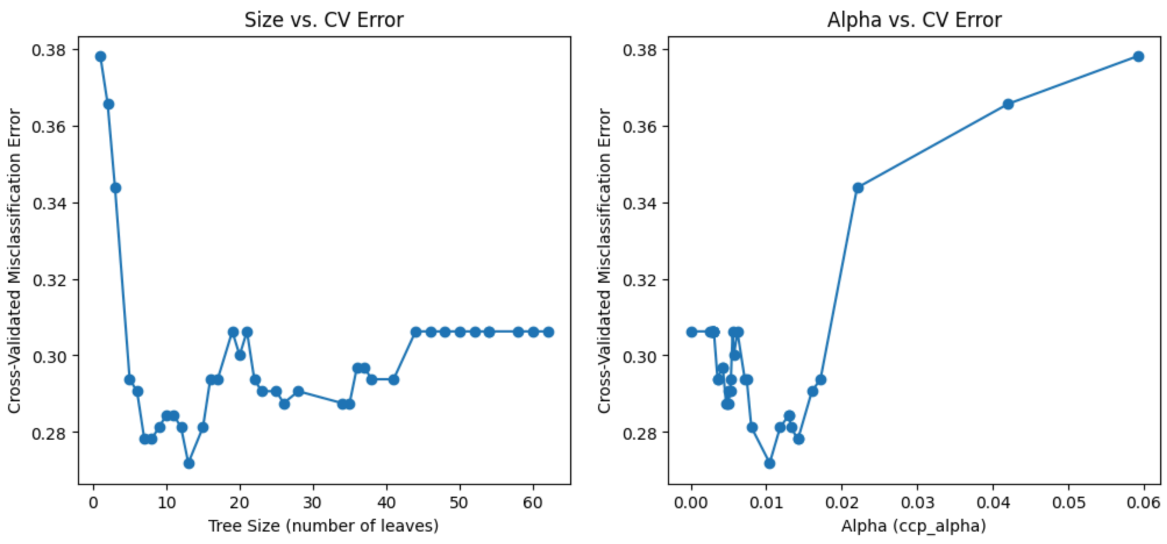
F1 score is a balance between precision and recall

$$F1 = 2 * \frac{Precision * recall}{precision + recall}$$

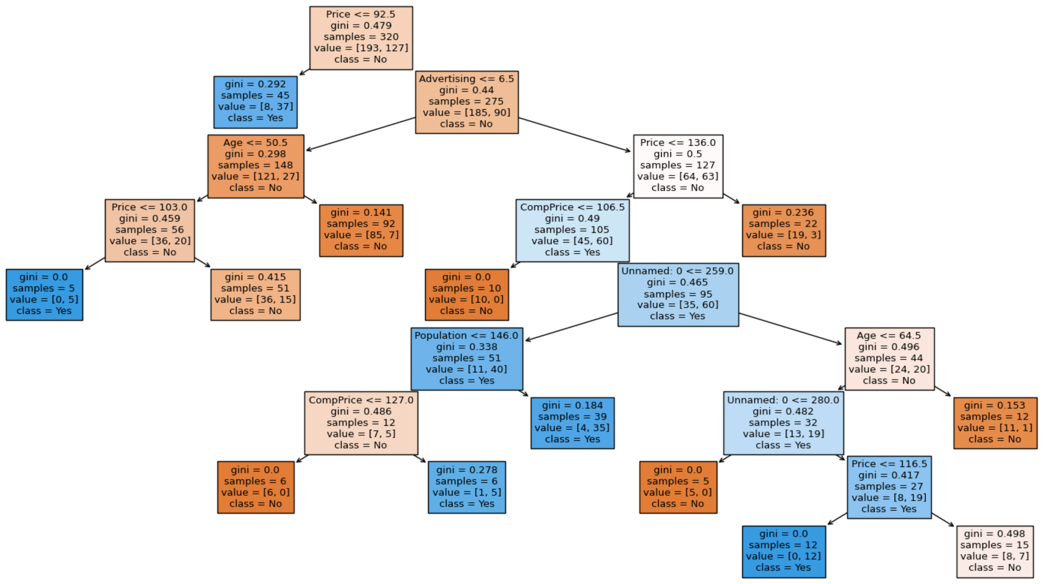
In [396…

```
tree_model_2 = DecisionTreeClassifier(random_state=42).cost_complexity_pruning_p
ccp_alphas, impurities = tree_model_2.ccp_alphas, tree_model_2.impurities

# Arrays to store results
alpha_values = []
mean_errors = []
std_errors = []
tree_sizes = []


for alpha in ccp_alphas:
    temp_tree = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)

    # Misclassification error is (1 - accuracy)
    errors = 1 - cross_val_score(temp_tree, X_train, y_train, cv=10, scoring='ac

    # Fit on the entire training set to measure the actual size (leaves)
    temp_tree.fit(X_train, y_train)
    size = temp_tree.get_n_leaves()

    alpha_values.append(alpha)
    mean_errors.append(errors.mean())
    std_errors.append(errors.std())
    tree_sizes.append(size)


fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# -- first plot
axes[0].plot(tree_sizes, mean_errors, marker='o')

axes[0].set_xlabel("Tree Size (number of leaves)")
axes[0].set_ylabel("Cross-Validated Misclassification Error")
axes[0].set_title("Size vs. CV Error")

# -- second plot
axes[1].plot(alpha_values, mean_errors, marker='o')

axes[1].set_xlabel("Alpha (ccp_alpha)")
axes[1].set_ylabel("Cross-Validated Misclassification Error")
```

```
axes[1].set_title("Alpha vs. CV Error")

plt.show()

best_idx = np.argmin(mean_errors)
best_alpha = alpha_values[best_idx]
print("Best alpha:", best_alpha)
print("CV error at best alpha:", mean_errors[best_idx])

pruned_tree = DecisionTreeClassifier(random_state=42, ccp_alpha=best_alpha)
pruned_tree.fit(X_train, y_train)

plt.figure(figsize=(20,10))
plot_tree(pruned_tree, filled=True, feature_names=X_train.columns.tolist(), clas
plt.show()
```



```
Best alpha: 0.010463235294117652
CV error at best alpha: 0.271875
```



In [397…

```
# Create a confusion matrix as a crosstab
y_pred = pruned_tree.predict(X_test)
confusion_tab = pd.crosstab(index=y_pred, columns=y_test, rownames=['Predicted']
print("Confusion Table (pred, actual)):")
print(confusion_tab)

# print accuracy
```

```python
y_pred = pruned_tree.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("accuracy: ", accuracy)

# F-score
precision = confusion_tab.iloc[1, 1] / (confusion_tab.iloc[1, 1] + confusion_tab
recall = confusion_tab.iloc[1, 1] / (confusion_tab.iloc[1, 1] + confusion_tab.il
f_score = 2 * (precision * recall) / (precision + recall)
print("precision: ", precision)
print("recall: ", recall)
print("f-score: ", f_score)
```

```
Confusion Table (pred, actual)):
Actual      0   1
Predicted
0          32  19
1          11  18
accuracy:  0.625
precision:  0.4864864864864865
recall:  0.6206896551724138
f-score:  0.5454545454545455
```

In this step i applied pruning in order to try to get a better tree that is going to be better at predicting our responses by essentially "cutting off" branches in order to reduce overfitting. This technique uses an "alpha parameter" which essentially is a parameter that decides how much of the tree is going to be cut off from the tree that was trained. in order to decrease tree complexity.

i applied cost complexity pruning (CCP) using cross validation to select an optimal alpha for pruning. My strategy was that i first computed the alphas and then used k-fold with 10 folds to evaluate which pruning levels were the best. In the end, the alphas that were chose was 0.01046 that minimized the classification error to 0.27 as we can see from the first two graphs

For this i also managed to create a confusion matrix that better describe the predictive power of the model. for this i will compile the difference between the unpruned and pruned tree in a table so that we can better see if the pruned tree was better or not.

metric	unpruned	pruned
accuracy	61,25%	62,25%
precision	60%	48,65%
recall	48,6%	62,1%
F1	53,6%	54,5%

The pruned tree had slightly better accuracy which indicates better generalization. The recall had a pretty large increase meaning that the model correctly caputerd more "yes" cases but had a smaller precision which means that more false positives occoured. Conclusively, the pruned tree did have a better F1-score then the non-pruned tree so we can say that the model overall was better.

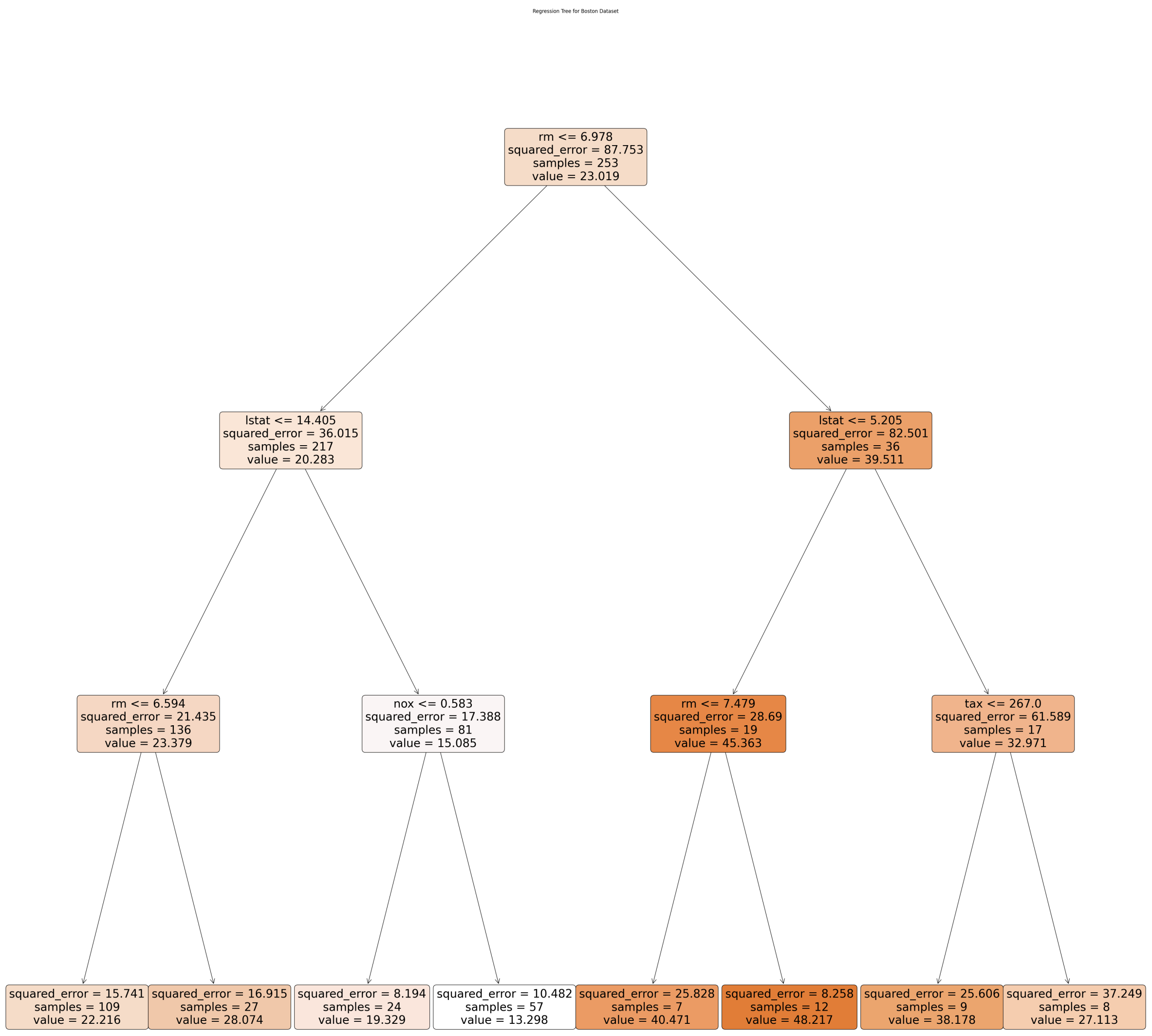
Learn and assess regression trees

Now we want to use regression tree-based models with the Boston dataset that only includes numerical data which is perfect for regression based solutions.

```
In [398…
# Load boston.csv
boston = pd.read_csv('Boston.csv')

# Set pandas option to display all columns
pd.set_option('display.max_columns', None)

np.random.seed(1)  # for reproducibility

# Suppose 'medv' is the target column
X = boston.drop(columns=['medv'])
y = boston['medv']

train_indices = np.random.choice(len(X), size=len(X)//2, replace=False) # 50/50
test_indices = np.setdiff1d(np.arange(len(X)), train_indices)

X_train = X.iloc[train_indices]
y_train = y.iloc[train_indices]
X_test = X.iloc[test_indices]
y_test = y.iloc[test_indices]

tree_boston = DecisionTreeRegressor(random_state=1, max_depth=3, min_samples_lea
tree_boston.fit(X_train, y_train)

# -------------------------------------------------------
# Sooo... sklearn doesnt have a built in summary function so i had to manually c

n_leaves = tree_boston.get_n_leaves()

# Residual mean deviance is basically the training MSE
train_preds = tree_boston.predict(X_train)
train_mse = mean_squared_error(y_train, train_preds)

# We can also examine the distribution of residuals on the training set
residuals = y_train - train_preds
res_min = residuals.min()
res_q1 = np.quantile(residuals, 0.25)
res_median = np.median(residuals)
res_q3 = np.quantile(residuals, 0.75)
res_max = residuals.max()

print("Regression tree for Boston dataset (Python equivalent of R's tree()):")
print(f"Number of terminal nodes: {n_leaves}")
print(f"Residual mean deviance (training MSE): {train_mse:.3f}")
print("Distribution of residuals (training set):")
print(f"   Min.    : {res_min:.3f}")
print(f"   1Q      : {res_q1:.3f}")
print(f"   Median : {res_median:.3f}")
print(f"   3Q      : {res_q3:.3f}")
print(f"   Max.    : {res_max:.3f}")
# -------------------------------------------------------


# make predictions on the test set
test_preds = tree_boston.predict(X_test)
test_mse = mean_squared_error(y_test, test_preds)
print(f"\nTest MSE: {test_mse:.3f}")
```

```python
# plot the tree
plt.figure(figsize=(50, 50))
plot_tree(
    tree_boston,
    feature_names=X.columns.tolist(),
    filled=True,
    rounded=True
)
plt.title("Regression Tree for Boston Dataset")
plt.show()
```

```
Regression tree for Boston dataset (Python equivalent of R's tree()):
Number of terminal nodes: 8
Residual mean deviance (training MSE): 14.921
Distribution of residuals (training set):
   Min.   : -12.113
   1Q     : -2.416
   Median : 0.002
   3Q     : 1.984
   Max.   : 27.784

Test MSE: 26.008
```



For this part of the assignment, i implemented a regression tree to predict median house values (**MEDV**) in the boston dataset based on numerical predictor variales. I used a 50/50 split of training / testing data in order to try and keep a similar approach to the example. I also limited the size of the tree in order to make it easier to read for this demonstration but if one would want to have the full size tree just remove the **max_depth=** argument when creating the model.

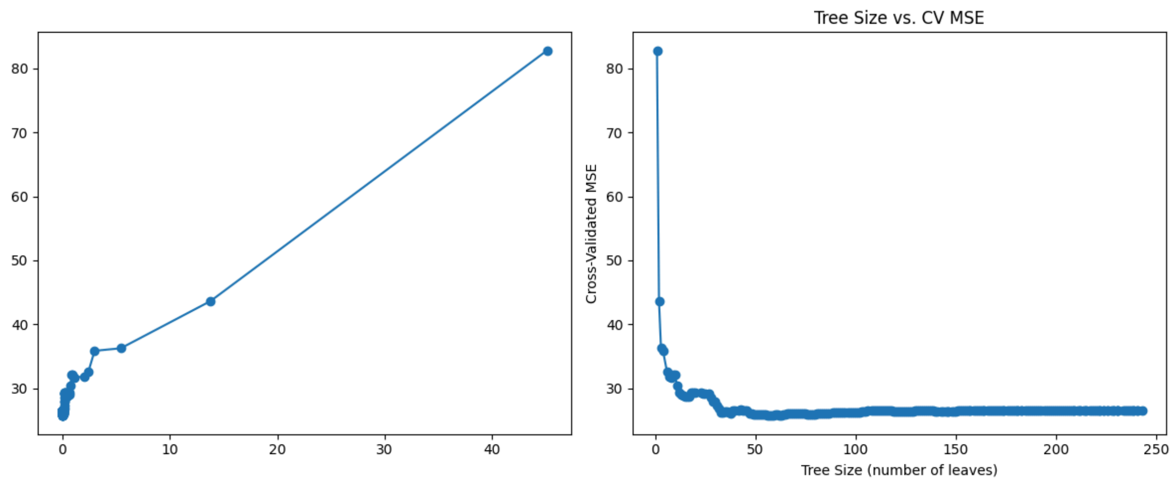
The tree plot shows that the top splits involve **RM (room per house)**, **LSTAT (percentage of lower-status population)** and **NOX (nitrogen oxide concentration)**. We can also recall that from previous assignments and from just using real-world expectations that more rooms and lower crime generally indicate higher house prices.

We can also see from the residuals that i calculated that the residual mean deviance (training MSE) is 14.92 which is a measurement of how well the model fit the training data. However we can see that the min-max residuals (-12.113, 27.784) show that the model sometimes can make large errors. We can also see that the *testing MSE* is 26.008 which is much higher than the training MSE which suggest that the model overfits as it performs better on the training set then the test set.

In [399…

```
tree_model_3 = DecisionTreeRegressor(random_state=1).cost_complexity_pruning_pat
ccp_alphas, impurities = tree_model_3.ccp_alphas, tree_model_3.impurities

# Arrays to store results
alpha_values = []
mean_cv_errors = []
std_cv_errors = []
tree_sizes = []

# -------------------------------------------------------
# For each alpha, train and cross-validate a DecisionTreeRegressor
# -------------------------------------------------------
for alpha in ccp_alphas:
    temp_tree = DecisionTreeRegressor(random_state=1, ccp_alpha=alpha)

    # 10-fold CV for MSE (scoring='neg_mean_squared_error' -> need to negate)
    cv_scores = cross_val_score(temp_tree, X_train, y_train, scoring='neg_mean_s
    # Convert from negative MSE to MSE
    mse_scores = -cv_scores

    # Fit on entire training set once to measure actual tree size
    temp_tree.fit(X_train, y_train)
    n_leaves = temp_tree.get_n_leaves()

    alpha_values.append(alpha)
    mean_cv_errors.append(mse_scores.mean())
    std_cv_errors.append(mse_scores.std())
    tree_sizes.append(n_leaves)

# Plot (1) Alpha vs. CV Error, (2) Tree Size vs. CV Error
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# -- first plot
axes[0].plot(alpha_values, mean_cv_errors, marker='o')

# -- second plot
```

```python
axes[1].plot(tree_sizes, mean_cv_errors, marker='o')

axes[1].set_xlabel('Tree Size (number of leaves)')
axes[1].set_ylabel('Cross-Validated MSE')
axes[1].set_title('Tree Size vs. CV MSE')

plt.tight_layout()
plt.show()


# Identify best alpha (lowest MSE) and refit a final pruned tree
best_idx = np.argmin(mean_cv_errors)
best_alpha = alpha_values[best_idx]
print("Best alpha:", best_alpha)
print("Cross-validated MSE at best alpha:", mean_cv_errors[best_idx])

pruned_tree = DecisionTreeRegressor(random_state=1, ccp_alpha=best_alpha)
pruned_tree.fit(X_train, y_train)

# Evaluate on the test set
test_preds_pruned = pruned_tree.predict(X_test)
test_mse_pruned = mean_squared_error(y_test, test_preds_pruned)
print("\nTest MSE (pruned tree):", test_mse_pruned)
```



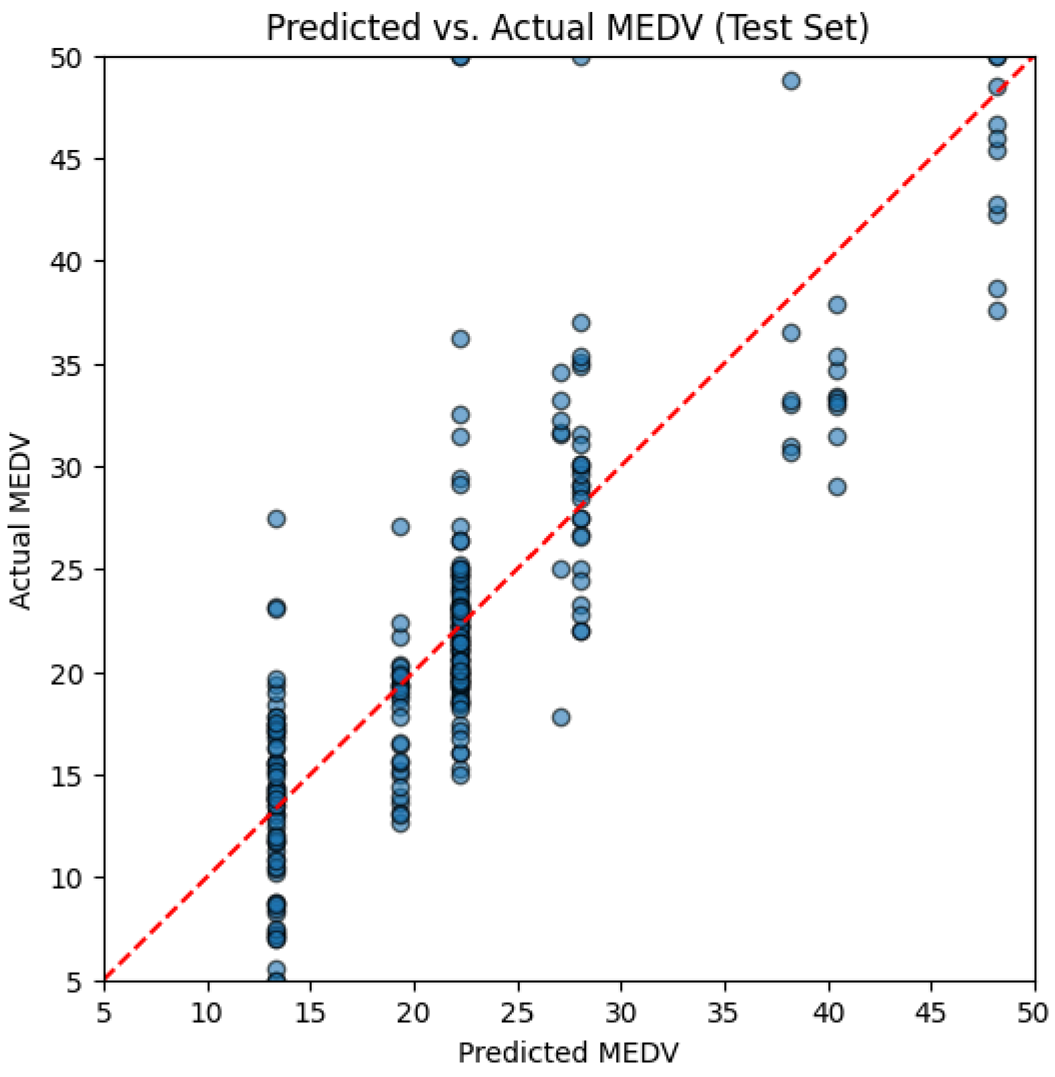
```
Best alpha: 0.040349143610008838
Cross-validated MSE at best alpha: 25.70247791867495

Test MSE (pruned tree): 18.24121103436141
```

In order to try and improve the regression tree model, i applied the same pruning strategy as i did with the classification tree. I used k-fold 10. The goal was to find an optimal alpha again that minimizes the MSE while preventing overfitting.

The two plots show that when alpha increases, the model does become simpler which makes sence because more of the tree gets cut off. but at a certain point, the MSE will increase due to underfitting. The best alpha was found at 0.0403.

In [400…

```python
# 1. Generate predictions on the test set
yhat = tree_boston.predict(X_test)

# 2. Plot predicted vs. actual
plt.figure(figsize=(6, 6))
plt.scatter(yhat, y_test, alpha=0.6, edgecolors='k')
plt.xlabel("Predicted MEDV")
plt.ylabel("Actual MEDV")
plt.title("Predicted vs. Actual MEDV (Test Set)")

# 3. Plot the line y = x for reference
lims = [min(yhat.min(), y_test.min()), max(yhat.max(), y_test.max())]
plt.plot(lims, lims, 'r--')  # diagonal line
plt.xlim(lims)
plt.ylim(lims)

plt.show()

# 4. Compute and display MSE
mse = np.mean((yhat - y_test)**2)
print("Test MSE:", mse)
```



```
Test MSE: 26.00834875580148
```

This compares the predicted vs. actual medv values and the red dotted line represents the ideal scenario where predictions meet the actual values perfectly. Since we can see that the dots are quite scattered, we cans ee that there is some variance which means that the predictions arent perfect and we have some clustering at the lower prices which suggests that there could be some bias in lower-valued homes.

i can also directly compare the CV MSE and MSE between the pruned and unpruned tree by creating a nice markdown table as before:

metric	unpruned	pruned
cross validated MSE	X	26,01
test MSE	25,70	18,24

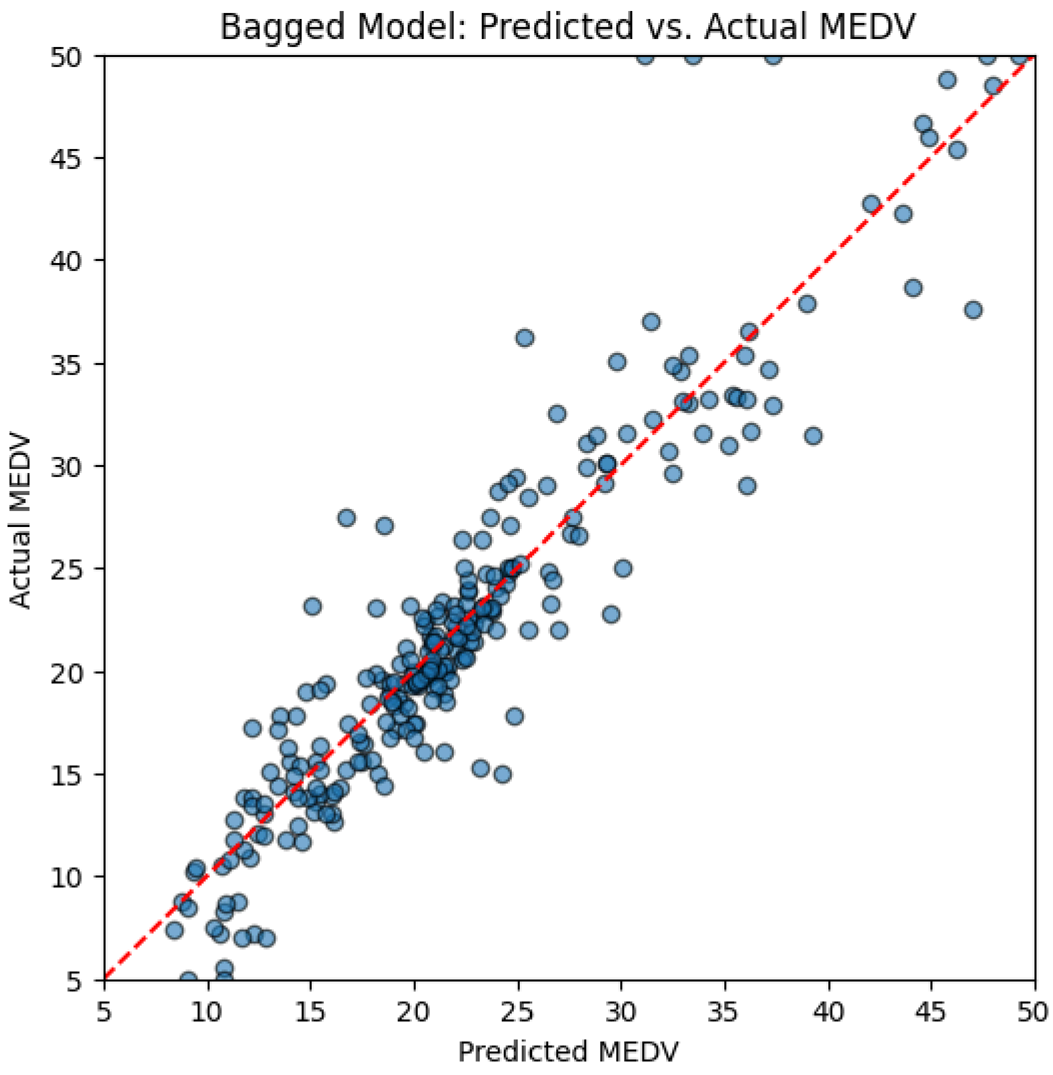
As we can see, the pruning reduced the test MSE from 26.01 to 18.24 which is quite a big leap which means that the pruned tree model does have better predictive power.

Learn and assess regression bagging trees and random forest

In [401…

```
X = boston.drop(columns=['medv'])
y = boston['medv']

bag_boston = RandomForestRegressor(
    n_estimators=500,
    max_features=X_train.shape[1],  # same as settingmtry = 13 which ises all fe
    bootstrap=True,
    oob_score=True,
    random_state=1
)
bag_boston.fit(X_train, y_train)
yhat_bag = bag_boston.predict(X_test)

plt.figure(figsize=(6, 6))
plt.scatter(yhat_bag, y_test, alpha=0.6, edgecolors='k')
plt.xlabel("Predicted MEDV")
plt.ylabel("Actual MEDV")
plt.title("Bagged Model: Predicted vs. Actual MEDV")

# Draw a 45-degree line for reference
lims = [min(yhat_bag.min(), y_test.min()), max(yhat_bag.max(), y_test.max())]
plt.plot(lims, lims, 'r--')
plt.xlim(lims)
plt.ylim(lims)
plt.show()


mse_bag = mean_squared_error(y_test, yhat_bag)
print("Test MSE (Bagged Model):", mse_bag)


# View feature importances
importances = bag_boston.feature_importances_
features = X.columns
feat_importance_df = pd.DataFrame({"Feature": features, "Importance": importance
feat_importance_df = feat_importance_df.sort_values(by="Importance", ascending=F

print("\nFeature Importances:")
print(feat_importance_df)

oob_preds = bag_boston.oob_prediction_

# OOB MSE (analogous to "Mean of squared residuals" in R)
oob_mse = mean_squared_error(y_train, oob_preds)

# OOB R^2 (fraction of variance explained), scikit-learn calculates this automat
```

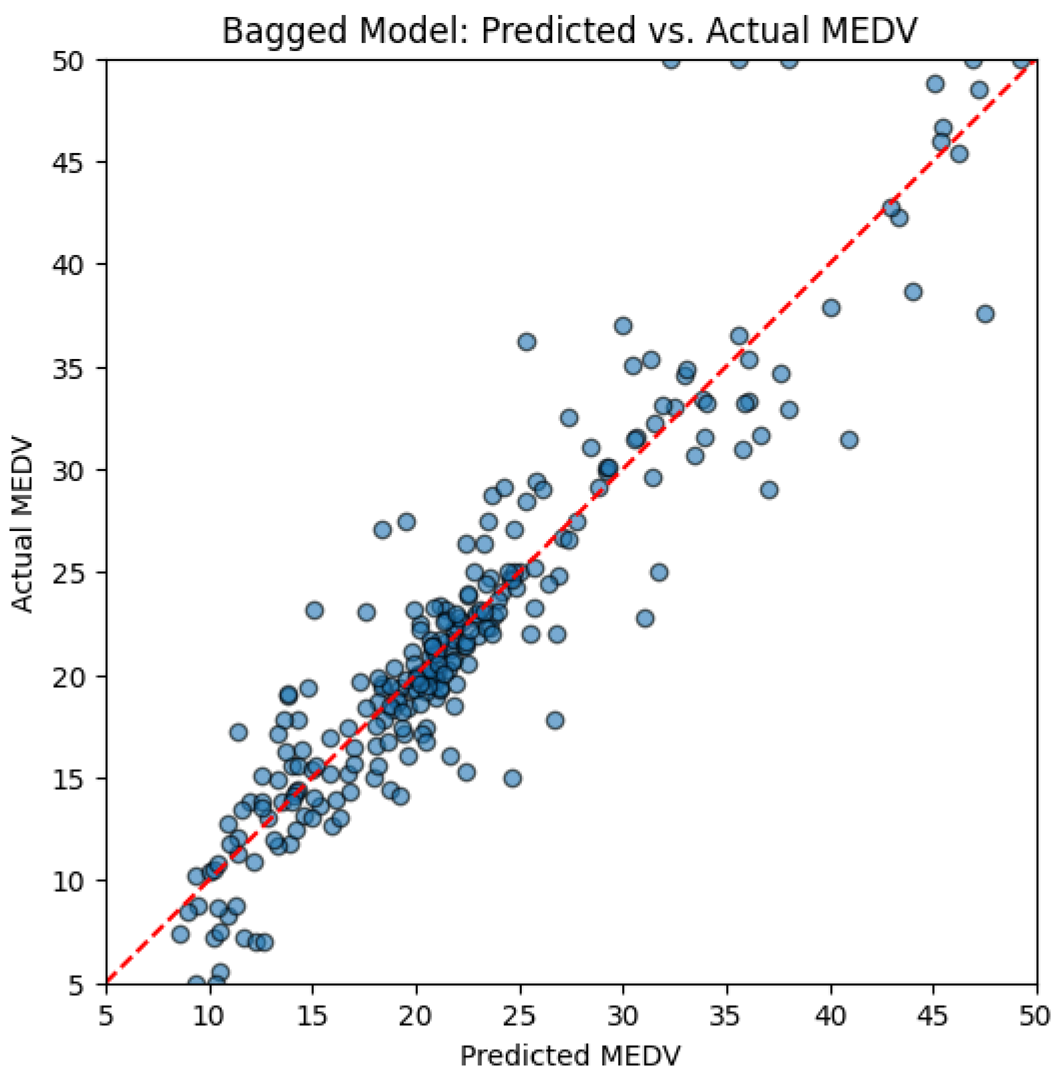
```
oob_r2 = bag_boston.oob_score_

# Convert R^2 to % Var explained
oob_var_explained = 100 * oob_r2

print(f"Mean of squared residuals (OOB): {oob_mse:.5f}")
print(f"% Var explained (OOB): {oob_var_explained:.2f}")
```



```
Test MSE (Bagged Model): 11.071022049486121

Feature Importances:
       Feature  Importance
6           rm    0.572738
13       lstat    0.251502
8          dis    0.040805
1         crim    0.026792
5          nox    0.025290
0   Unnamed: 0    0.017843
7          age    0.014766
12       black    0.013513
11     ptratio    0.013462
10         tax    0.011277
3        indus    0.004303
9          rad    0.003788
4         chas    0.002235
2           zn    0.001687
Mean of squared residuals (OOB): 16.58060
% Var explained (OOB): 81.11
```

In [402…

```python
X = boston.drop(columns=['medv'])
y = boston['medv']

bag_boston = RandomForestRegressor(
    n_estimators=25,
    max_features=X_train.shape[1],  # same as settingmtry = 13 which ises all fe
    bootstrap=True,
    oob_score=True,
    random_state=1
)
bag_boston.fit(X_train, y_train)
yhat_bag = bag_boston.predict(X_test)

# 4. Plot predicted vs. actual
plt.figure(figsize=(6, 6))
plt.scatter(yhat_bag, y_test, alpha=0.6, edgecolors='k')
plt.xlabel("Predicted MEDV")
plt.ylabel("Actual MEDV")
plt.title("Bagged Model: Predicted vs. Actual MEDV")

# Draw a 45-degree line for reference
lims = [min(yhat_bag.min(), y_test.min()), max(yhat_bag.max(), y_test.max())]
plt.plot(lims, lims, 'r--')
plt.xlim(lims)
plt.ylim(lims)
plt.show()


# 5. Compute test MSE
mse_bag = mean_squared_error(y_test, yhat_bag)
print("Test MSE (Bagged Model):", mse_bag)


# View feature importances

importances = bag_boston.feature_importances_
features = X.columns
feat_importance_df = pd.DataFrame({"Feature": features, "Importance": importance
feat_importance_df = feat_importance_df.sort_values(by="Importance", ascending=F

print("\nFeature Importances:")
print(feat_importance_df)

oob_preds = bag_boston.oob_prediction_

# OOB MSE (analogous to "Mean of squared residuals" in R)
oob_mse = mean_squared_error(y_train, oob_preds)

# OOB R^2 (fraction of variance explained), scikit-learn calculates this automat
oob_r2 = bag_boston.oob_score_

# Convert R^2 to % Var explained
oob_var_explained = 100 * oob_r2

print(f"Mean of squared residuals (OOB): {oob_mse:.5f}")
print(f"% Var explained (OOB): {oob_var_explained:.2f}")
```



```
Test MSE (Bagged Model): 11.160242529644266

Feature Importances:
       Feature  Importance
6           rm    0.600212
13       lstat    0.226871
8          dis    0.041182
5          nox    0.030586
1         crim    0.021742
12       black    0.019356
0   Unnamed: 0    0.015697
7          age    0.014745
10         tax    0.011418
11     ptratio    0.008078
9          rad    0.003175
3        indus    0.003047
2           zn    0.002202
4         chas    0.001687
Mean of squared residuals (OOB): 18.76511
% Var explained (OOB): 78.62
```

so here i applied bagging (bootstrapping aggregation) using a random forest regression with all features at each split. The first model used 500 trees and the other model used 25 trees.

Bagging with random forest is an ensemble technique that improves models by combining multiple decision trees. Random forest is an extension of bagging that introduces additional randomness in the tree generation. I have a bootstrap sampling mechanism that randomly selects multiple subsets of the training data with replacement and each subset is then used to train an individial decision tree. Theese trees are then combined using average aggregation.

Random forest will then include a out-of-bag (OOB) errpr which explains to us how much of the remaining unused samples can be used to estimate model accuracy without needing a separate validation set.

model	test MSE	OOB MSE	explained variance
500 trees	11.07	15.58	81.11%
25 trees	11.16	18.77	78.62%

So from this we can see that the bagging improved the model compared to the single pruned decision tree which had an MSE of 18.24 and we are now down to 11.1-ranges.

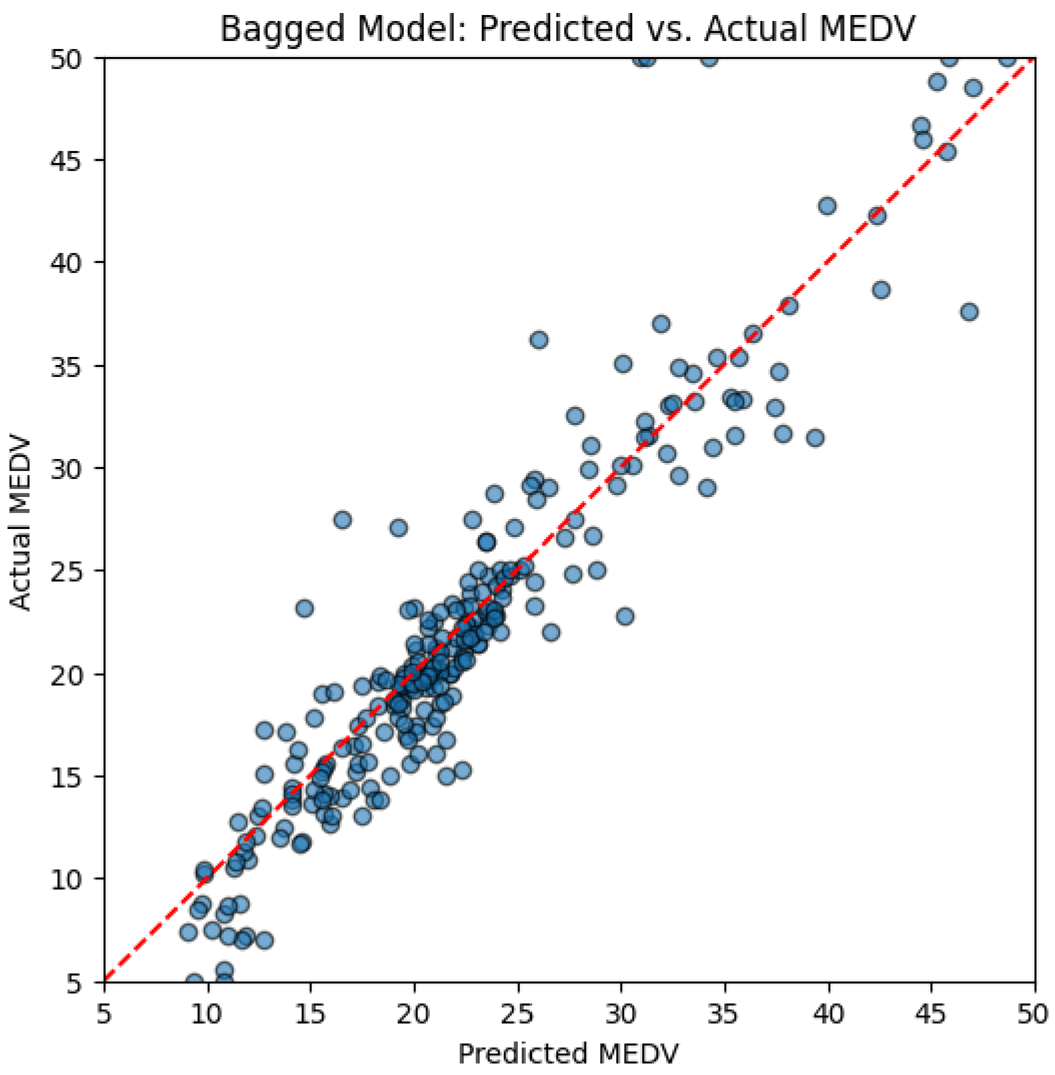
when comparing the same method but with different trees we can see that the model that uses 500 trees has better performance metrics when it comes to test MSE, OOB MSE and variance explained.

apart from this i also measured the importance of each feature by showing a cosistent ranking of how important predictors are for the model and as we can see from that, rm, lstat, dis and nox rank the highest and theese rankings are consistent across both models which means that both models make a fairly good attempt at consistently identifying key predictors.

we can also see from the scatter plots that both models are quite similar. they try to follow the red dotted lines but obviously with quite a lot of deviation and again, a lot of clustering in the lower ranges of MEDV.

In [403…

```
X = boston.drop(columns=['medv'])
y = boston['medv']

bag_boston = RandomForestRegressor(
    n_estimators=500,
    max_features=6,  # same as settingmtry = 13 which ises all features for each
    bootstrap=True,
    oob_score=True,
    random_state=1
)
bag_boston.fit(X_train, y_train)
yhat_bag = bag_boston.predict(X_test)

plt.figure(figsize=(6, 6))
plt.scatter(yhat_bag, y_test, alpha=0.6, edgecolors='k')
plt.xlabel("Predicted MEDV")
plt.ylabel("Actual MEDV")
plt.title("Bagged Model: Predicted vs. Actual MEDV")

lims = [min(yhat_bag.min(), y_test.min()), max(yhat_bag.max(), y_test.max())]
plt.plot(lims, lims, 'r--')
plt.xlim(lims)
plt.ylim(lims)
plt.show()


# Compute test MSE
mse_bag = mean_squared_error(y_test, yhat_bag)
print("Test MSE (Bagged Model):", mse_bag)

importances = bag_boston.feature_importances_
features = X.columns
feat_importance_df = pd.DataFrame({"Feature": features, "Importance": importance
feat_importance_df = feat_importance_df.sort_values(by="Importance", ascending=F

print("\nFeature Importances:")
print(feat_importance_df)

oob_preds = bag_boston.oob_prediction_

# OOB MSE (analogous to "Mean of squared residuals" in R)
oob_mse = mean_squared_error(y_train, oob_preds)

# OOB R^2 (fraction of variance explained), scikit-learn calculates this automat
oob_r2 = bag_boston.oob_score_

# Convert R^2 to % Var explained
oob_var_explained = 100 * oob_r2

print(f"Mean of squared residuals (OOB): {oob_mse:.5f}")
print(f"% Var explained (OOB): {oob_var_explained:.2f}")
```



```
Test MSE (Bagged Model): 11.169320679367592

Feature Importances:
       Feature  Importance
6           rm    0.384064
13       lstat    0.272006
11     ptratio    0.055674
0   Unnamed: 0    0.054816
8          dis    0.042311
1         crim    0.040531
5          nox    0.038456
10         tax    0.026708
7          age    0.025004
3        indus    0.022870
12       black    0.018394
4         chas    0.009161
9          rad    0.005080
2           zn    0.004925
Mean of squared residuals (OOB): 14.39685
% Var explained (OOB): 83.59
```

Here i again used 500 trees but excluded some features and only did it for 6 futures and as we can see this model actually had a slightly higher test MSE but had higher decreases in OOB MSE and had a higher ratio of variance explained from 81.59% for the 13 feature model to 83.59% for the 6 feature model.

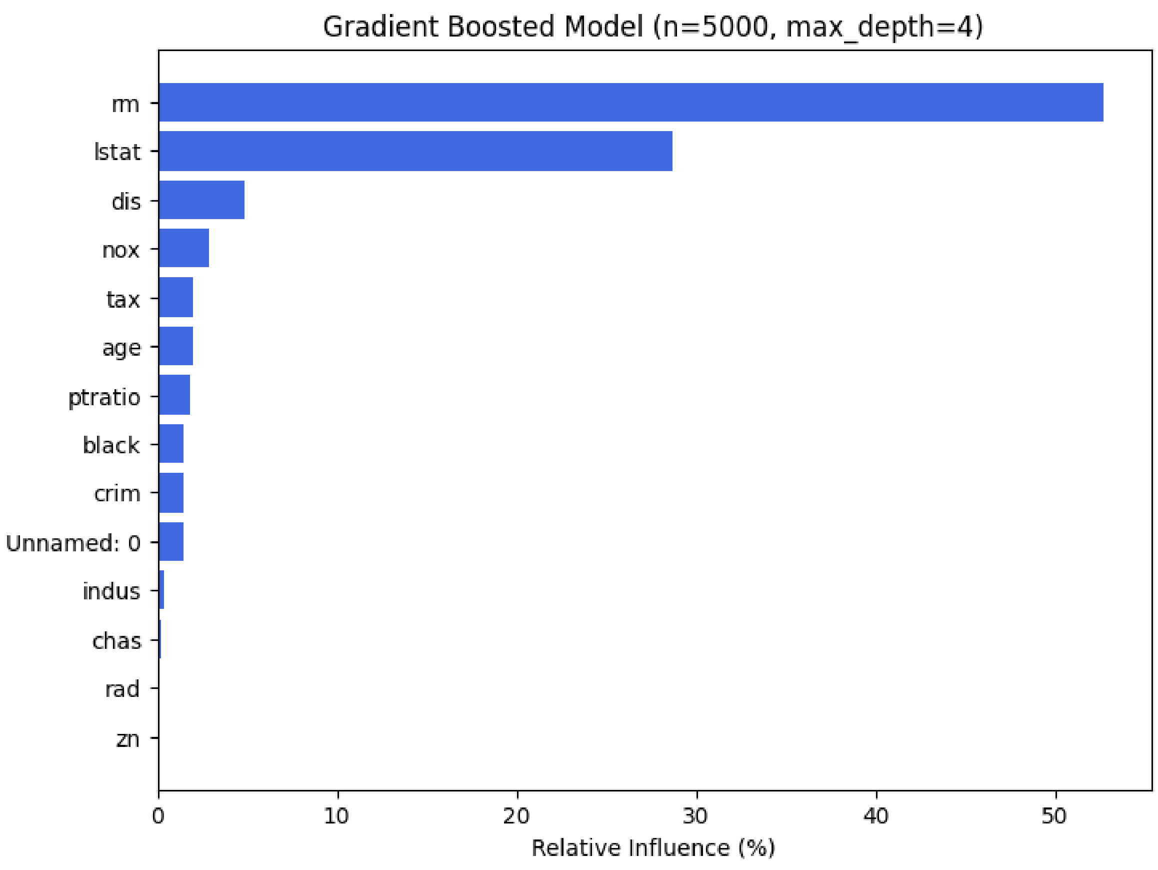
here i also plotted scatterplot where we actually can see that the dots are a bit tighter to the red line and the feature analysis showed that the highest ranking predictors remain which means that the model makes consistent predictions.

Learn and assess regression boosting (trees)

In [404…

```
boost_boston = GradientBoostingRegressor(
    n_estimators=5000,   # like n.trees=5000
    max_depth=4,         # like interaction.depth=4
    random_state=1       # replicates set.seed(1)
)
boost_boston.fit(X_train, y_train)


#Evaluate performance (train & test MSE)
train_mse = mean_squared_error(y_train, boost_boston.predict(X_train))
test_mse = mean_squared_error(y_test, boost_boston.predict(X_test))

print("Boosting (GradientBoostingRegressor) with 5000 trees, depth=4")
print(f"Train MSE: {train_mse:.2f}")
print(f"Test MSE : {test_mse:.2f}")


# Compute feature importances ("relative influence")

feature_importances = boost_boston.feature_importances_

# Convert to percentages that sum to 100
relative_influence = 100.0 * (feature_importances / feature_importances.sum())

# Make a DataFrame similar to R's "summary(boost.boston)"
importances_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Rel. Influence': relative_influence
}).sort_values(by='Rel. Influence', ascending=False)

print("\nRelative Influences (like R's summary(boost.boston)):")
print(importances_df)


# Plot the relative influences as a bar chart
plt.figure(figsize=(8, 6))
plt.barh(importances_df['Feature'], importances_df['Rel. Influence'], color='roy
plt.gca().invert_yaxis()
plt.xlabel('Relative Influence (%)')
plt.title('Gradient Boosted Model (n=5000, max_depth=4)')
plt.show()
```

```
Boosting (GradientBoostingRegressor) with 5000 trees, depth=4
Train MSE: 0.00
Test MSE : 10.82

Relative Influences (like R's summary(boost.boston)):
       Feature  Rel. Influence
6           rm       52.679666
13       lstat       28.671848
8          dis        4.857972
5          nox        2.880143
10         tax        2.000601
7          age        1.969645
11     ptratio        1.843647
12       black        1.441715
1         crim        1.430770
0   Unnamed: 0        1.424467
3        indus        0.415045
4         chas        0.238433
9          rad        0.091219
2           zn        0.054830
```



For this i used gradient boosting which i assume was the boosting technique used in the example because the package "gbm" was imported. Gradient boosting is a method that builds sequential decision trees, where each tree corrects the mistake of the previous one. Unlike random forest that averages independet trees, gradient boosting will build trees sequentially to minimize residual errors.

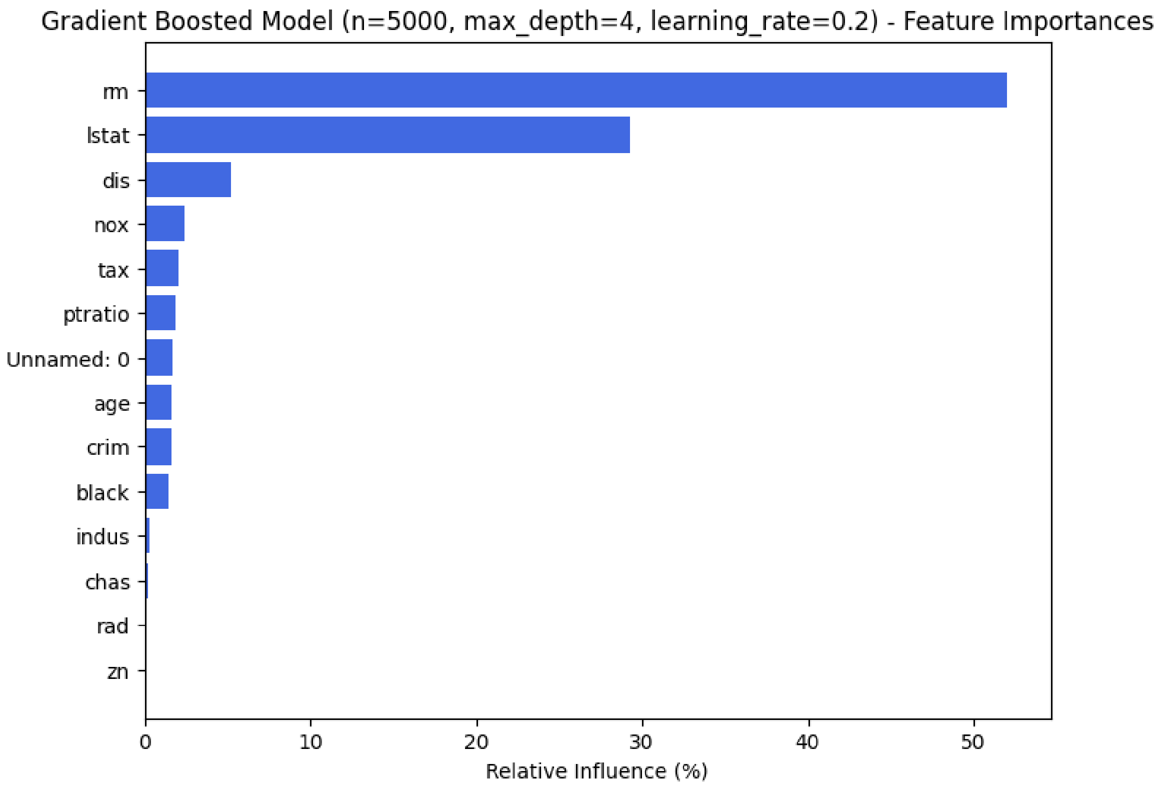
I did 5000 boosting iterations and kept the max depth to 4 and also used the (1)-seed.

The test MSE that i got was 10.82 which is better than bagging which suggests that boosting has a better generalization. There is also a risk of overfitting because the training error is too low (0.0), but it could also be that i calculated it wrongly.

I once again did another feature analysis which showed that rm is the most dominant predictor followed by lstat

```
In [405…
boost_boston = GradientBoostingRegressor(
    n_estimators=5000,   # like n.trees=5000
    max_depth=4,         # like interaction.depth=4
    learning_rate=0.2,   # shrinkage parameter set to 0.2
    random_state=1       # replicates set.seed(1)
)
boost_boston.fit(X_train, y_train)

# Evaluate performance (train & test MSE)
train_mse = mean_squared_error(y_train, boost_boston.predict(X_train))
test_mse = mean_squared_error(y_test, boost_boston.predict(X_test))

print("Boosting (GradientBoostingRegressor) with 5000 trees, depth=4, learning_r
print(f"Train MSE: {train_mse:.2f}")
print(f"Test MSE : {test_mse:.2f}")


# 3. Compute feature importances ("relative influence")
feature_importances = boost_boston.feature_importances_

# Convert to percentages that sum to 100
relative_influence = 100.0 * (feature_importances / feature_importances.sum())

# Make a DataFrame similar to R's "summary(boost.boston)"
importances_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Rel. Influence': relative_influence
}).sort_values(by='Rel. Influence', ascending=False)

print("\nRelative Influences (like R's summary(boost.boston)):")
print(importances_df)

#Plot the relative influences as a bar chart

plt.figure(figsize=(8, 6))
plt.barh(importances_df['Feature'], importances_df['Rel. Influence'], color='roy
plt.gca().invert_yaxis()  # Highest influence at the top
plt.xlabel('Relative Influence (%)')
plt.title('Gradient Boosted Model (n=5000, max_depth=4, learning_rate=0.2) - Fea
plt.show()
```

```
Boosting (GradientBoostingRegressor) with 5000 trees, depth=4, learning_rate=0.2
Train MSE: 0.00
Test MSE : 9.92

Relative Influences (like R's summary(boost.boston)):
       Feature  Rel. Influence
6           rm       52.034911
13       lstat       29.316652
8          dis        5.226337
5          nox        2.415937
10         tax        2.074480
11     ptratio        1.844310
0   Unnamed: 0        1.739872
7          age        1.614592
1         crim        1.597999
12       black        1.426953
3        indus        0.301517
4         chas        0.213892
9          rad        0.125212
2           zn        0.067337
```



Did the exact same thing but just included an argument that would include a learning rate argument for the model which controls which step size the gradient boosting updates predictions in each iteration. This larger learning rate has a faster learning rate and can reach optimal performance with fewer trees but can risk overfitting.

as we can see we got an all-time low test MSE of 9.92 which means that this model is the best treee-based model we have used this far.