lab8

December 5, 2023

0.1 Lab 8: Tree Based Methods

Jack Krebsbach Math 313

Imports

[ ]:
```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
from statsmodels.datasets import get_rdataset
import sklearn.model_selection as skm
from ISLP import load_data , confusion_table
from ISLP.models import ModelSpec as MS
```

New imports

[ ]:
```
from sklearn.tree import (DecisionTreeClassifier as DTC, DecisionTreeRegressor␣
  ↪as DTR,
plot_tree ,
export_text)
from sklearn.metrics import (accuracy_score ,
log_loss)
from sklearn.ensemble import \
(RandomForestRegressor as RF, GradientBoostingRegressor as GBR)
from ISLP.bart import BART
```

Fittting Classification Trees We will start off using the Cars Seats data set.

[ ]:
```
# Load Data
Carseats = load_data('Carseats')
# Where Sales are high
High = np.where(Carseats.Sales > 8,
"Yes", "No")
```

[ ]:
```
model = MS(Carseats.columns.drop('Sales'), intercept=False)
D = model.fit_transform(Carseats)
feature_names = list(D.columns)
X = np.asarray(D)
```

We need to specify to the classifier certain hyperparameters such as `max_depth` or `min_samples_split`.

[ ]:
```
# Initialize classifier
clf = DTC(criterion='entropy', max_depth=3,
random_state=0)
# Fit the classifier
clf.fit(X, High)
```

[ ]:
```
DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
```

[ ]:
```
accuracy_score(High, clf.predict(X))
```

[ ]:
```
0.79
```

This pretty good, with only the default arguments the training error rate is only 21%. We can access the value of the deviance using `log_loss()`.

$-2\sum_m \sum_k n_{mk} \log \hat{p}_{mk}$,

n_{mk} is the number of observations in the mth terminal node.

[ ]:
```
resid_dev = np.sum(log_loss(High, clf.predict_proba(X)))
resid_dev
```

[ ]:
```
0.4710647062649358
```

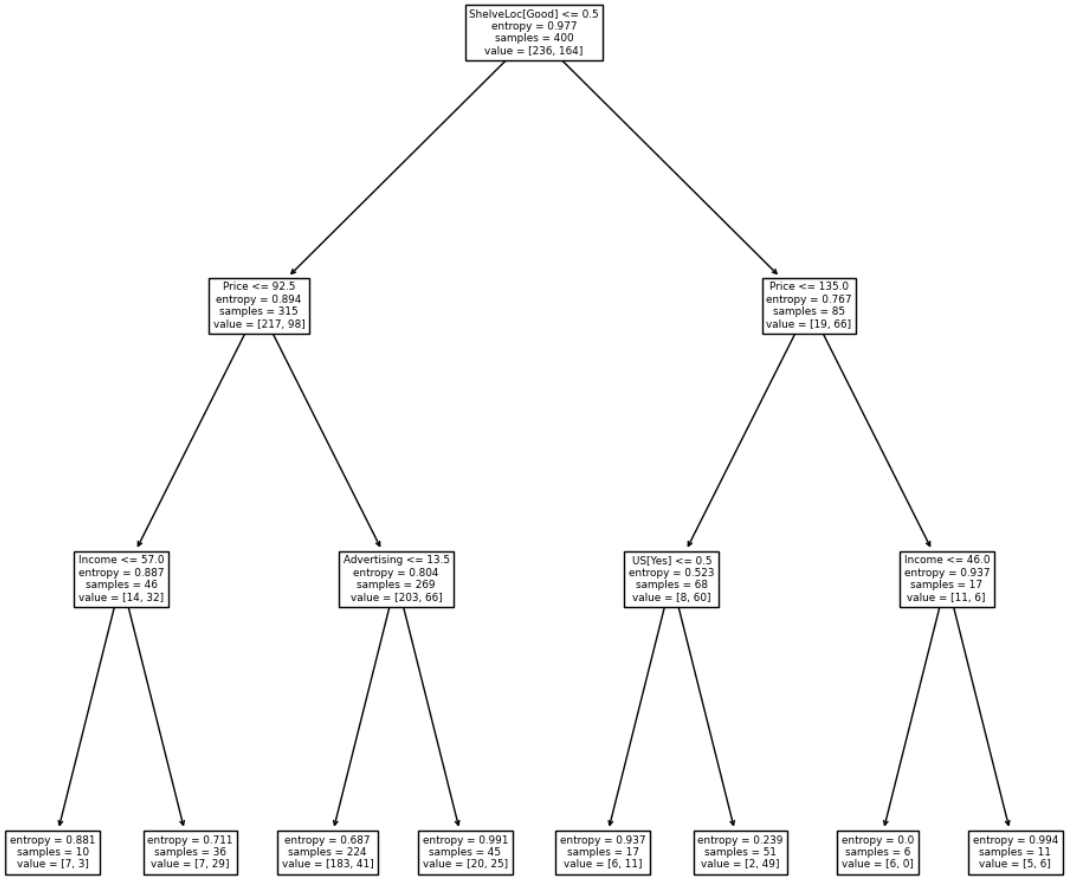
This value is closely related to entropy. A small deviance means the tree generally has good fit to the data.

[ ]:
```
High
```

[ ]:
```
array(['Yes', 'Yes', 'Yes', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No',
       'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes',
       'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No',
       'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No', 'No', 'No',
       'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes', 'No', 'No', 'Yes',
       'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes',
       'No', 'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes',
       'No', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes',
       'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No',
       'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'No',
       'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'Yes', 'Yes',
       'No', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No', 'No', 'No',
       'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No', 'Yes',
       'No', 'Yes', 'No', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'No',
       'No', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes',
       'Yes', 'No', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No',
       'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'Yes', 'Yes',
```

```
'Yes', 'Yes', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'No', 'No',
'No', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes',
'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'Yes', 'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes',
'Yes', 'Yes', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No',
'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes',
'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes',
'No', 'No', 'Yes', 'Yes', 'No', 'No', 'No', 'No', 'Yes', 'No',
'Yes', 'No', 'Yes', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes', 'No',
'Yes', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes',
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'No',
'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No',
'No', 'Yes', 'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No',
'No', 'Yes', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', 'No',
'Yes', 'Yes', 'No', 'Yes', 'No', 'No', 'No', 'Yes', 'No', 'Yes',
'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', 'No',
'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'No', 'No', 'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No',
'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No',
'No', 'Yes', 'No', 'Yes', 'No', 'No', 'No', 'Yes', 'No', 'No',
'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No',
'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes'], dtype='<U3')
```

[ ]:
```
ax = subplots(figsize=(12,12))[1]
plot_tree(clf,
feature_names=feature_names, ax=ax);
```



We see that the most relavent predictor for high sales is shelf location.

We can also get a text representation of the tree by the following.

We can extract different data about the tree: - Split Criterion - Number of observations in each leaf (show_weights=True)

[ ]:
```
print(export_text(clf, feature_names=feature_names,
show_weights=True))
```

```
|--- ShelveLoc[Good] <= 0.50
|   |--- Price <= 92.50
|   |   |--- Income <= 57.00
|   |   |   |--- weights: [7.00, 3.00] class: No
|   |   |--- Income >  57.00
```

```
|   |   |   |--- weights: [7.00, 29.00] class: Yes
|   |--- Price >  92.50
|   |   |--- Advertising <= 13.50
|   |   |   |--- weights: [183.00, 41.00] class: No
|   |   |--- Advertising >  13.50
|   |   |   |--- weights: [20.00, 25.00] class: Yes
|--- ShelveLoc[Good] >  0.50
|   |--- Price <= 135.00
|   |   |--- US[Yes] <= 0.50
|   |   |   |--- weights: [6.00, 11.00] class: Yes
|   |   |--- US[Yes] >  0.50
|   |   |   |--- weights: [2.00, 49.00] class: Yes
|   |--- Price >  135.00
|   |   |--- Income <= 46.00
|   |   |   |--- weights: [6.00, 0.00] class: No
|   |   |--- Income >  46.00
|   |   |   |--- weights: [5.00, 6.00] class: Yes
```

However, to properly evaluate the performance of the classification tree on the data we use a training and testing set.

[ ]:
```
validation = skm.ShuffleSplit(n_splits=1, test_size=200, random_state=0)
results = skm.cross_validate(clf, D, High, cv=validation)
results['test_score']
```

[ ]:
```
array([0.685])
```

Now we see if pruning the tree will improve the performance.

[ ]:
```
(X_train,
X_test ,
High_train ,
High_test) = skm.train_test_split(X, High , test_size=0.5, random_state=0)
```

[ ]:
```
# We do not need to set max_depth because that will be learned through cross␣
↪validation
clf = DTC(criterion='entropy', random_state=0)
clf.fit(X_train, High_train)
accuracy_score(High_test, clf.predict(X_test))
```

[ ]:
```
0.735
```

[ ]:
```
ccp_path = clf.cost_complexity_pruning_path(X_train, High_train)
kfold = skm.KFold(10, random_state=1, shuffle=True)
```

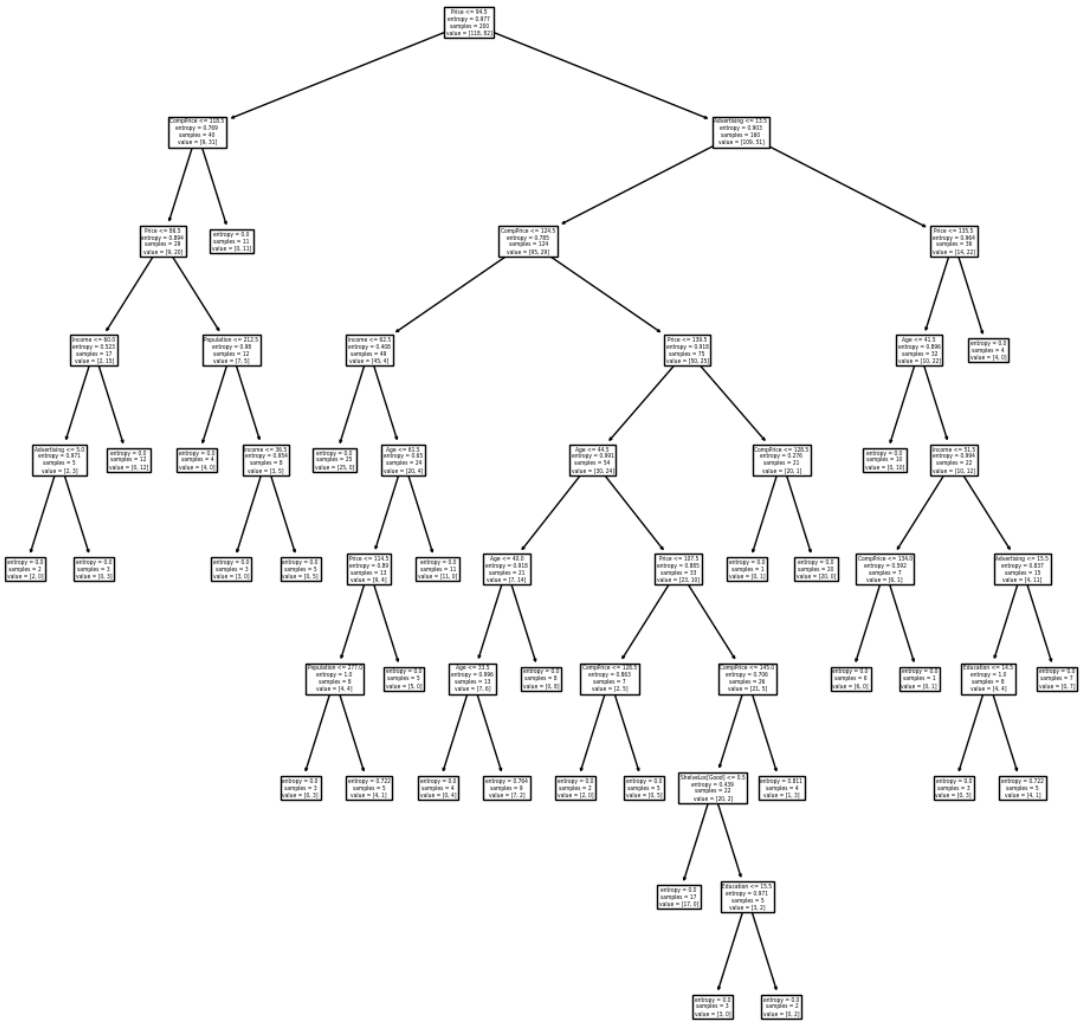
[ ]:
```
grid = skm.GridSearchCV(clf,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True, cv=kfold, scoring='accuracy')
```

```
grid.fit(X_train, High_train)
grid.best_score_
```

[ ]: 0.685

The pruned tree:

[ ]:
```
ax = subplots(figsize=(12, 12))[1]
best_ = grid.best_estimator_
plot_tree(best_,
feature_names=feature_names, ax=ax);
```



[ ]:
```
# Count the leaves
best_.tree_.n_leaves
```

[ ]:
```
30
```

Test it on the test data

[ ]:
```
print(accuracy_score(High_test, best_.predict(X_test)))
confusion = confusion_table(best_.predict(X_test),High_test)
confusion
```

```
0.72
```

[ ]:
```
Truth       No  Yes
Predicted
No          94   32
Yes         24   50
```

Fitting Regression Trees Now we will focus on fitting regression trees to the Boston data.

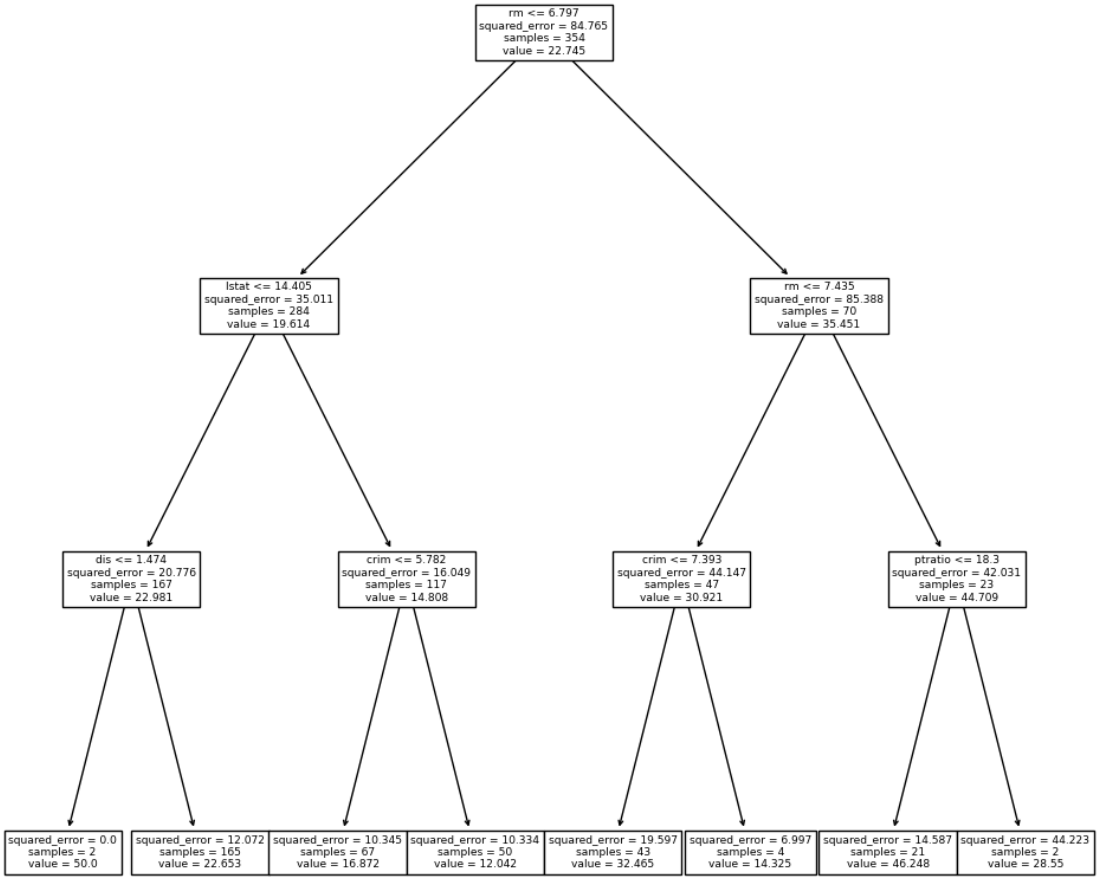
[ ]:
```
# Load the data
Boston = load_data("Boston")
# Make model matrix
model = MS(Boston.columns.drop('medv'), intercept=False)
D = model.fit_transform(Boston)
feature_names = list(D.columns)
# Convert to numpy array
X = np.asarray(D)
```

Use 30% for the test and training set.

[ ]:
```
(X_train,X_test,
 y_train ,
 y_test) = skm.train_test_split(X,
                                Boston['medv'], test_size=0.3, random_state=0)
```

Now that we have the training and testing sets we can fit our regression tree.

[ ]:
```
reg = DTR(max_depth=3)
reg.fit(X_train, y_train)
ax = subplots(figsize=(12,12))[1]
plot_tree(reg,
feature_names=feature_names, ax=ax);
```



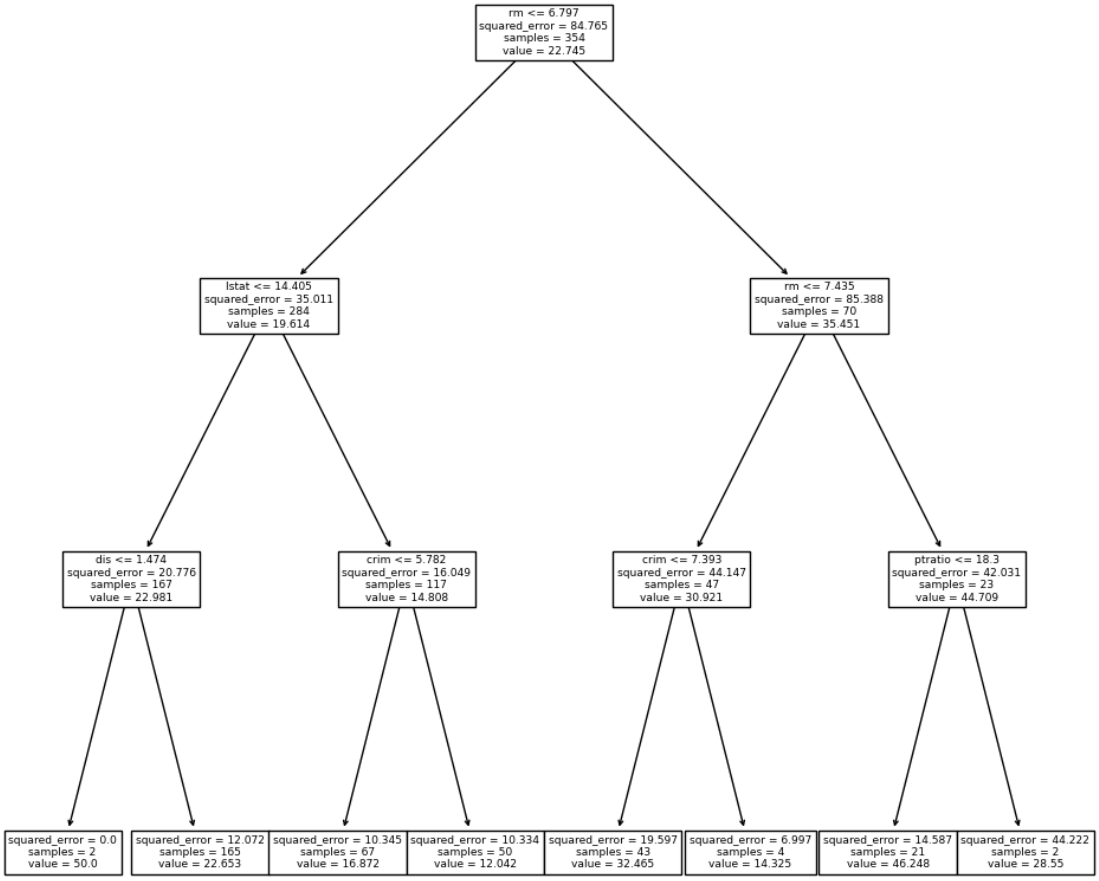
Using the cross-validation function we can see if pruning the tree improves performance.

[ ]:
```
ccp_path = reg.cost_complexity_pruning_path(X_train, y_train)
kfold = skm.KFold(5, shuffle=True, random_state=10)
grid = skm.GridSearchCV(reg,
                        {'ccp_alpha': ccp_path.ccp_alphas}, refit=True,
                        cv=kfold, scoring='neg_mean_squared_error')
G = grid.fit(X_train, y_train)
```

[ ]:
```
# Extract the best tree
best_= grid.best_estimator_
np.mean((y_test - best_.predict(X_test))**2)
```

[ ]: 28.06985754975404

[ ]:
```
ax = subplots(figsize=(12,12))[1]
plot_tree(G.best_estimator_ ,
feature_names=feature_names, ax=ax);
```



The above plot is the best tree based on the cross validation result. We can see the tree is not too dense so is reasonably interpretable.

Bagging and Random Forests

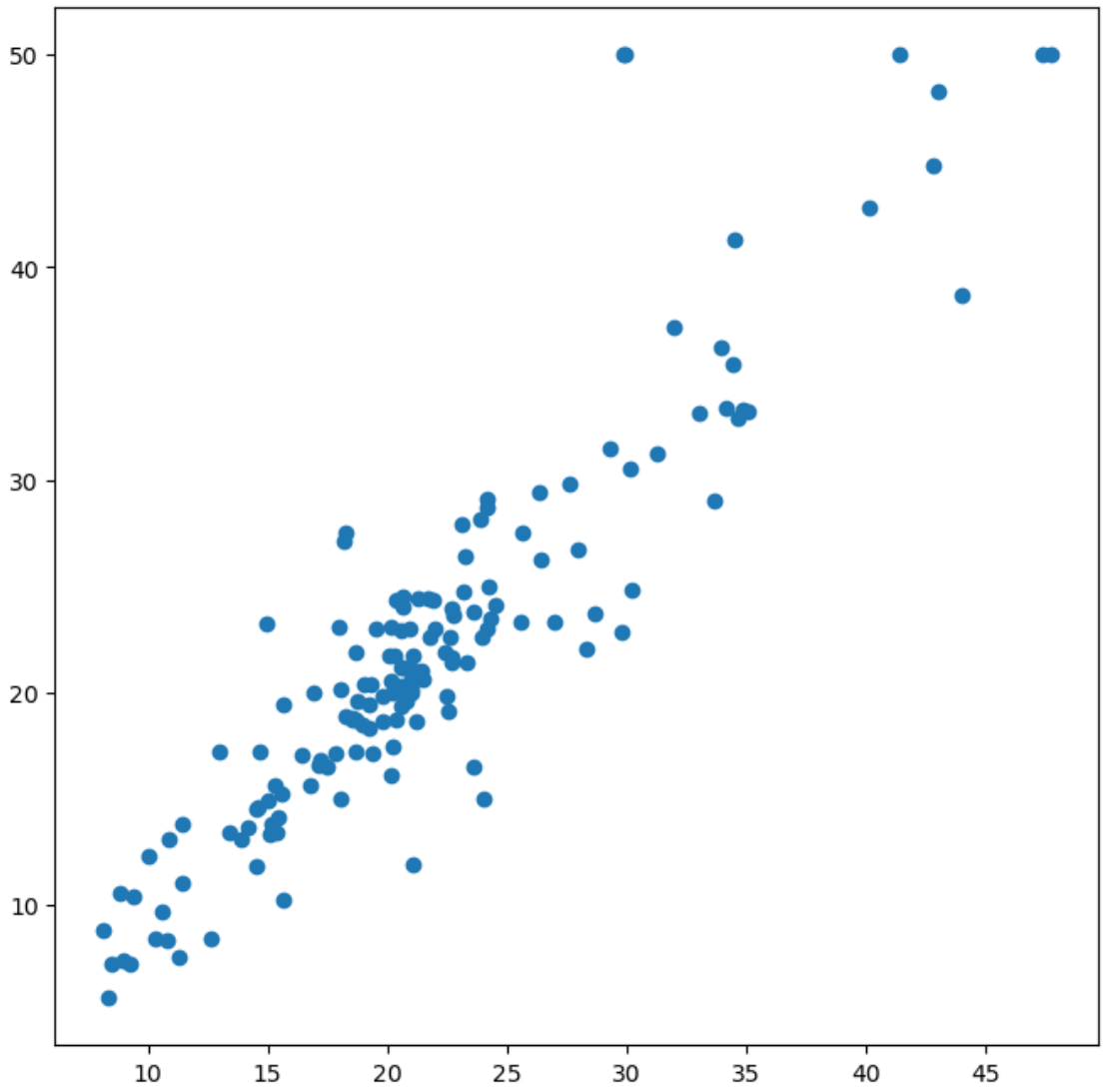
[ ]:
```
bag_boston = RF(max_features=X_train.shape[1], random_state=0)
bag_boston.fit(X_train,y_train)
```

[ ]:
```
RandomForestRegressor(max_features=12, random_state=0)
```

We can visually inspect how the bagged model performs on the test set by plotting the predicted values against the observed values.

[ ]:
```
ax = subplots(figsize=(8,8))[1]
y_hat_bag = bag_boston.predict(X_test)
ax.scatter(y_hat_bag, y_test)
np.mean((y_test - y_hat_bag)**2)
```

[ ]:
```
14.634700151315787
```



We see the MSE with the bagged regression tree is 14.63. Have of that of the pruned single tree.

```
[ ]: # Increase the number of estimators from 100 to 500
bag_boston = RF(max_features=X_train.shape[1], n_estimators=500,
random_state=0).fit(X_train, y_train)
y_hat_bag = bag_boston.predict(X_test)
np.mean((y_test - y_hat_bag)**2)
```

```
[ ]: 14.605662565263161
```

We see that here there is not much change. Now we use random forest, but see that it does worse than the bagged model.

```
[ ]: RF_boston = RF(max_features = 6,random_state=0).fit(X_train, y_train)
y_hat_RF = RF_boston.predict(X_test)
np.mean((y_test - y_hat_RF)**2)
```

```
[ ]: 20.04276446710527
```

```
[ ]: # Extract importance
feature_imp = pd.DataFrame( {'importance':RF_boston.feature_importances_},␣
  ↪index=feature_names)
feature_imp.sort_values(by='importance', ascending=False)
```

```
[ ]:          importance
lstat       0.356203
rm          0.332163
ptratio     0.067270
crim        0.055404
indus       0.053851
dis         0.041582
nox         0.035225
tax         0.025355
age         0.021506
rad         0.004784
chas        0.004203
zn          0.002454
```

Importance in this case is the relative measure of the total decrease in node impurity that results from the variables, averaged over all trees.

Boosting We use `GradientBoostingRegress()` to fit boosted regression trees to the Boston data set.

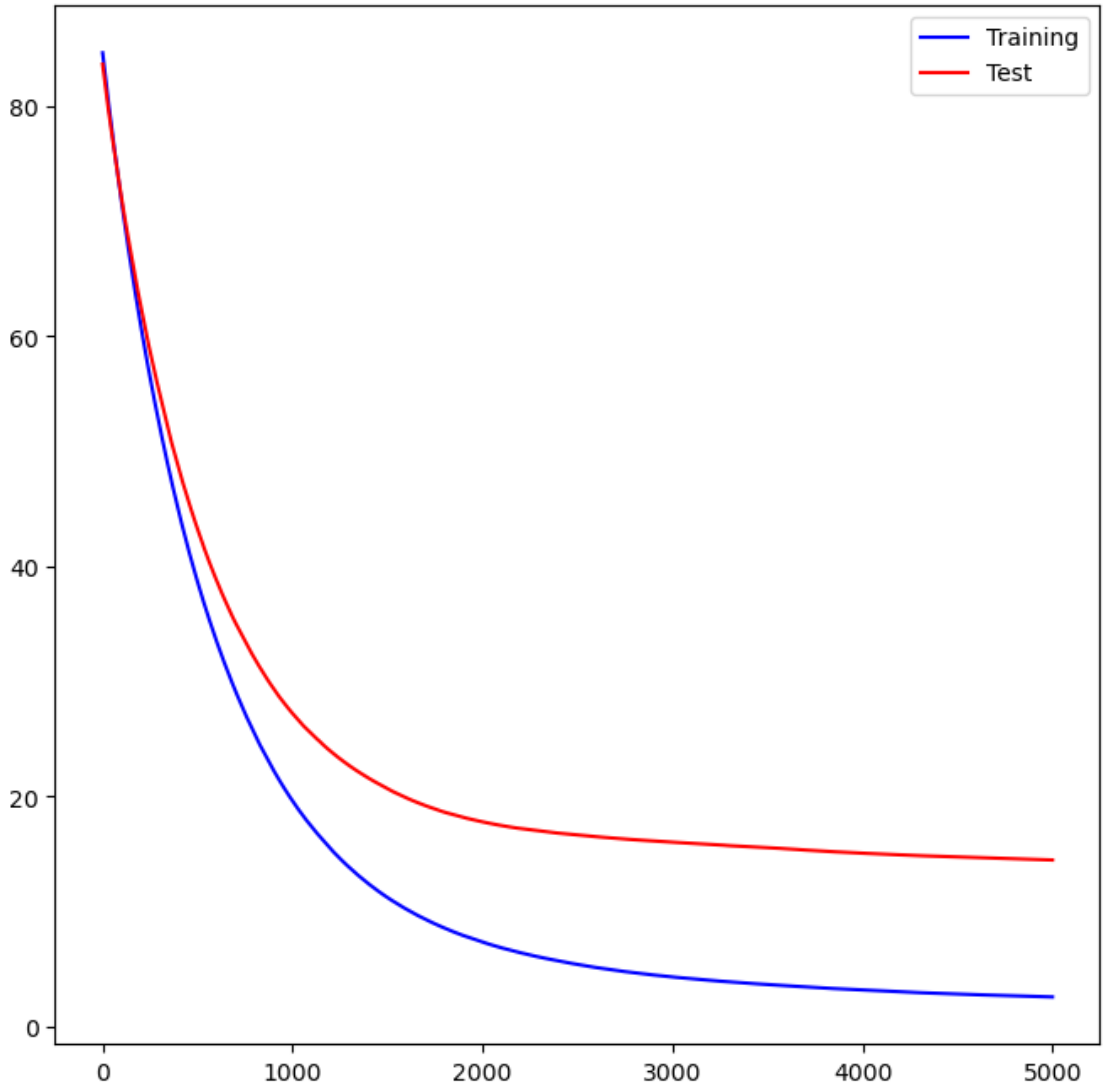
```
[ ]: boost_boston = GBR(n_estimators=5000,
learning_rate=0.001, max_depth=3, random_state=0)
boost_boston.fit(X_train, y_train)
```

```
[ ]: GradientBoostingRegressor(learning_rate=0.001, n_estimators=5000,
                             random_state=0)
```

Here we can see that the training error decreases with `train_score` attribute.

[ ]:
```
# Test error
test_error = np.zeros_like(boost_boston.train_score_)

for idx, y_ in enumerate(boost_boston.staged_predict(X_test)):
    test_error[idx] = np.mean((y_test - y_)**2)
plot_idx = np.arange(boost_boston.train_score_.shape[0])
ax = subplots(figsize=(8,8))[1]
ax.plot(plot_idx, boost_boston.train_score_, 'b',
label='Training')
ax.plot(plot_idx, test_error , 'r', label='Test')
ax.legend();
```



```
[ ]: y_hat_boost = boost_boston.predict(X_test)
     np.mean((y_test - y_hat_boost)**2)
```

```
[ ]: 14.481405918831591
```

The boosted model has a similar MSE to that of the bagged model. We can also use different values of λ.

```
[ ]: boost_boston = GBR(n_estimators=5000,
     learning_rate=0.2, max_depth=3, random_state=0)
     boost_boston.fit(X_train, y_train)
     y_hat_boost = boost_boston.predict(X_test); np.mean((y_test - y_hat_boost)**2)
```

```
[ ]: 14.501514553719565
```

We see essentially no difference between the two different lambdas.

Bayesian Additive Regression Trees This estimator is primarily used for quantitative outcome variables. However, it can also be used to fit logistic and probit models to categorical outcomes.

```
[ ]: # Initialize model
     bart_boston = BART(random_state=0, burnin=5, ndraw=15)
     # Fit the model
     bart_boston.fit(X_train, y_train)
```

```
[ ]: BART(burnin=5, ndraw=15, random_state=0)
```

```
[ ]: yhat_test = bart_boston.predict(X_test.astype(np.float32))
     np.mean((y_test - yhat_test)**2)
```

```
[ ]: 20.739185417498756
```

This MSE is similar to that of Random Forest. Finally, we can see how many times each variable appeared in the collections of trees. This provides a similar metric of variable importance like we saw in boosting and random forests.

```
[ ]: var_inclusion = pd.Series(bart_boston.variable_inclusion_.mean(0), index=D.
      ↪columns)
     var_inclusion
```

```
[ ]: crim       25.466667
     zn         30.600000
     indus      24.933333
     chas       21.133333
     nox        27.333333
     rm         28.800000
     age        23.466667
```

```
dis        26.000000
rad        25.000000
tax        21.733333
ptratio    26.800000
lstat      31.866667
dtype: float64
```