DataCamp Data Science Courses

Extreme Gradient Boosting with XGBoost

Chap 1: Classification with XGBoost

In [1]:
```
# Import plotting modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('ggplot')

import xgboost as xgb
from sklearn.model_selection import train_test_split
```

Welcome to the course!

Supervised Learning

- Regression: covered in chapter 2
- Classification
  - binary: predict a person will buy insurance
    - metric: AUC (Area under the ROC Curve)
  - multi-class: classifying the species of a given bird in an image
    - metric: accuracy score, confusion matrix

common classification models: logistic regression, decision trees.

Supervised Learning considerations

- Features can be either numeric or categorical
- Numeric features should be scaled (Z-scored)
  - e.g. essential to train SVM models
- Categorical features should be encoded (one-hot)

Other kinds of Supervised Learning problems

- Ranking: Predicting an ordering on a set of choices
  - Google search suggestions
- Recommendation: Recommending an item to a user based on his/her consumption history and profile
  - Netflix

Introducing XGBoost

- Optimized gradient-boosting machine learning library
- Originally written in C++
- Why it is popular
  - Speed and performance
  - Core algorithm is parallelizable
  - Consistently outperforms single-algorithm methods
  - State-of-the-art performance in many ML benchmark datasets

In [ ]:
```
# Using XGBoost: A Quick Example

class_data = pd.read_csv("datasets/classification_data.csv")

X, y = class_data.iloc[:,:-1], class_data.iloc[:,-1]
X_train, X_test, y_train, y_test= train_test_split(X, y,test_size=0.2, random_state=123)

xg_cl = xgb.XGBClassifier(objective='binary:logistic',n_estimators=10, seed=123)

xg_cl.fit(X_train, y_train)
preds = xg_cl.predict(X_test)

accuracy = float(np.sum(preds==y_test))/y_test.shape[0]
print("accuracy: %f" % (accuracy))
```

In [ ]:
```
# EXERCISES
```

In [ ]:
```
# XGBoost: Fit/Predict

# Import xgboost
import xgboost as xgb

# Create arrays for the features and the target: X, y
X, y = churn_data.iloc[:,:-1], churn_data.iloc[:,-1]

# Create the training and test sets
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=123)

# Instantiate the XGBClassifier: xg_cl
xg_cl = xgb.XGBClassifier(objective='binary:logistic', n_estimators=10, seed=123)

# Fit the classifier to the training set
xg_cl.fit(X_train,y_train)

# Predict the labels of the test set: preds
preds = xg_cl.predict(X_test)

# Compute the accuracy: accuracy
accuracy = float(np.sum(preds==y_test))/y_test.shape[0]
print("accuracy: %f" % (accuracy))
```

What is a decision tree?

Decision Trees

- Base learner - Individual learning algorithm in an ensemble algorithm
- Composed of a series of binary questions
- Predictions happen at the "leaves" of the tree
  - leaf nodes always contain decision values
- Constructed iteratively (one decision at a time)
  - Until a stopping criterion is met
- Individual decision trees tend to overfit
  - low bias, high variance
  - tend to overfit training data, and generalize poorly to new data

XGBoost

- Uses classification and regression trees (CART)
- Contain real-valued score in each leaf
  - regardless of classification or regression problem
  - can be thresholded to convert into categories for classification problems

In [4]:
```
# EXERCISES
```

In [5]:
```
# Decision trees

# Import the necessary modules
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

bc = datasets.load_breast_cancer()
X = bc.data
y = bc.target

X.shape, y.shape
```

Out[5]:
```
((569, 30), (569,))
```

In [6]:
```
# Create the training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)

# Instantiate the classifier: dt_clf_4
dt_clf_4 = DecisionTreeClassifier(max_depth=4)

# Fit the classifier to the training set
dt_clf_4.fit(X_train,y_train)

# Predict the labels of the test set: y_pred_4
y_pred_4 = dt_clf_4.predict(X_test)

# Compute the accuracy of the predictions: accuracy
accuracy = float(np.sum(y_pred_4==y_test))/y_test.shape[0]
print("accuracy:", accuracy)
```

```
accuracy: 0.9736842105263158
```

What is Boosting?

Boosting overview

- Not a specific machine learning algorithm
- Concept that can be applied to a set of machine learning models
    - "Meta-algorithm"
- Ensemble meta-algorithm used to convert many weak learners into a strong learner

- Weak learner: ML algorithm that is slightly better than chance
    - Example: Decision tree whose predictions are slightly better than 50%
- Boosting converts a collection of weak learners into a strong learner
- Strong learner: Any algorithm that can be tuned to achieve good performance

How boosting is accomplished

- Iteratively learning a set of weak models on subsets of the data
- Weighing each weak prediction according to each weak learner's performance
- Combine the weighted predictions to obtain a single weighted prediction that is much better than the individual predictions themselves!

Model evaluation through cross-validation

- Cross-validation: Robust method for estimating the performance of a model on unseen data
- Generates many non-overlapping train/test splits on training data
- Reports the average test set performance across all data splits

In [ ]:
```
# Cross-validation in XGBoost example

class_data = pd.read_csv("classification_data.csv")

churn_dmatrix = xgb.DMatrix(data=churn_data.iloc[:,:-1],
                            label=churn_data.month_5_still_here)

params={"objective":"binary:logistic","max_depth":4}
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=4,
                    num_boost_round=10, metrics="error", as_pandas=True)

print("Accuracy: %f" %((1-cv_results["test-error-mean"]).iloc[-1]))
```

num_boost_round: number of trees to run

In [8]:
```
# EXERCISES
```

In [9]:
```
# Measuring accuracy

# Create the DMatrix: churn_dmatrix
churn_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:logistic", "max_depth":3}

# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=3, num_boost_round=5, metrics="error", as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Print the accuracy
print(((1-cv_results["test-error-mean"]).iloc[-1]))
```

```
   test-error-mean  test-error-std  train-error-mean  train-error-std
0         0.066824        0.019564          0.025480         0.002451
1         0.061524        0.013876          0.021969         0.001257
2         0.056252        0.010004          0.014945         0.006589
3         0.052734        0.011418          0.012306         0.003300
4         0.054497        0.012485          0.010549         0.004314
0.945502666667
```

In [10]:
```
# Measuring AUC

# Perform cross_validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=3, num_boost_round=5, metrics="auc", as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Print the AUC
print((cv_results["test-auc-mean"]).iloc[-1])
```

```
   test-auc-mean  test-auc-std  train-auc-mean  train-auc-std
0       0.961473      0.024760        0.987225       0.001301
1       0.969078      0.022616        0.993244       0.004295
2       0.972491      0.024377        0.995224       0.003751
3       0.971354      0.025405        0.997125       0.002042
4       0.974002      0.026527        0.997610       0.001871
0.974002
```

When should I use XGBoost?

When to use XGBoost

- You have a large number of training samples
  - Greater than 1000 training samples and less 100 features
  - The number of features < number of training samples
- You have a mixture of categorical and numeric features
  - Or just numeric features

When NOT to use XGBoost

- Image recognition
- Computer vision
- Natural language processing and understanding problems
- When the number of training samples is significantly smaller than the number of features

Chap 2: Regression with XGBoost

In [11]:
```
# Import plotting modules
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
plt.style.use('ggplot')

from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.metrics import mean_squared_error
```

Regression review

Common regression metrics

- Root mean squared error (RMSE)
  - Treats negative and positive differences equally
  - Tends to punish larger differences more than smaller ones
- Mean absolute error (MAE)
  - Not affected by large error values as RMSE
  - Lacks nice mathematical properties
  - Less frequently used as evaluation metric

Common regression algorithms

- Linear regression
- Decision trees
  - Can be used for both regression/classification tasks
  - Important building block for XGBoost models

Objective (loss) functions and base learners

Loss/Objective Functions

- Quantifies how far off a prediction is from the actual result
- Measures the difference between estimated and true values for some collection of data
- **Goal**: Find the model that yields the minimum value of the loss function

- Loss function names in xgboost:
  - **reg:linear** - use for regression problems
  - **reg:logistic** - use for classification problems when you want just decision, not probability
  - **binary:logistic** - use when you want probability rather than just decision

Base Learners

- XGBoost involves creating a meta-model that is composed of many individual models that combine to give a final prediction
- Individual models = base learners
  - Want base learners that are slightly better than random guessing for some part of data but are uniformly bad for the remaining majority of the data.
  - When combined they create final prediction that is **non-linear**
  - Each base learner should be good at distinguishing or predicting different parts of the dataset
- Two kinds of base learners: **tree** and **linear**

In [12]:
```
# TREES BASE LEARNERS example: Scikit-learn API

boston_data = pd.read_csv("datasets/boston.csv")
X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=123)

xg_reg = xgb.XGBRegressor(objective='reg:linear',n_estimators=10, seed=123)

xg_reg.fit(X_train, y_train)
preds = xg_reg.predict(X_test)
```

In [13]:
```
rmse = np.sqrt(mean_squared_error(y_test,preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 9.749041
```

In [14]:
```
# LINEAR BASE LEARNERS Example: Learning API Only

boston_data = pd.read_csv("datasets/boston.csv")
X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=123)

DM_train = xgb.DMatrix(data=X_train,label=y_train)
DM_test = xgb.DMatrix(data=X_test,label=y_test)

# OPTIONS FOR BOOSTER: gbtree(default), gblinear or dart
params = {"booster":"gblinear","objective":"reg:linear"}

xg_reg = xgb.train(params = params, dtrain=DM_train,num_boost_round=10)
preds = xg_reg.predict(DM_test)
```

In [15]:
```
rmse = np.sqrt(mean_squared_error(y_test,preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 5.492696
```

In [16]:
```
# EXERCISES
```

In [17]:
```
# Decision trees as base learners

df = pd.read_csv('datasets/ames_housing_trimmed_processed.csv')
X, y = df.iloc[:,:-1],df.iloc[:,-1]
X.shape,y.shape
```

Out[17]:
```
((1460, 56), (1460,))
```

In [18]:
```
# Create the training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)

# Instantiate the XGBRegressor: xg_reg
xg_reg = xgb.XGBRegressor(objective='reg:linear',n_estimators=10)

# Fit the regressor to the training set
xg_reg.fit(X_train,y_train)

# Predict the labels of the test set: preds
preds = xg_reg.predict(X_test)

# Compute the rmse: rmse
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 78847.401758
```

In [19]:
```
# Linear base learners

# Convert the training and testing sets into DMatrixes: DM_train, DM_test
DM_train = xgb.DMatrix(data=X_train,label=y_train)
DM_test =  xgb.DMatrix(data=X_test,label=y_test)

# Create the parameter dictionary: params
params = {"booster":"gblinear", "objective":"reg:linear"}

# Train the model: xg_reg
xg_reg = xgb.train(dtrain= DM_train, params=params, num_boost_round=5)

# Predict the labels of the test set: preds
preds = xg_reg.predict(DM_test)

# Compute and print the RMSE
rmse = np.sqrt(mean_squared_error(y_test,preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 43566.535658
```

In [20]:
```
# Evaluating model quality

# RMSE Model
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":4}

# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=4, num_boost_round=5, metrics='rmse', as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Extract and print final boosting round metric
print((cv_results["test-rmse-mean"]).tail(1))
```

```
   test-rmse-mean  test-rmse-std  train-rmse-mean  train-rmse-std
0   142980.433594    1193.791602    141767.531250      429.454591
1   104891.394532    1223.158855    102832.544922      322.469930
2    79478.937500    1601.344539     75872.615235      266.475960
3    62411.920899    2220.150028     57245.652344      273.625086
4    51348.279297    2963.377719     44401.298828      316.423666
4    51348.279297
Name: test-rmse-mean, dtype: float64
```

In [21]:
```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":4}

# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=4, num_boost_round=5, metrics='mae', as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Extract and print final boosting round metric
print((cv_results["test-mae-mean"]).tail(1))
```

```
   test-mae-mean  test-mae-std  train-mae-mean  train-mae-std
0  127634.000000   2404.009898   127343.482422     668.308109
1   90122.501954   2107.912810    89770.056641     456.965267
2   64278.558594   1887.567576    63580.791016     263.404950
3   46819.168946   1459.818607    45633.155274     151.883420
4   35670.646485   1140.607452    33587.090820      86.999396
4    35670.646485
Name: test-mae-mean, dtype: float64
```

Regularization and base learners in XGBoost

Regularization

- Regularization is a control on model complexity
- Want models that are both accurate and as simple as possible
- Tweak parameters to limit model complexity by altering loss function
- Regularization parameters in XGBoost:
  - gamma
    - for tree base learners
    - controls whether a node on base learner split based on the expected reduction in the loss that would occur after performing the split
    - higher values lead to fewer splits
    - minimum loss reduction allowed for a split to occur
  - alpha
    - another name for l1 regularization

    - penalty on leaf weights rather than feature weights, as is the case in linear or logistic regression
    - higher alpha values mean more regularization, causes many leaf weights in the base learners to be 0
  - lambda
    - another name for l2 regularization
    - much smoother penalty than l1
    - causes leaf weights to smoothly decrease instead of enforcing strong sparsity constraints on the leaf weights as in l1.

In [22]:
```
# L1 Regularization in XGBoost example

boston_data = pd.read_csv("datasets/boston.csv")
X,y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

boston_dmatrix = xgb.DMatrix(data=X,label=y)
params={"objective":"reg:linear","max_depth":4}

l1_params = [1,10,100]
rmses_l1=[]

for reg in l1_params:
    params["alpha"] = reg
    cv_results = xgb.cv(dtrain=boston_dmatrix, params=params,
                        nfold=4, num_boost_round=10,
                        metrics="rmse", as_pandas=True, seed=123)
    rmses_l1.append(cv_results["test-rmse-mean"] \
                    .tail(1).values[0])
```

In [23]:
```
print("Best rmse as a function of l1:")
print(pd.DataFrame(list(zip(l1_params,rmses_l1)),columns=["l1","rmse"]))
```

```
Best rmse as a function of l1:
    l1      rmse
0    1  3.461474
1   10  3.821152
2  100  4.645519
```

Base learners in XGBoost

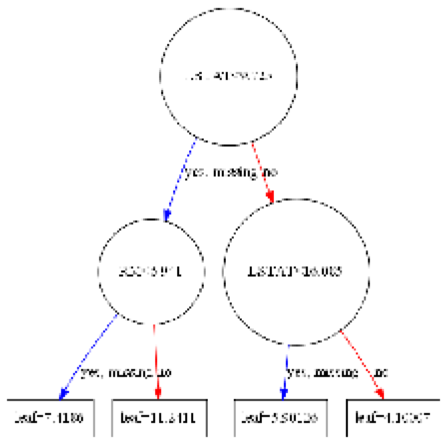
- Linear Base Learner:
  - Sum of linear terms (same as in linear/logistic regression models)
  - When combined into an ensemble, the boosted model is weighted sum of linear models (thus is itself linear)
  - Don't get any nonlinear combination of features in the final model
  - Rarely used
- Tree Base Learner:
  - Decision trees as base model
  - When combined into an ensemble, boosted model is weighted sum of decision trees (nonlinear)
  - Almost exclusively used in XGBoost

In [24]:
```
# EXERCISES
```

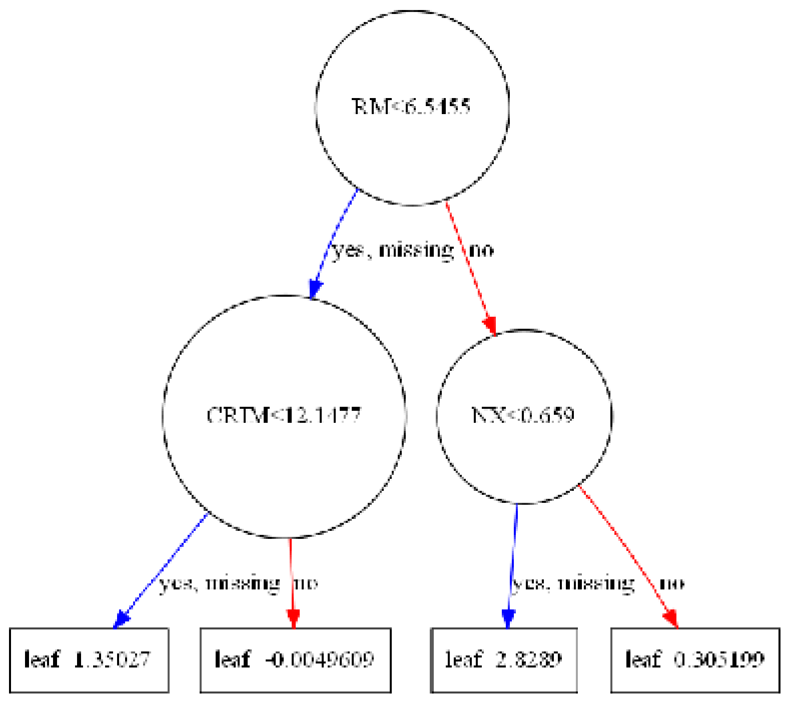
In [25]:
```
# Using L2 regularization in XGBoost

# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

reg_params = [1, 10, 100]

# Create the initial parameter dictionary for varying l2 strength: params
params = {"objective":"reg:linear","max_depth":3}

# Create an empty list for storing rmses as a function of l2 complexity
rmses_l2 = []

# Iterate over reg_params
for reg in reg_params:

    # Update l2 strength
    params["lambda"] = reg

    # Pass this updated param dictionary into cv
    cv_results_rmse = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2, num_boost_round=5, metrics="rmse", as_pandas=True, see

    # Append best rmse (final round) to rmses_l2
    rmses_l2.append(cv_results_rmse["test-rmse-mean"].tail(1).values[0])

# Look at best rmse per l2 param
print("Best rmse as a function of l2:")
print(pd.DataFrame(list(zip(reg_params, rmses_l2)), columns=["l2", "rmse"]))
```

```
Best rmse as a function of l2:
    l2       rmse
0    1   6.022222
1   10   7.201520
2  100  10.692151
```

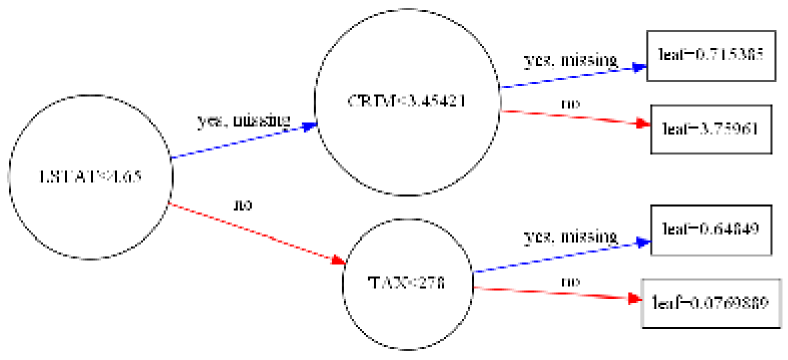
In [26]:
```
# Visualizing individual XGBoost trees

# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":2}

# Train the model: xg_reg
xg_reg = xgb.train(params=params, dtrain=housing_dmatrix, num_boost_round=10)

# Plot the first tree
xgb.plot_tree(xg_reg, num_trees=0)
plt.rcParams["figure.figsize"] = [7,7]
plt.show()
```



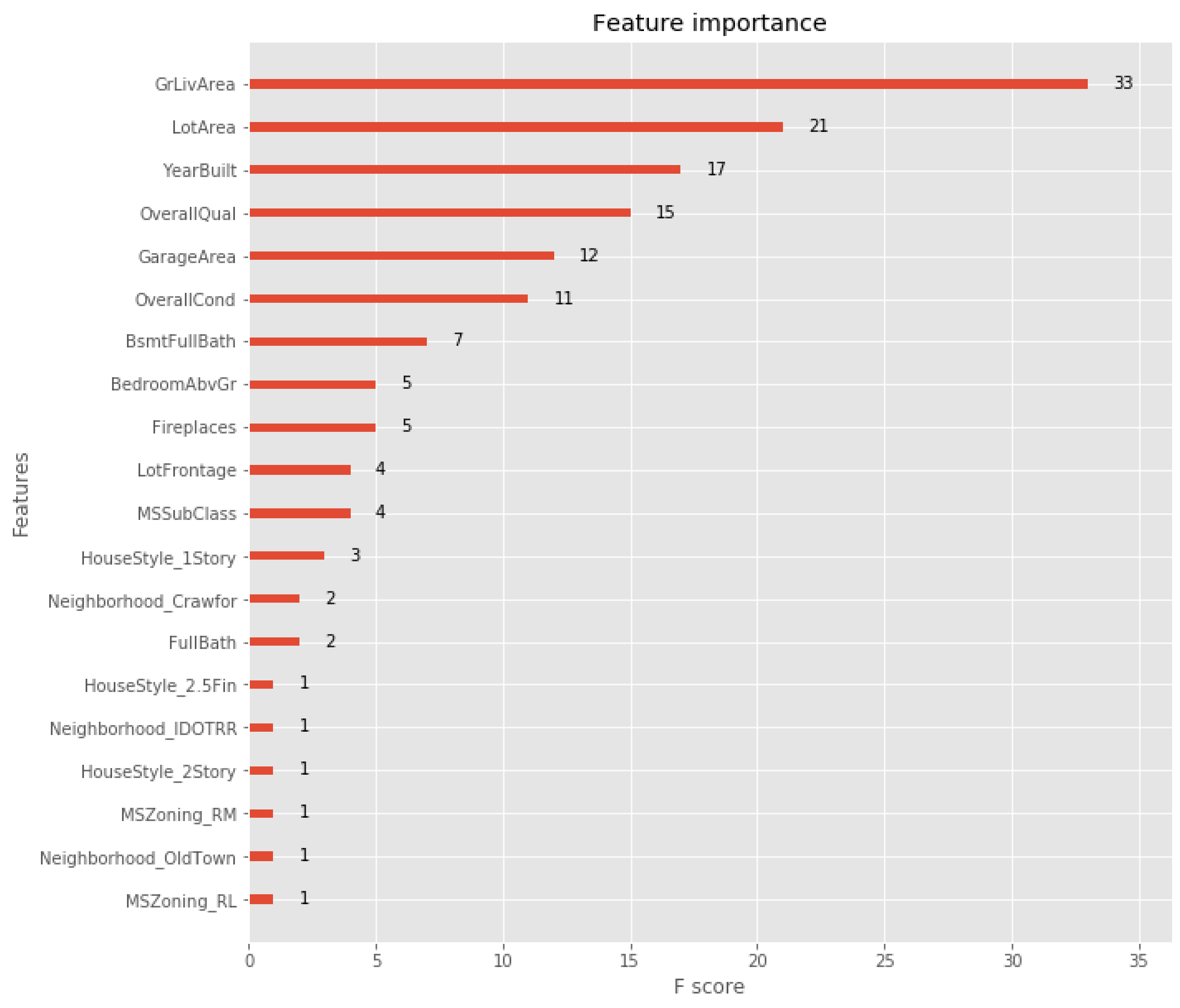
In [27]:
```
# Plot the fifth tree
xgb.plot_tree(xg_reg, num_trees=4)
plt.rcParams["figure.figsize"] = [7,7]
plt.show()
```



In [28]:
```
# Plot the last tree sideways
xgb.plot_tree(xg_reg, num_trees=9,rankdir="LR")
plt.rcParams["figure.figsize"] = [10,10]
plt.show()
```



In [29]:
```
# Visualizing feature importances: What features are most important in my dataset

# Create the DMatrix: housing_dmatrix
df = pd.read_csv('datasets/ames_housing_trimmed_processed.csv')
X, y = df.iloc[:,:-1],df.iloc[:,-1]
housing_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary: params
params = {'objective':'reg:linear',
    'max_depth':4
}

# Train the model: xg_reg
xg_reg = xgb.train(dtrain=housing_dmatrix,params=params,num_boost_round=10)

# Plot the feature importances
xgb.plot_importance(xg_reg)
plt.show()
```



Chap 3: Fine-tuning your XGBoost model

In [30]:
```
# Import plotting modules
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
plt.style.use('ggplot')

from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.metrics import mean_squared_error
```

Why tune your model?

In [31]:
```
# Untuned Model Example

housing_data = pd.read_csv("datasets/ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],housing_data[housing_data.columns.tolist()[-1]]

housing_dmatrix = xgb.DMatrix(data=X,label=y)
untuned_params={"objective":"reg:linear"}

untuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,params=untuned_params,
                                 nfold=4,metrics="rmse",as_pandas=True,seed=123)

print("Untuned rmse: %f" %((untuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Untuned rmse: 34624.229980
```

In [32]:
```
# Tuned Model Example

housing_dmatrix = xgb.DMatrix(data=X,label=y)
tuned_params = {"objective":"reg:linear",'colsample_bytree': 0.3,
                'learning_rate': 0.1, 'max_depth': 5}

tuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,params=tuned_params,
                               nfold=4, num_boost_round=200, metrics="rmse",
                               as_pandas=True, seed=123)

print("Tuned rmse: %f" %((tuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Tuned rmse: 30187.115723
```

In [33]:
```
# EXERCISES
```

In [34]:
```
# Tuning the number of boosting rounds

# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":3}

# Create list of number of boosting rounds
num_rounds = [5, 10, 15]

# Empty list to store final round rmse per XGBoost model
final_rmse_per_round = []

# Iterate over num_rounds and build one model per num_boost_round parameter
for curr_num_rounds in num_rounds:

    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3, num_boost_round=curr_num_rounds, metrics="rmse", as_pandas=

    # Append final round RMSE
    final_rmse_per_round.append(cv_results["test-rmse-mean"].tail(1).values[0])

# Print the resultant DataFrame
num_rounds_rmses = list(zip(num_rounds, final_rmse_per_round))
print(pd.DataFrame(num_rounds_rmses,columns=["num_boosting_rounds","rmse"]))
```

```
   num_boosting_rounds          rmse
0                    5  50903.299479
1                   10  34774.194011
2                   15  32895.098958
```

In [35]:
```
# Automated boosting round selection using early_stopping

# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":4}

# Perform cross-validation with early stopping: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix,params=params,num_boost_round=50,nfold=3,metrics='rmse',early_stopping_rounds=10,as_pand

# Print cv_results
print(cv_results.tail())
```

```
    test-rmse-mean  test-rmse-std  train-rmse-mean  train-rmse-std
45    30758.543620    1947.454953     11356.552734      565.368794
46    30729.971354    1985.698867     11193.557943      552.299272
47    30732.662760    1966.997355     11071.315755      604.090310
48    30712.241537    1957.751573     10950.778320      574.862779
49    30720.854167    1950.511057     10824.865560      576.665674
```

Overview of XGBoost's hyperparameters

Common TREE tunable parameters

- eta:
  - learning rate - value between 0 and 1
  - affects how quickly the model fits the residual error using additional base learners
  - low learning rate will require more boosting rounds to achieve the same reduction in residual error as an XGBoost model with high learning rate
- gamma:
  - min loss reduction to create new tree split
- lambda:
  - L2 reg on leaf weights
- alpha:
  - L1 reg on leaf weights
- max_depth:

- max depth per tree
- must be a positive integer value
- affects how deeply each tree is allowed to grow during any given boosting round
- subsample:
    - % samples used per tree
    - must be a value between 0 and 1
    - fraction of the total training set that can be used for any given boosting round
    - if value is low, fraction of training data used per boosting round would be low, causing underfitting problems.
    - if value is very high, can lead to overfitting
- colsample_bytree:
    - % features used per tree
    - fraction of features that you can select from during any given boosting round
    - same as RandomForest 'max_features' attribute
    - must also be a value between 0 and 1
    - large value means almost all features can be used to build a tree during a boosting
        - may in certain cases overfit a trained model
    - small value means that the fraction of features that can be selected from is very small.
        - smaller values can be thought of as providing additional regularization to the model
- n_estimators / no. of boosting rounds is tunable in both models
    - either no. of trees to build
    - or no. of linear base learners to construct

LINEAR tunable parameters

- no. of tunable parameters is significantly smaller
- lambda:
    - L2 reg on weights associated with any given feature
- alpha:
    - L1 reg on weights
- lambda_bias:
    - L2 reg term on model's bias

In [36]:
```
# EXERCISES
```

In [37]:
```
# Tuning eta

# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree (boosting round)
params = {"objective":"reg:linear", "max_depth":3}

# Create list of eta values and empty list to store final round rmse per xgboost model
eta_vals = [0.001, 0.01, 0.1]
best_rmse = []

# Systematically vary the eta
for curr_val in eta_vals:
    params["eta"] = curr_val
    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix,params=params,nfold=3,
                        num_boost_round=10,early_stopping_rounds=5,
                        metrics='rmse',seed=123)
    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(eta_vals, best_rmse)), columns=["eta","best_rmse"]))
```

```
     eta      best_rmse
0  0.001  195736.406250
1  0.010  179932.182292
2  0.100   79759.411459
```

In [38]:
```python
# Tuning max_depth

# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary
params = {"objective":"reg:linear"}

# Create list of max_depth values
max_depths = [2,5,10,20]
best_rmse = []

# Systematically vary the max_depth
for curr_val in max_depths:
    params["max_depth"] = curr_val
    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix,params=params,metrics='rmse',
                        num_boost_round=10,early_stopping_rounds=5,
                        nfold=2,as_pandas=True,seed=123)
    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(max_depths, best_rmse)),columns=["max_depth","best_rmse"]))
```

```
   max_depth     best_rmse
0          2  37957.468750
1          5  35596.599610
2         10  36065.546875
3         20  36739.576172
```

In [39]:
```python
# Tuning colsample_bytree

# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary
params={"objective":"reg:linear","max_depth":3}

# Create list of hyperparameter values: colsample_bytree_vals
colsample_bytree_vals = [0.1,0.5,0.8,1]
best_rmse = []

# Systematically vary the hyperparameter value
for curr_val in colsample_bytree_vals:

    params['colsample_bytree'] = curr_val

    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2,
                 num_boost_round=10, early_stopping_rounds=5,
                 metrics="rmse", as_pandas=True, seed=123)

    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(colsample_bytree_vals, best_rmse)), columns=["colsample_bytree","best_rmse"]))
```

```
   colsample_bytree     best_rmse
0               0.1  51764.712890
1               0.5  35612.806641
2               0.8  35509.833985
3               1.0  35836.046875
```

Review of Grid Search and Random Search

- Hyperparameter values interact in non-obvious/non-linear ways
- Two common search strategies are:
  - Grid search
  - Random search
- both can be used with XGBoost and scikit-learn packages

Grid Search: Review

- Search exhaustively over a given set of hyperparameters, once per set of hyperparameters
- Number of models = number of distinct values per hyperparameter multiplied across each hyperparameter
  - for 2 hyperparameters to tune with 4 possible values for each, total number of models tried by grid search will be 16
- Pick final model hyperparameter values that give best cross-validated evaluation metric value

In [40]:
```
# Grid Search: Example

from sklearn.model_selection import GridSearchCV

housing_data = pd.read_csv("datasets/ames_housing_trimmed_processed.csv")
X, y = housing_data[housing_data.columns.tolist()[:-1]], housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)

gbm_param_grid = {'learning_rate': [0.01,0.1,0.5,0.9],
                  'n_estimators': [200],
                  'subsample': [0.3, 0.5, 0.9]}

gbm = xgb.XGBRegressor()
grid_mse = GridSearchCV(estimator=gbm, cv=4, verbose=1,
                        param_grid=gbm_param_grid,
                        scoring='neg_mean_squared_error')
grid_mse.fit(X, y)
```

```
Fitting 4 folds for each of 12 candidates, totalling 48 fits

[Parallel(n_jobs=1)]: Done  48 out of  48 | elapsed:   18.8s finished
```

Out[40]:
```
GridSearchCV(cv=4, error_score='raise',
       estimator=XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'learning_rate': [0.01, 0.1, 0.5, 0.9], 'n_estimators': [200], 'subsample': [0.3, 0.5, 0.9]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_squared_error', verbose=1)
```

In [41]:
```
print("Best parameters found: ", grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

```
Best parameters found:  {'learning_rate': 0.1, 'n_estimators': 200, 'subsample': 0.5}
Lowest RMSE found:  28574.9861732
```

Random Search: Review

- Significantly different from Grid search
- No. of models required to iterate over doesn't grow as you expand the overall hyperparameter space
- Create a (possibly infinite) range of hyperparameter values per hyperparameter that you would like to search over
- Set the number of iterations you would like for the random search to continue
- During each iteration, randomly draw a value in the range of specified values for each hyperparameter searched over and train/evaluate a model with those hyperparameters
- After you've reached the maximum number of iterations, select the hyperparameter configuration with the best evaluated score

In [42]:
```
# Random Search: Example

from sklearn.model_selection import RandomizedSearchCV

housing_data = pd.read_csv("datasets/ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)

gbm_param_grid = {'learning_rate': np.arange(0.05,1.05,.05),
                  'n_estimators': [200],
                  'subsample': np.arange(0.05,1.05,.05)}

gbm = xgb.XGBRegressor()
randomized_mse = RandomizedSearchCV(estimator=gbm, n_iter=25,
                                    scoring='neg_mean_squared_error',
                                    param_distributions=gbm_param_grid,
                                    cv=4, verbose=1)
randomized_mse.fit(X, y)
```

```
Fitting 4 folds for each of 25 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   41.1s finished
```

Out[42]:
```
RandomizedSearchCV(cv=4, error_score='raise',
          estimator=XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1),
          fit_params=None, iid=True, n_iter=25, n_jobs=1,
          param_distributions={'learning_rate': array([ 0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,  0.45,
        0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,
        0.95,  1.  ]), 'n_estimators': [200], 'subsample': array([ 0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,  0.45,
        0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,
        0.95,  1.  ])},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', scoring='neg_mean_squared_error',
          verbose=1)
```

In [43]:
```
print("Best parameters found: ",randomized_mse.best_params_)
print("Lowest RMSE found: ",np.sqrt(np.abs(randomized_mse.best_score_)))
```

```
Best parameters found:  {'subsample': 0.35000000000000003, 'n_estimators': 200, 'learning_rate': 0.15000000000000002}
Lowest RMSE found:  28753.2590098
```

In [44]:
```
# EXERCISES
```

In [45]:
```
# Grid Search with XGBoost

# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
    'colsample_bytree': [0.3, 0.7],
    'n_estimators': [50],
    'max_depth': [2, 5]
}

# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor()

# Perform grid search: grid_mse
grid_mse = GridSearchCV(estimator=gbm,param_grid=gbm_param_grid,cv=4,scoring='neg_mean_squared_error',verbose=1)


# Fit grid_mse to the data
grid_mse.fit(X,y)

# Print the best parameters and lowest RMSE
print("Best parameters found: ", grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

```
Fitting 4 folds for each of 4 candidates, totalling 16 fits
Best parameters found:  {'colsample_bytree': 0.3, 'max_depth': 5, 'n_estimators': 50}
Lowest RMSE found:  30031.6171206

[Parallel(n_jobs=1)]: Done  16 out of  16 | elapsed:    1.2s finished
```

In [46]:
```
# Random Search with XGBoost

# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
    'n_estimators': [25],
    'max_depth': range(2, 12)
}

# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor(n_estimators=10)

# Perform random search: grid_mse
randomized_mse = RandomizedSearchCV(estimator=gbm,cv=4,n_iter=5,verbose=1,
                                    param_distributions=gbm_param_grid,
                                    scoring='neg_mean_squared_error')

# Fit randomized_mse to the data
randomized_mse.fit(X,y)

# Print the best parameters and lowest RMSE
print("Best parameters found: ", randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))
```

```
Fitting 4 folds for each of 5 candidates, totalling 20 fits

[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    2.4s finished

Best parameters found:  {'n_estimators': 25, 'max_depth': 11}
Lowest RMSE found:  37502.1924786
```

Limits of Grid Search and Random Search

Grid Search

- Number of models you must build with every additional new parameter grows very quickly
- Optimal if no. of hyperparameters and distinct values per hyperparameter is kept small
- Time taken increases exponentially

Random Search

- Parameter space to explore can be massive
- Randomly jumping throughout the space looking for a "best" result becomes a waiting game

Chap 4: Using XGBoost in pipelines

In [87]:
```
# Import plotting modules
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
plt.style.use('ggplot')
```

Review of pipelines using sklearn

Pipeline Review

- Takes a list of named 2-tuples (name, pipeline_step) as input
- Tuples can contain any arbitrary scikit-learn compatible estimator or transformer object
- Pipeline implements fit/predict methods
- Can be used as input estimator into methods like
    - grid/randomized search approaches - for tuning hyperparameters
    - cross_val_score - for efficient cross-validation and out of sample metric calculation

In [67]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

names = ["crime","zone","industry","charles","no","rooms","age",
         "distance","radial","tax","pupil","aam","lower","med_price"]

data = pd.read_csv("datasets/boston.csv",names=names,skiprows=1)
X, y = data.iloc[:,:-1], data.iloc[:,-1]

rf_pipeline = Pipeline([("st_scaler",StandardScaler()),
                        ("rf_model",RandomForestRegressor())])

scores = cross_val_score(rf_pipeline, X, y, cv=10,
                          scoring="neg_mean_squared_error")
```

In [68]:
```
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print("Final RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.30351262932
```

- neg_mean_squared_error is scikit-learn's way of calculating the mean squared error in an API-compatible way.
- Negative mean squared errors don't exist as all squares must be positive when working with real numbers.
- Thus we simply take the absolute value of the scores, take each of their square roots, and compute their mean to get root mean squared error across all 10 cross-validation folds.

Preprocessing

- Do the same preprocessing in two different ways
- only one of them can be done within a pipeline

First Approach - LabelEncoder and OneHotEncoder

- LabelEncoder: Converts a categorical column of strings into integers
- OneHotEncoder: Takes the column of integers and encodes them as dummy variables
- Cannot be done within a pipeline

Second Approach - DictVectorizer

- can accomplish both steps in one line of code
- found in feature extraction sub-module
- Traditionally used in text processing pipelines
- Converts lists of feature mappings into vectors
- Does not directly work with dataframes
    - Using pandas dataframes we don't initially have these features in lists form
    - Need to convert DataFrame into a list of dictionary entries

In [90]:
```
# EXERCISES
```

In [71]:
```
# Exploratory data analysis
# (of unprocessed Ames housing dataset)

df = pd.read_csv('datasets/ames_unprocessed_data.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 21 columns):
MSSubClass      1460 non-null int64
MSZoning        1460 non-null object
LotFrontage     1201 non-null float64
LotArea         1460 non-null int64
Neighborhood    1460 non-null object
BldgType        1460 non-null object
HouseStyle      1460 non-null object
OverallQual     1460 non-null int64
OverallCond     1460 non-null int64
YearBuilt       1460 non-null int64
Remodeled       1460 non-null int64
GrLivArea       1460 non-null int64
BsmtFullBath    1460 non-null int64
BsmtHalfBath    1460 non-null int64
FullBath        1460 non-null int64
HalfBath        1460 non-null int64
BedroomAbvGr    1460 non-null int64
Fireplaces      1460 non-null int64
GarageArea      1460 non-null int64
PavedDrive      1460 non-null object
SalePrice       1460 non-null int64
dtypes: float64(1), int64(15), object(5)
memory usage: 239.6+ KB
```

In [72]:
```
# Encoding categorical columns I: LabelEncoder

# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Fill missing values with 0
df.LotFrontage = df.LotFrontage.fillna(0)

# Create a boolean mask for categorical columns
categorical_mask = (df.dtypes == 'object')

# Get list of categorical column names
categorical_columns = df.columns[categorical_mask].tolist()

# Print the head of the categorical columns
print(df[categorical_columns].head())

# Create LabelEncoder object: le
le = LabelEncoder()

# Apply LabelEncoder to categorical columns
df[categorical_columns] = df[categorical_columns].apply(lambda x: le.fit_transform(x))

# Print the head of the LabelEncoded categorical columns
print(df[categorical_columns].head())
```

```
  MSZoning Neighborhood BldgType HouseStyle PavedDrive
0       RL      CollgCr     1Fam     2Story          Y
1       RL      Veenker     1Fam     1Story          Y
2       RL      CollgCr     1Fam     2Story          Y
3       RL      Crawfor     1Fam     2Story          Y
4       RL      NoRidge     1Fam     2Story          Y
   MSZoning  Neighborhood  BldgType  HouseStyle  PavedDrive
0         3             5         0           5           2
1         3            24         0           2           2
2         3             5         0           5           2
3         3             6         0           5           2
4         3            15         0           5           2
```

```
In [73]: # Encoding categorical columns II: OneHotEncoder

         # Import OneHotEncoder
         from sklearn.preprocessing import OneHotEncoder

         # Create OneHotEncoder: ohe
         ohe = OneHotEncoder(categorical_features=categorical_mask,sparse=False)

         # Apply OneHotEncoder to categorical columns - output is no longer a dataframe: df_encoded
         df_encoded = ohe.fit_transform(df)

         # Print first 5 rows of the resulting dataset - again, this will no longer be a pandas dataframe
         print(df_encoded[:5, :])

         # Print the shape of the original DataFrame
         print(df.shape)

         # Print the shape of the transformed array
         print(df_encoded.shape)
```

```
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   6.00000000e+01   6.50000000e+01
    8.45000000e+03   7.00000000e+00   5.00000000e+00   2.00300000e+03
    0.00000000e+00   1.71000000e+03   1.00000000e+00   0.00000000e+00
    2.00000000e+00   1.00000000e+00   3.00000000e+00   0.00000000e+00
    5.48000000e+02   2.08500000e+05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   2.00000000e+01   8.00000000e+01
    9.60000000e+03   6.00000000e+00   8.00000000e+00   1.97600000e+03
    0.00000000e+00   1.26200000e+03   0.00000000e+00   1.00000000e+00
    2.00000000e+00   0.00000000e+00   3.00000000e+00   1.00000000e+00
    4.60000000e+02   1.81500000e+05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   6.00000000e+01   6.80000000e+01
    1.12500000e+04   7.00000000e+00   5.00000000e+00   2.00100000e+03
    1.00000000e+00   1.78600000e+03   1.00000000e+00   0.00000000e+00
    2.00000000e+00   1.00000000e+00   3.00000000e+00   1.00000000e+00
    6.08000000e+02   2.23500000e+05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   1.00000000e+00   7.00000000e+01   6.00000000e+01
    9.55000000e+03   7.00000000e+00   5.00000000e+00   1.91500000e+03
    1.00000000e+00   1.71700000e+03   1.00000000e+00   0.00000000e+00
    1.00000000e+00   0.00000000e+00   3.00000000e+00   1.00000000e+00
    6.42000000e+02   1.40000000e+05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
```

```
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   1.00000000e+00   6.00000000e+01   8.40000000e+01
   1.42600000e+04   8.00000000e+00   5.00000000e+00   2.00000000e+03
   0.00000000e+00   2.19800000e+03   1.00000000e+00   0.00000000e+00
   2.00000000e+00   1.00000000e+00   4.00000000e+00   1.00000000e+00
   8.36000000e+02   2.50000000e+05]]
(1460, 21)
(1460, 62)
```

In [75]:
```
# Encoding categorical columns III: DictVectorizer

# Import DictVectorizer
from sklearn.feature_extraction import DictVectorizer

# Convert df into a dictionary: df_dict
df_dict = df.to_dict('records')

# Create the DictVectorizer object: dv
dv = DictVectorizer()

# Apply dv on df: df_encoded
df_encoded = dv.fit_transform(df_dict)

# Print the resulting first five rows
print(df_encoded[:5,:])

# Print the vocabulary
print(dv.vocabulary_)
```

```
  (0, 0)	3.0
  (0, 2)	1.0
  (0, 5)	2.0
  (0, 6)	548.0
  (0, 7)	1710.0
  (0, 8)	1.0
  (0, 9)	5.0
  (0, 10)	8450.0
  (0, 11)	65.0
  (0, 12)	60.0
  (0, 13)	3.0
  (0, 14)	5.0
  (0, 15)	5.0
  (0, 16)	7.0
  (0, 17)	2.0
  (0, 19)	208500.0
  (0, 20)	2003.0
  (1, 0)	3.0
  (1, 3)	1.0
  (1, 4)	1.0
  (1, 5)	2.0
  (1, 6)	460.0
  (1, 7)	1262.0
  (1, 9)	2.0
  (1, 10)	9600.0
  :	:
  (3, 14)	6.0
  (3, 15)	5.0
  (3, 16)	7.0
  (3, 17)	2.0
  (3, 18)	1.0
  (3, 19)	140000.0
  (3, 20)	1915.0
  (4, 0)	4.0
  (4, 2)	1.0
  (4, 4)	1.0
  (4, 5)	2.0
  (4, 6)	836.0
  (4, 7)	2198.0
  (4, 8)	1.0
  (4, 9)	5.0
  (4, 10)	14260.0
  (4, 11)	84.0
  (4, 12)	60.0
  (4, 13)	3.0
  (4, 14)	15.0
  (4, 15)	5.0
  (4, 16)	8.0
  (4, 17)	2.0
  (4, 19)	250000.0
  (4, 20)	2000.0
{'MSSubClass': 12, 'MSZoning': 13, 'LotFrontage': 11, 'LotArea': 10, 'Neighborhood': 14, 'BldgType': 1, 'HouseStyle': 9, 'OverallQual': 16, 'OverallCond': 15, 'YearBuilt': 20, 'Remodeled': 18, 'GrLivArea': 7, 'BsmtFullBath': 2, 'BsmtHalfBath': 3, 'FullBath': 5, 'HalfBath': 8, 'BedroomAbvGr': 0, 'Fireplaces': 4, 'GarageArea': 6, 'PavedDrive': 17, 'SalePrice': 19}
```

In [77]:
```
# Preprocessing within a pipeline

# Import necessary modules
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction import DictVectorizer

X, y = df.iloc[:,:-1], df.iloc[:,-1]

# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)

# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
         ("xgb_model", xgb.XGBRegressor())]

# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)

# Fit the pipeline
xgb_pipeline.fit(X.to_dict('records'),y)
```

Out[77]:
```
Pipeline(memory=None,
     steps=[('ohe_onestep', DictVectorizer(dtype=<class 'numpy.float64'>, separator='=', sort=True,
        sparse=False)), ('xgb_model', XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_ch...
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1))])
```

Incorporating XGBoost into pipelines

To have XGBoost in pipeline, use its scikit-learn API within a pipeline object.

In [80]:
```
# Scikit-Learn Pipeline Example With XGBoost

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

names = ["crime","zone","industry","charles","no","rooms","age",
         "distance","radial","tax","pupil","aam","lower","med_price"]
data = pd.read_csv("datasets/boston.csv",names=names,skiprows=1)
X, y = data.iloc[:,:-1], data.iloc[:,-1]

xgb_pipeline = Pipeline([("st_scaler",StandardScaler()),
                         ("xgb_model",xgb.XGBRegressor())])

scores = cross_val_score(xgb_pipeline, X, y,cv=10,
                         scoring="neg_mean_squared_error")
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print("Final XGB RMSE:", final_avg_rmse)
```

```
Final XGB RMSE: 4.02719593323
```

Additional Components Introduced For Pipelines

- sklearn_pandas: (sklearn and pandas both not always work seamlessly)
  - DataFrameMapper - Interoperability between pandas and scikit-learn
  - CategoricalImputer - Allow for imputation of categorical variables before conversion to integers
- sklearn.preprocessing:
  - Imputer - Native imputation of numerical columns in scikit-learn
- sklearn.pipeline:
  - FeatureUnion - combine multiple pipelines of features into a single pipeline of features

In [100]:
```
# EXERCISES
```

In [82]:
```
# Cross-validating your XGBoost model

# Import necessary modules
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

X, y = df.iloc[:,:-1], df.iloc[:,-1]

# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)

# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
         ("xgb_model", xgb.XGBRegressor(max_depth=2, objective="reg:linear"))]

# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)

# Cross-validate the model
cross_val_scores = cross_val_score(xgb_pipeline,X.to_dict('records'),y,cv=10,scoring='neg_mean_squared_error')

# Print the 10-fold RMSE
print("10-fold RMSE: ", np.mean(np.sqrt(np.abs(cross_val_scores))))
```

```
10-fold RMSE:  30343.4865518
```

In [100]:
```
# Kidney disease case study I: Categorical Imputer

kidney_data = pd.read_csv('datasets/chronic_kidney_disease.csv',header=None,na_values='?')

kidney_feature_names = ['age','bp','sg','al','su','bgr','bu','sc','sod',
                        'pot','hemo','pcv','wc','rc','rbc','pc','pcc',
                        'ba','htn','dm','cad','appet','pe','ane']
kidney_target_name = ['class']
df.columns = kidney_feature_names + kidney_target_name
X, y = kidney_data.iloc[:,:-1], kidney_data.iloc[:,-1]
```

In [ ]:
```
# Import necessary modules
from sklearn_pandas import DataFrameMapper
from sklearn_pandas import CategoricalImputer

# Check number of nulls in each feature column
nulls_per_column = X.isnull().sum()
print(nulls_per_column)

# Create a boolean mask for categorical columns
categorical_feature_mask = X.dtypes == object

# Get list of categorical column names
categorical_columns = X.columns[categorical_feature_mask].tolist()

# Get list of non-categorical column names
non_categorical_columns = X.columns[~categorical_feature_mask].tolist()

# Apply numeric imputer
numeric_imputation_mapper = \
DataFrameMapper([([numeric_feature], Imputer(strategy="median")) \
                  for numeric_feature in non_categorical_columns], \
                 input_df=True,
                 df_out=True)

# Apply categorical imputer
categorical_imputation_mapper = \
DataFrameMapper([(category_feature, CategoricalImputer()) \
                  for category_feature in categorical_columns],\
                 input_df=True,
                 df_out=True)
```

In [ ]:
```
# Kidney disease case study II: Feature Union

# Import FeatureUnion
from sklearn.pipeline import FeatureUnion

# Combine the numeric and categorical transformations
numeric_categorical_union = \
FeatureUnion([("num_mapper", numeric_imputation_mapper),\
              ("cat_mapper", categorical_imputation_mapper)])
```

In [ ]:
```
# Kidney disease case study III: Full pipeline

# Create full pipeline
pipeline = Pipeline([
                     ("featureunion", numeric_categorical_union),
                     ("dictifier", Dictifier()),
                     ("vectorizer", DictVectorizer(sort=False)),
                     ("clf", xgb.XGBClassifier(max_depth=3))
                    ])

# Perform cross-validation
cross_val_scores = cross_val_score(pipeline, X, y, scoring="roc_auc", cv=3)

# Print avg. AUC
print("3-fold AUC: ", np.mean(cross_val_scores))
```

Tuning XGBoost hyperparameters

NOTE: Remember, in order to pass hyperparameters to the appropriate step,

- you have to name the parameters in the dictionary with the name of the step being referenced
- followed by 2 underscore signs and then the name of the hyperparameter you want to iterate over.

Since the xgboost step is called xgb_model, all of our hyperparameter keys will start with xgboost_model__.

In [105]:
```
# Tuning XGBoost hyperparameters in a Pipeline

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import RandomizedSearchCV

names = ["crime","zone","industry","charles","no","rooms","age",
         "distance","radial","tax","pupil","aam","lower","med_price"]
data = pd.read_csv("datasets/boston.csv",names=names,skiprows=1)
X, y = data.iloc[:,:-1],data.iloc[:,-1]

xgb_pipeline = Pipeline([("st_scaler",StandardScaler()),
                         ("xgb_model",xgb.XGBRegressor())])

gbm_param_grid = {'xgb_model__subsample': np.arange(.05, 1, .05),
                  'xgb_model__max_depth': np.arange(3,20,1),
                  'xgb_model__colsample_bytree': np.arange(.1,1.05,.05) }

randomized_neg_mse = \
RandomizedSearchCV (estimator=xgb_pipeline,n_iter=10,
                    param_distributions=gbm_param_grid,
                    scoring='neg_mean_squared_error', cv=4)

randomized_neg_mse.fit(X, y)
```

Out[105]:
```
RandomizedSearchCV(cv=4, error_score='raise',
          estimator=Pipeline(memory=None,
     steps=[('st_scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('xgb_model', XGBRegressor(base_score=0.5, b
ooster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_chil...       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1))]),
          fit_params=None, iid=True, n_iter=10, n_jobs=1,
          param_distributions={'xgb_model__subsample': array([ 0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,  0.45,
        0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,  0.95]), 'xgb_model__max_depth': array([ 3,  4,  5,  6,  7,
  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]), 'xgb_model__colsample_bytree': array([ 0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.3
5,  0.4 ,  0.45,  0.5 ,
        0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,  0.95,  1.  ])},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', scoring='neg_mean_squared_error',
          verbose=0)
```

In [107]:
```
# Tuning XGBoost hyperparameters in a Pipeline II

print("Best rmse: ", np.sqrt(np.abs(randomized_neg_mse.best_score_)))
```

```
Best rmse:  4.47691694445
```

In [108]:
```
print("Best model: ", randomized_neg_mse.best_estimator_)
```

```
Best model:  Pipeline(memory=None,
     steps=[('st_scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('xgb_model', XGBRegressor(base_score=0.5, b
ooster='gbtree', colsample_bylevel=1,
       colsample_bytree=1.0000000000000004, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=11, min_child_weight=1, missing...0, reg_lambda=1, scale_pos_weight=1,
       seed=None, silent=True, subsample=0.75000000000000011))])
```

In [103]:
```
# EXERCISES
```

In [ ]:
```
# Bringing it all together

# Create the parameter grid
gbm_param_grid = {
    'clf__learning_rate': np.arange(0.05, 1, 0.05),
    'clf__n_estimators': np.arange(50, 200, 50),
    'clf__max_depth': np.arange(3, 10, 1)}

# Perform RandomizedSearchCV
randomized_roc_auc = RandomizedSearchCV(estimator=pipeline,param_distributions=gbm_param_grid,scoring='roc_auc',cv=2,n_iter=2,verbo

# Fit the estimator
randomized_roc_auc.fit(X,y)

# Compute metrics
print(randomized_roc_auc.best_score_)
print(randomized_roc_auc.best_estimator_)
```

Final Thoughts

What We Have Covered And You Have Learned

- Using XGBoost for classification tasks
- Using XGBoost for regression tasks
- Tuning XGBoost's most important hyperparameters
- Incorporating XGBoost into sklearn pipelines
- Advanced functions that allow us to seamlessly work with Pandas DataFrames and scikit-learn.

What We Have NOT COVERED (And How You Can Proceed)

- Using XGBoost for ranking/recommendation problems (Netflix/Amazon problem)
  - can be done by modifying the loss function we use when constructing the model
- Using more sophisticated hyperparameter tuning strategies for tuning XGBoost models (Bayesian Optimization)
  - entire companies have been created just for using this method in tuning models.
  - example: the company 'sigpot'
- Using XGBoost as part of an ensemble of other models for regression/classification
  - predictions we get from XGBoost model can be combined with other models
  - very powerful additional way to squeeze the last bit of juice from the data.

In [ ]:

In [ ]: