Extreme Gradient Boosting with XGBoost

Efficient modelling of tabular dataset

Black Raven (James Ng)
11 Oct 2020

This is a memo to share what I have learnt in Extreme Gradient Boosting with XGBoost, capturing the learning objectives as well as my personal notes. The course is taught by Sergey Fogelson from DataCamp, and it includes 4 chapters:

Chapter 1. Classification with XGBoost
Chapter 2. Regression with XGBoost
Chapter 3. Fine-tuning your XGBoost model
Chapter 4. Using XGBoost in pipelines

Photo by Joakim Honkasalo on Unsplash

Do you know the basics of supervised learning and want to use state-of-the-art models on real-world datasets? Gradient boosting is currently one of the most popular techniques for efficient modelling of tabular datasets of all sizes. XGboost is a very fast, scalable implementation of gradient boosting, with models using XGBoost regularly winning online data science competitions and being used at scale across different industries. In this course, you'll learn how to use this powerful library alongside pandas and scikit-learn to build and tune supervised learning models. You'll work with real-world datasets to solve classification and regression problems.

Chapter 1. Classification with XGBoost

This chapter will introduce you to the fundamental idea behind XGBoost—boosted learners. Once you understand how XGBoost works, you'll apply it to solve a common classification problem found in industry: predicting whether a customer will stop being a customer at some point in the future.

Classification – supervised learning, relies on labelled data, eg. does an image contain a person’s face, whether a customer will buy a product.

Most versatile evaluation metric for binary classification models = Area under ROC curve (AUC).

For multi-class classification problems, use accuracy score and confusion matrix.

Which of these is a classification problem?

Given below are 4 potential machine learning problems you might encounter in the wild. Pick the one that is a classification problem.

- ○ Given past performance of stocks and various other financial data, predicting the exact price of a given stock (Google) tomorrow.
- ○ Given a large dataset of user behaviors on a website, generating an informative segmentation of the users based on their behaviors.
- ◉ Predicting whether a given user will click on an ad given the ad content and metadata associated with the user.
- ○ Given a user's past behavior on a video platform, presenting him/her with a series of recommended videos to watch next.

Answer: Predicting whether a given user will click on an ad given the ad content and metadata associated with the user.

Which of these is a binary classification problem?

A classification problem involves predicting the category a given data point belongs to out of a finite set of possible categories. Depending on how many possible categories there are to predict, a classification problem can be either binary or multi-class. Let's do another quick refresher here. Your job is to pick the **binary** classification problem out of the following list of supervised learning problems.

- (●) Predicting whether a given image contains a cat.
- ( ) Predicting the emotional valence of a sentence (Valence can be positive, negative, or neutral).
- ( ) Recommending the most tax-efficient strategy for tax filing in an automated accounting system.
- ( ) Given a list of symptoms, generating a rank-ordered list of most likely diseases.

Answer: Predicting whether a given image contains a cat. A binary classification problem involves picking between 2 choices.

Introducing XGBoost

- Optimised gradient-boosting machine learning library
- Originally developed as a C++ command-line application
- Has APIs in several languages: Python, R, Scala, Julia, Java
- Popular because of speed and performance
- Core algorithm is parallelizable, able to harness all of the processing power of modern multi-core computers and GPUs
- Consistently outperforms single-algorithm methods
- State-of-the-art performance in many ML tasks

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
class_data = pd.read_csv("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))
```

XGBoost: Fit/Predict

It's time to create your first XGBoost model! As Sergey showed you in the video, you can use the scikit-learn `.fit()` / `.predict()` paradigm that you are already familiar to build your XGBoost models, as the `xgboost` library has a scikit-learn compatible API!

Here, you'll be working with churn data. This dataset contains imaginary data from a ride-sharing app with user behaviors over their first month of app usage in a set of imaginary cities as well as whether they used the service 5 months after sign-up. It has been pre-loaded for you into a DataFrame called `churn_data` - explore it in the Shell!

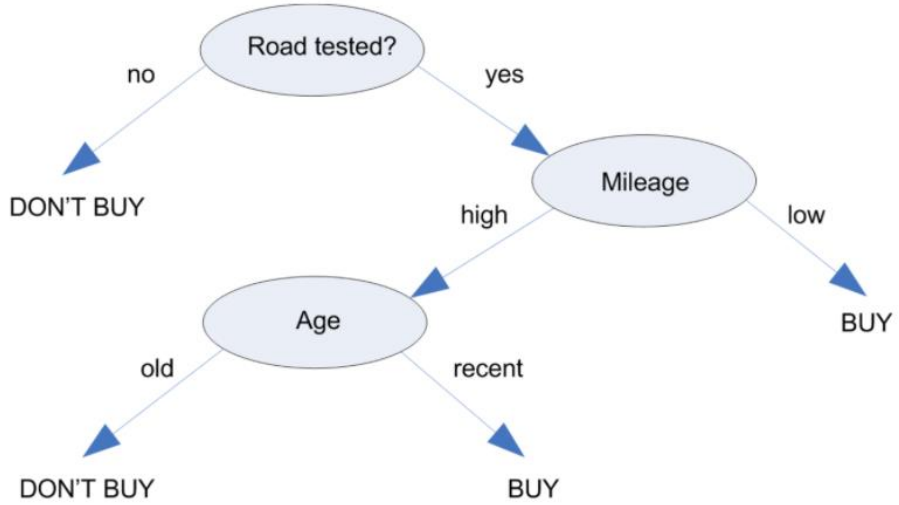
Your goal is to use the first month's worth of data to predict whether the app's users will remain users of the service at the 5 month mark. This is a typical setup for a churn prediction problem. To do this, you'll split the data into training and test sets, fit a small `xgboost` model on the training set, and evaluate its performance on the test set by computing its accuracy.

`pandas` and `numpy` have been imported as `pd` and `np`, and `train_test_split` has been imported from `sklearn.model_selection`. Additionally, the arrays for the features and the target have been created as `X` and `y`.

```
# 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
```

```
<script.py> output:
    accuracy: 0.743300
```

What is a decision tree?



Base learner = individual learning algorithm in an ensemble, composed of a series of binary questions, constructed iteratively (one decision at a time), predictions happen at the 'leaves' (end) of the tree.

Decision trees are low bias and high variance, tend to overfit the training data and generalise poorly on new data.

XGBoost uses a slightly different kind of a decision tree, called a classification and regression tree (CART). The leaf nodes of CART trees contain a real-value score, regardless of whether they are used for classification or regression. The real-valued scores can then be thresholded to convert into categories for classification problems if necessary.

Decision trees

Your task in this exercise is to make a simple decision tree using scikit-learn's `DecisionTreeClassifier` on the `breast cancer` dataset that comes pre-loaded with scikit-learn.

This dataset contains numeric measurements of various dimensions of individual tumors (such as perimeter and texture) from breast biopsies and a single outcome value (the tumor is either malignant, or benign).

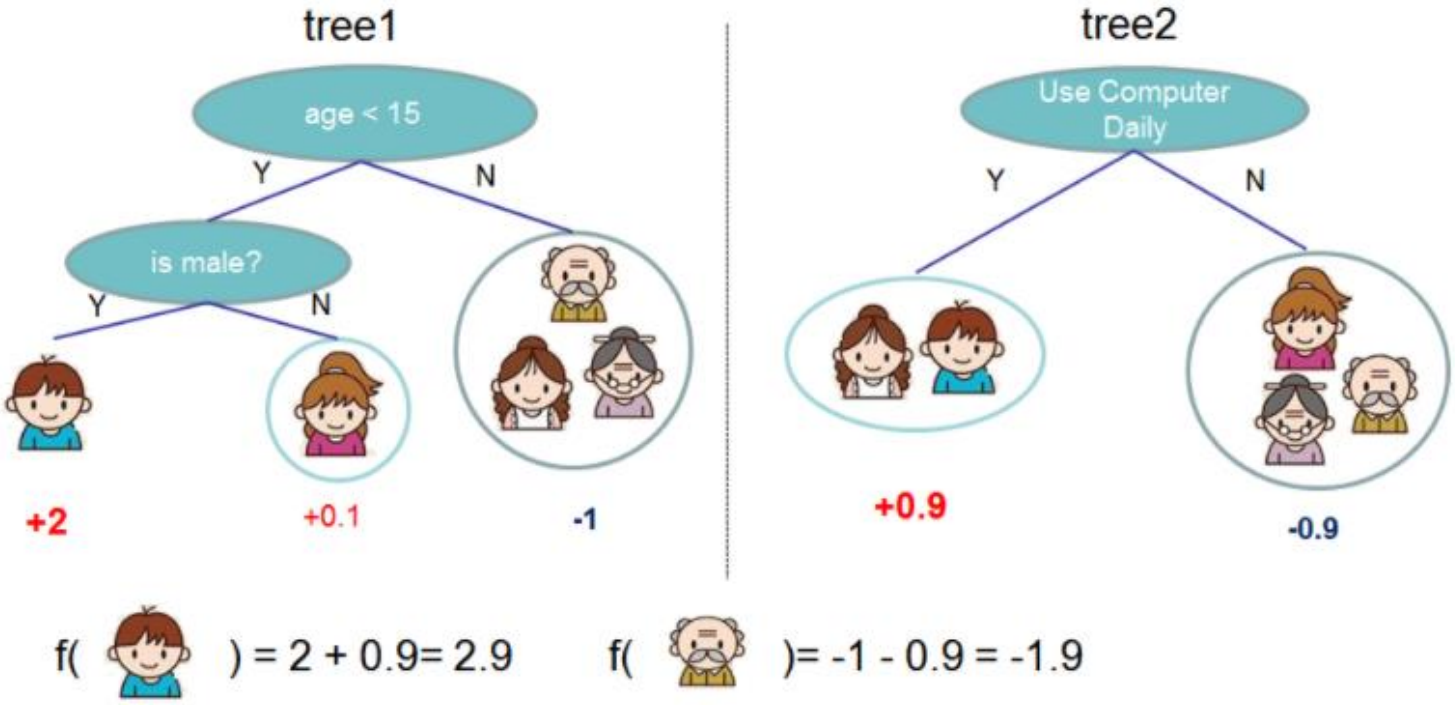
We've preloaded the dataset of samples (measurements) into `X` and the target values per tumor into `y`. Now, you have to split the complete dataset into training and testing sets, and then train a `DecisionTreeClassifier`. You'll specify a parameter called `max_depth`. Many other parameters can be modified within this model, and you can check all of them out **here**.

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

# 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)
```

```
<script.py> output:
    accuracy: 0.9649122807017544
```

What is Boosting?

Boosting is not a specific machine learning algorithm, but a concept (meta-algorithm) that can be applied to a set of machine learning models. Ensemble meta-algorithm is used to convert many weak learners (algorithms that is slightly better than chance) into a strong learner.



Cross validation = robust method for estimating the performance of a model on unseen data, generates many non-overlapping train/test splits on training data and reports the average test set performance across all data splits.

```
import xgboost as xgb
import pandas as pd
churn_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]))
```

```
Accuracy: 0.88315
```

Measuring accuracy

You'll now practice using XGBoost's learning API through its baked in cross-validation capabilities. As Sergey discussed in the previous video, XGBoost gets its lauded performance and efficiency gains by utilizing its own optimized data structure for datasets called a `DMatrix`.

In the previous exercise, the input datasets were converted into `DMatrix` data on the fly, but when you use the `xgboost` `cv` object, you have to first explicitly convert your data into a `DMatrix`. So, that's what you will do here before running cross-validation on `churn_data`.

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

# Create the DMatrix from X and y: 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]))
```

```
<script.py> output:
       train-error-mean  train-error-std  test-error-mean  test-error-std
    0           0.28232         0.002366          0.28378        0.001932
    1           0.26951         0.001855          0.27190        0.001932
    2           0.25605         0.003213          0.25798        0.003963
    3           0.25090         0.001845          0.25434        0.003827
    4           0.24654         0.001981          0.24852        0.000934
    0.75148
```

`cv_results` stores the training and test mean and standard deviation of the error per boosting round (tree built) as a DataFrame. From `cv_results`, the final round `'test-error-mean'` is extracted and converted into an accuracy, where accuracy is `1-error`. The final accuracy of around 75% is an improvement from earlier!

Measuring AUC

Now that you've used cross-validation to compute average out-of-sample accuracy (after converting from an error), it's very easy to compute any other metric you might be interested in. All you have to do is pass it (or a list of metrics) in as an argument to the `metrics` parameter of `xgb.cv()`.

Your job in this exercise is to compute another common metric used in binary classification - the area under the curve (`"auc"`). As before, `churn_data` is available in your workspace, along with the DMatrix `churn_dmatrix` and parameter dictionary `params`.

```
# 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])
```

```
<script.py> output:
       train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
    0        0.768893       0.001544       0.767863      0.002820
    1        0.790864       0.006758       0.789157      0.006846
    2        0.815872       0.003900       0.814476      0.005997
    3        0.822959       0.002018       0.821682      0.003912
    4        0.827528       0.000769       0.826191      0.001937
    0.826191
```

An AUC of 0.84 is quite strong. As you have seen, XGBoost's learning API makes it very easy to compute any metric you may be interested in. In Chapter 3, you'll learn about techniques to fine-tune your XGBoost models to improve their performance even further. For now, it's time to learn a little about exactly **when** to use XGBoost.

When should I use XGBoost?

When a large number of training samples: >1000 training samples and <100 features, mixture of categorical and numerical features, or just numerical features.

Not for image recognition, not computer vision, not Natural Language Processing (NLP), not when number of training samples (<100) are lesser then number of features

Using XGBoost

XGBoost is a powerful library that scales very well to many samples and works for a variety of supervised learning problems. That said, as Sergey described in the video, you shouldn't always pick it as your default machine learning library when starting a new project, since there are some situations in which it is not the best option. In this exercise, your job is to consider the below examples and select the one which would be the best use of XGBoost.

- ○ Visualizing the similarity between stocks by comparing the time series of their historical prices relative to each other.
- ○ Predicting whether a person will develop cancer using genetic data with millions of genes, 23 examples of genomes of people that didn't develop cancer, 3 genomes of people who wound up getting cancer.
- ○ Clustering documents into topics based on the terms used in them.
- ◉ Predicting the likelihood that a given user will click an ad from a very large clickstream log with millions of users and their web interactions.

Chapter 2. Regression with XGBoost

After a brief review of supervised regression, you'll apply XGBoost to the regression task of predicting house prices in Ames, Iowa. You'll learn about the two kinds of base learners that XGboost can use as its weak learners, and review how to evaluate the quality of your regression models.

Regression review

Common regression metrics: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE).

Decision trees can be used for both regression and classification tasks, which makes them prime candidates to be building blocks for XGBoost models.

Which of these is a regression problem?

Here are 4 potential machine learning problems you might encounter in the wild. Pick the one that is a clear example of a regression problem.

- Recommending a restaurant to a user given their past history of restaurant visits and reviews for a dining aggregator app.
- Predicting which of several thousand diseases a given person is most likely to have given their symptoms.
- Tagging an email as spam/not spam based on its content and metadata (sender, time sent, etc.).
- (●) Predicting the expected payout of an auto insurance claim given claim properties (car, accident type, driver prior history, etc.).

Objective (loss) functions and base learners

Objective function or loss function quantifies how far off our prediction is from the actual result for a given data point. It maps the difference between the prediction and the target to a real number. The goal is to minimise the loss function in a machine learning model.
Loss function names in XGBoost:
· **reg:linear** – used for regression problems
· **reg:logistic** – used for classification problems with decision
· **binary:logistic** – used for classification problems with probability

XGBoost involves creating a meta-model that is composed of many individual models that combine to give a final prediction. Each individual models that are trained and combined are called base learners. The goal of XGBoost is to have base learners that is slightly better than random guessing on certain subsets of training examples, and uniformly bad at the remainder. So that when all of the predictions are combined, the

uniformly bad predictions cancel out, and those slightly better than chance combine into a single very good prediction.

Trees as base learners example: Scikit-learn API

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

boston_data = pd.read_csv("boston_housing.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)
rmse = np.sqrt(mean_squared_error(y_test,preds))

print("RMSE: %f" % (rmse))
```

```
RMSE: 129043.2314
```

Linear base learners example: learning API only

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

boston_data = pd.read_csv("boston_housing.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)
params = {"booster":"gblinear","objective":"reg:linear"}
xg_reg = xgb.train(params = params, dtrain=DM_train, num_boost_round=10)
```

```
rmse = np.sqrt(mean_squared_error(y_test,preds))

print("RMSE: %f" % (rmse))
```

```
RMSE: 124326.24465
```

Decision trees as base learners

It's now time to build an XGBoost model to predict house prices - not in Boston, Massachusetts, as you saw in the video, but in Ames, Iowa! This dataset of housing prices has been pre-loaded into a DataFrame called `df`. If you explore it in the Shell, you'll see that there are a variety of features about the house and its location in the city.

In this exercise, your goal is to use trees as base learners. By default, XGBoost uses trees as base learners, so you don't have to specify that you want to use trees here with `booster="gbtree"`.

`xgboost` has been imported as `xgb` and the arrays for the features and the target are available in `X` and `y`, respectively.

```
# 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, seed=123)

# 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))
```

```
<script.py> output:
    [07:17:08] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of
        reg:squarederror.
    RMSE: 78847.401758
```

Linear base learners

Now that you've used trees as base models in XGBoost, let's use the other kind of base model that can be used with XGBoost - a linear learner. This model, although not as commonly used in XGBoost, allows you to create a regularized linear regression using XGBoost's powerful learning API. However, because it's uncommon, you have to use XGBoost's own non-scikit-learn compatible functions to build the model, such as `xgb.train()`.

In order to do this you must create the parameter dictionary that describes the kind of booster you want to use (similarly to how **you created the dictionary in Chapter 1** when you used `xgb.cv()`). The key-value pair that defines the booster type (base model) you need is `"booster":"gblinear"`.

Once you've created the model, you can use the `.train()` and `.predict()` methods of the model just like you've done in the past. Here, the data has already been split into training and testing sets, so you can dive right into creating the `DMatrix` objects required by the XGBoost learning API.

```
# 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(params = params, dtrain=DM_train, 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))
```

```
<script.py> output:
    [07:19:47] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of
        reg:squarederror.
    RMSE: 44519.215959
```

Interesting - it looks like linear base learners performed better!

Evaluating model quality

It's now time to begin evaluating model quality.
Here, you will compare the RMSE and MAE of a cross-validated XGBoost model on the Ames housing data. As in previous exercises, all necessary modules have been pre-loaded and the data is available in the DataFrame `df`.

```
# 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))
```

```
   train-rmse-mean  train-rmse-std  test-rmse-mean  test-rmse-std
0    141767.531250      429.448328   142980.429688    1193.794436
1    102832.542969      322.473304   104891.396485    1223.159762
2     75872.619140      266.472468    79478.935547    1601.344218
3     57245.651367      273.625016    62411.921875    2220.149857
4     44401.295899      316.422824    51348.281250    2963.378741
4    51348.28125
Name: test-rmse-mean, dtype: float64
```

```
# 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))
```

```
   train-mae-mean  train-mae-std  test-mae-mean  test-mae-std
0   127343.476562     668.340697  127633.984375   2404.011173
1    89770.050782     456.959206   90122.503906   2107.909888
2    63580.790039     263.405712   64278.561524   1887.567869
3    45633.153320     151.884919   46819.166992   1459.820450
4    33587.093750      86.999137   35670.645508   1140.606558
4    35670.645508
Name: test-mae-mean, dtype: float64
```

Regularization and base learners in XGBoost

Regularization = to peanalise models as they become more complex. So, loss functions in XGBoost are used to find models that are both accurate and as simple as they can possibly be.

Regularization parameters in XGBoost:
· gamma – minimum loss reduction allowed for a split to occur.
· alpha – **L1 regularization** on leaf weights, larger values mean more regularization.
· lambda – **L2 regularization** on leaf weights, much smoother penalty than L1, leaf weights to smoothly decrease instead of enforcing strong sparsity constraints.

```python
import xgboost as xgb
import pandas as pd
boston_data = pd.read_csv("boston_data.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])
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  69572.517742
1   10  73721.967141
2  100  82312.312413
```

Tree-Based Learner = decision tree, boosted model is weighted sum of decision trees, non-linear, almost exclusively used in XGBoost.

Using regularization in XGBoost

Having seen an example of l1 regularization in the video, you'll now vary the l2 regularization penalty - also known as `"lambda"` - and see its effect on overall model performance on the Ames housing dataset.

```
# 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, seed=123)

    # 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  52275.359375
1   10  57746.064453
2  100  76624.625000
```

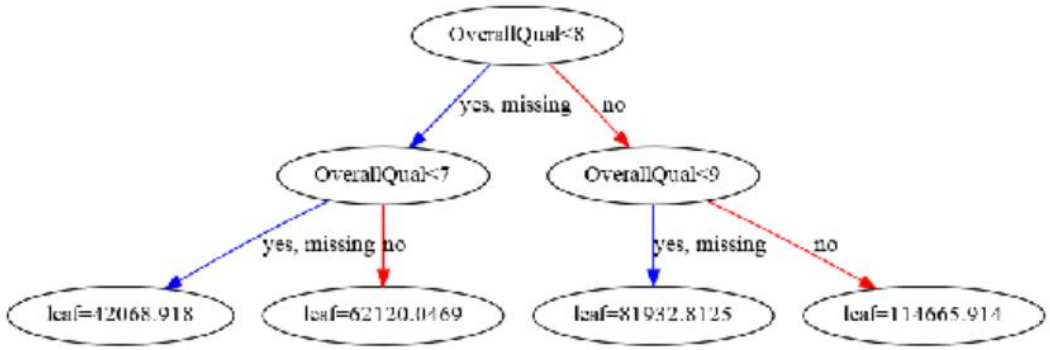
It looks like as as the value of `'lambda'` increases, so does the RMSE.

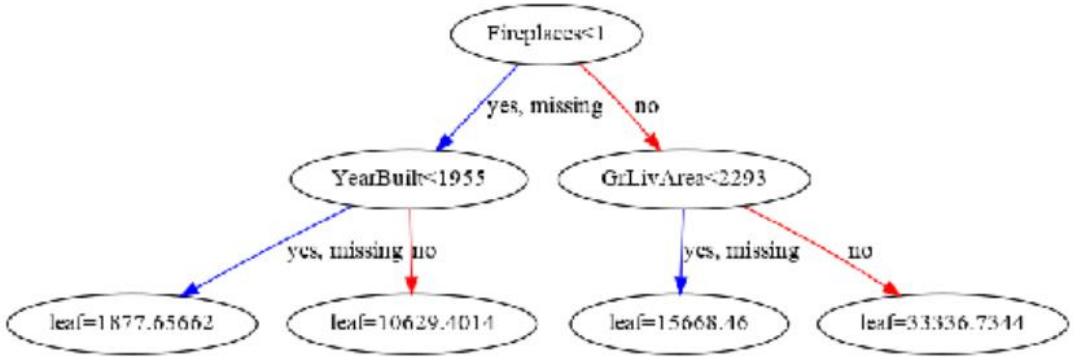
Visualizing individual XGBoost trees

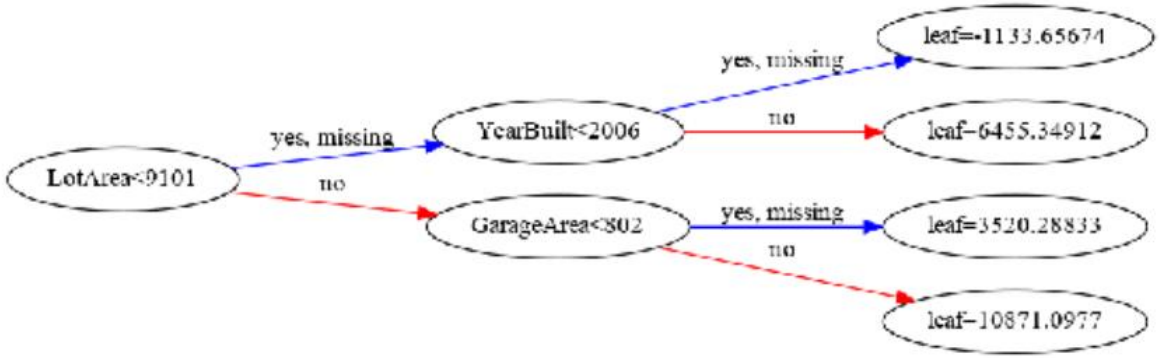
Now that you've used XGBoost to both build and evaluate regression as well as classification models, you should get a handle on how to visually explore your models. Here, you will visualize individual trees from the fully boosted model that XGBoost creates using the entire housing dataset.

XGBoost has a `plot_tree()` function that makes this type of visualization easy. Once you train a model using the XGBoost learning API, you can pass it to the `plot_tree()` function along with the number of trees you want to plot using the `num_trees` argument.

```
# 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.show()

# Plot the fifth tree
xgb.plot_tree(xg_reg, num_trees=4)
plt.show()

# Plot the last tree sideways
xgb.plot_tree(xg_reg, num_trees=9, rankdir='LR')
plt.show()
```







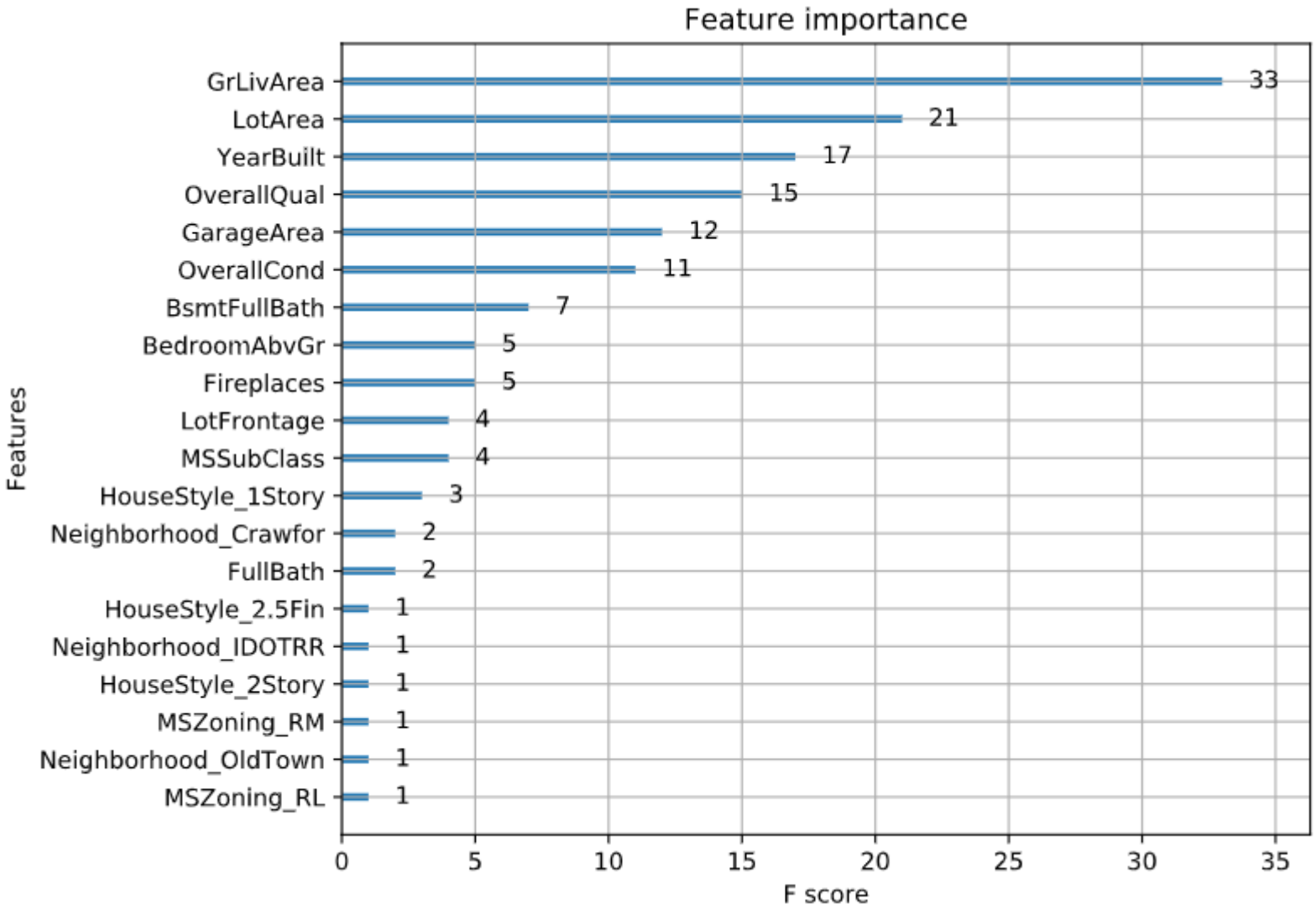
Have a look at each of the plots. They provide insight into how the model arrived at its final decisions and what splits it made to arrive at those decisions. This allows us to identify which features are the most important in determining house price. In the next exercise, you'll learn another way of visualizing feature importances.

Visualizing feature importances: What features are most important in my dataset

Another way to visualize your XGBoost models is to examine the importance of each feature column in the original dataset within the model.

One simple way of doing this involves counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear. XGBoost has a `plot_importance()` function that allows you to do exactly this, and you'll get a chance to use it in this exercise!

```
# 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}

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

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



It looks like `GrLivArea` is the most important feature.

Chapter 3. Fine-tuning your XGBoost model

This chapter will teach you how to make your XGBoost models as performant as possible. You'll learn about the variety of parameters that can be adjusted to alter the behavior of XGBoost and how to tune them efficiently so that you can supercharge the performance of your models.

Why tune your model?

Comparison between **untuned** XGBoost model and **tuned** XGBoost model – 14% reduction (improvement)

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("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
```

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("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)
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: 29812.683594
```

When is tuning your model a bad idea?

Now that you've seen the effect that tuning has on the overall performance of your XGBoost model, let's turn the question on its head and see if you can figure out when tuning your model might not be the best idea.

Given that model tuning can be time-intensive and complicated, which of the following scenarios would NOT call for careful tuning of your model?

- ○ You have lots of examples from some dataset and very many features at your disposal.
- ◉ You are very short on time before you must push an initial model to production and have little data to train your model on.
- ○ You have access to a multi-core (64 cores) server with lots of memory (200GB RAM) and no time constraints.
- ○ You must squeeze out every last bit of performance out of your xgboost model.

Answer: You cannot tune if you do not have time!

Tuning the number of boosting rounds

Let's start with parameter tuning by seeing how the number of boosting rounds (number of trees you build) impacts the out-of-sample performance of your XGBoost model. You'll use `xgb.cv()` inside a `for` loop and build one model per `num_boost_round` parameter.

Here, you'll continue working with the Ames housing dataset. The features are available in the array `X`, and the target vector is contained in `y`.

```
# 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=True, seed=123)

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

# 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  34774.191406
2                   15  32895.098307
```

As you can see, increasing the number of boosting rounds decreases the RMSE.

Automated boosting round selection using early_stopping

Now, instead of attempting to cherry pick the best possible number of boosting rounds, you can very easily have XGBoost automatically select the number of boosting rounds for you within `xgb.cv()`. This is done using a technique called **early stopping**.

Early stopping works by testing the XGBoost model after every boosting round against a hold-out dataset and stopping the creation of additional boosting rounds (thereby finishing training of the model early) if the hold-out metric (`"rmse"` in our case) does not improve for a given number of rounds. Here you will use the `early_stopping_rounds` parameter in `xgb.cv()` with a large possible number of boosting rounds (50). Bear in mind that if the holdout metric continuously improves up through when `num_boost_rounds` is reached, then early stopping does not occur.

Here, the `DMatrix` and parameter dictionary have been created for you. Your task is to use cross-validation with early stopping. Go for it!

```
# 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, nfold=3, num_boost_round=50, metrics="rmse",
as_pandas=True, seed=123, early_stopping_rounds=10)

# Print cv_results
print(cv_results)
```

	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	141871.630208	403.632626	142640.656250	705.559400
1	103057.036458	73.769561	104907.666667	111.114933
2	75975.966146	253.726099	79262.057291	563.767892
3	57420.531250	521.656754	61620.136719	1087.694282
4	44552.955729	544.170190	50437.561198	1846.446330
5	35763.946615	681.797429	43035.661458	2034.469207
6	29861.464193	769.571238	38600.880208	2169.796232
7	25994.676432	756.520565	36071.817708	2109.795430
8	23306.836588	759.238254	34383.184896	1934.546688
9	21459.769531	745.624998	33509.142578	1887.377024
10	20148.721354	749.612769	32916.807943	1850.894475
11	19215.382162	641.387014	32197.833333	1734.456784
12	18627.388021	716.257152	31770.853516	1802.154726
13	17960.694661	557.043073	31482.782552	1779.123767
14	17559.736328	631.412555	31389.990886	1892.319967
15	17205.713216	590.172372	31302.882162	1955.165902
16	16876.571940	703.631755	31234.058594	1880.705796
17	16597.662110	703.677609	31318.348308	1828.860391
18	16330.460937	607.274494	31323.634766	1775.910706
19	16005.972982	520.470911	31204.136068	1739.078273
20	15814.301432	518.604477	31089.862630	1756.021674
21	15493.405599	505.616447	31047.996094	1624.673955
22	15270.734049	502.019527	31056.916015	1668.042812
23	15086.382162	503.913199	31024.984375	1548.985605
24	14917.608724	486.206468	30983.684896	1663.130201
25	14709.589518	449.668010	30989.477214	1686.666560
26	14457.286133	376.787666	30952.113281	1613.173549
27	14185.567383	383.102234	31066.902344	1648.534310
28	13934.067057	473.465714	31095.641927	1709.224745
29	13749.644857	473.670123	31103.886719	1778.879529
30	13549.836914	454.898399	30976.085938	1744.515079
31	13413.485351	399.603618	30938.469401	1746.052597
32	13275.915690	415.408663	30931.000651	1772.468592
33	13085.878581	493.792860	30929.056640	1765.540153
34	12947.180990	517.789678	30890.629557	1786.510976
35	12846.027344	547.732372	30884.492839	1769.728719
36	12702.378906	505.522878	30833.542969	1691.002065
37	12532.243815	508.298673	30856.687500	1771.445978
38	12384.054687	536.224681	30818.016927	1782.784630
39	12198.443685	545.165136	30839.393229	1847.326103
40	12054.583659	508.841772	30776.965495	1912.781427
41	11897.036458	477.177568	30794.701823	1919.674347
42	11756.221680	502.992782	30780.955078	1906.820029
43	11618.846029	519.837502	30783.755208	1951.258863
44	11484.080078	578.428250	30776.731120	1953.446309
45	11356.553060	565.368380	30758.543620	1947.454953
46	11193.558268	552.298848	30729.972005	1985.699316
47	11071.315429	604.089960	30732.663411	1966.997822
48	10950.778320	574.862779	30712.241536	1957.751573
49	10824.865885	576.665756	30720.854167	1950.511057

Overview of XGBoost's hyperparameters

Parameters that can be tuned:
· learning rate (eta) – affects how quickly the model fits the residual error, low learning rate requires more boosting rounds.
· gamma – minimum loss reduction allowed for a split to occur.
· alpha – L1 regularization on leaf weights, larger values mean more regularization.
· lambda – L2 regularization on leaf weights, much smoother penalty than L1.
· max_depth – affects how deeply each tree is allowed to grow during any given boosting round.
· subsample – fraction of total training set (between 0 and 1) that can be used for any boosting round, low value may underfit, high value may overfit.
· colsamle_bytree – fraction of features you can select (between 0 and 1) from any boosting round, small value provides additional regularization to the model, large value may overfit a trained model.
· lambda_bias – L2 reg term on bias.
· n_estimators – number of trees or number of linear base learners.

Tuning eta (learning rate)

It's time to practice tuning other XGBoost hyperparameters in earnest and observing their effect on model performance! You'll begin by tuning the `"eta"`, also known as the learning rate.

The learning rate in XGBoost is a parameter that can range between `0` and `1`, with higher values of `"eta"` penalizing feature weights more strongly, causing much stronger regularization.

```
# 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", 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(eta_vals, best_rmse)), columns=["eta","best_rmse"]))
```

```
     eta      best_rmse
0  0.001  195736.406250
1  0.010  179932.187500
2  0.100   79759.411458
```

Tuning max_depth

In this exercise, your job is to tune `max_depth`, which is the parameter that dictates the maximum depth that each tree in a boosting round can grow to. Smaller values will lead to shallower trees, and larger values to deeper trees.

```
# Create your housing DMatrix: 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, 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(max_depths, best_rmse)),columns=["max_depth","best_rmse"]))
```

```
   max_depth     best_rmse
0          2  37957.468750
1          5  35596.599610
2         10  36065.548829
3         20  36739.578125
```

Tuning colsample_bytree

Now, it's time to tune `"colsample_bytree"`. You've already seen this if you've ever worked with scikit-learn's `RandomForestClassifier` or `RandomForestRegressor`, where it just was called `max_features`. In both `xgboost` and `sklearn`, this parameter

(although named differently) simply specifies the fraction of features to choose from at every split in a given tree. In `xgboost`, `colsample_bytree` must be specified as a float between 0 and 1.

```
# 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 = [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  48193.451172
1               0.5  36013.544922
2               0.8  35932.962891
3               1.0  35836.042969
```

There are several other individual parameters that you can tune, such as `"subsample"`, which dictates the fraction of the training data that is used during any given boosting round. Next up: Grid Search and Random Search to tune XGBoost hyperparameters more efficiently!

Review of grid search and random search

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import GridSearchCV
housing_data = pd.read_csv("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,param_grid=gbm_param_grid,
            scoring='neg_mean_squared_error', cv=4, verbose=1)
grid_mse.fit(X, y)
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:  28530.1829341
```

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
housing_data = pd.read_csv("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, param_distributions=gbm_param_grid,
                        n_iter=25, scoring='neg_mean_squared_error', cv=4, verbose=1)
randomized_mse.fit(X, y)
print("Best parameters found: ",randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))
```

```
Best parameters found: {'subsample': 0.6000000000000009,
'n_estimators': 200, 'learning_rate': 0.2000000000000001}
Lowest RMSE found: 28300.2374291
```

Grid search with XGBoost

Now that you've learned how to tune parameters individually with XGBoost, let's take your parameter tuning to the next level by using scikit-learn's `GridSearch` and `RandomizedSearch` capabilities with internal cross-validation using the `GridSearchCV` and `RandomizedSearchCV` functions. You will use these to find the best model exhaustively from a collection of possible parameter values across multiple parameters simultaneously. Let's get to work, starting with `GridSearchCV`!

```
# 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,
                        scoring='neg_mean_squared_error', cv=4, verbose=1)
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_)))
```

```
Best parameters found:  {'colsample_bytree': 0.7, 'max_depth': 5, 'n_estimators': 50}
Lowest RMSE found:  29916.562522854438
```

Random search with XGBoost

Often, `GridSearchCV` can be really time consuming, so in practice, you may want to use `RandomizedSearchCV` instead, as you will do in this exercise. The good news is you only have to make a few modifications to your `GridSearchCV` code to do `RandomizedSearchCV`. The key difference is you have to specify a `param_distributions` parameter instead of a `param_grid` parameter.

```
# 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, param_distributions=gbm_param_grid,
                                    n_iter=5, scoring='neg_mean_squared_error', cv=4, verbose=1)
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_)))
```

```
Best parameters found:  {'n_estimators': 25, 'max_depth': 6}
Lowest RMSE found:  36909.98213965752
```

Limits of grid search and random search

- Grid Search
  - Number of models you must build with every additional new parameter grows very quickly
- Random Search
  - Parameter space to explore can be massive
  - Randomly jumping throughout the space looking for a "best" result becomes a waiting game

When should you use grid search and random search?

Now that you've seen some of the drawbacks of grid search and random search, which of the following most accurately describes why both random search and grid search are non-ideal search hyperparameter tuning strategies in all scenarios?

- ○ Grid Search and Random Search both take a very long time to perform, regardless of the number of parameters you want to tune.
- ○ Grid Search and Random Search both scale exponentially in the number of hyperparameters you want to tune.
- ◉ The search space size can be massive for Grid Search in certain cases, whereas for Random Search the number of hyperparameters has a significant effect on how long it takes to run.
- ○ Grid Search and Random Search require that you have some idea of where the ideal values for hyperparameters reside.

Answer: The search space size can be massive for Grid Search in certain cases, whereas for Random Search the number of hyperparameters has a significant effect on how long it takes to run. This is why random search and grid search should not always be used.

Chapter 4. Using XGBoost in pipelines

Take your XGBoost skills to the next level by incorporating your models into two end-to-end machine learning pipelines. You'll learn how to tune the most important XGBoost hyperparameters efficiently within a pipeline, and get an introduction to some more advanced preprocessing techniques.

Review of pipelines using sklearn

Takes a list of named 2-tuples (name_string, 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 grid/randomized search and cross_val_score methods.

Scikit-learn pipeline example:

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import numpy as np
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("boston_housing.csv",names=names)

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,
scoring="neg_mean_squared_error",cv=10)
```

Note: 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 actually exist as all squares must be positive when working with real numbers.

```
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))

print("Final RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.54530686529
```

Preprocessing I: LabelEncorder and OneHotEncoder

LabelEncorder = converts a categorical column of strings into integers.

OneHotEncoder = takes the column of integers and encodes them as dummy variables.

(These cannot be done in a pipeline)

Preprocessing II: DictVectorizer

Found in scikit-learn's feature extraction submodule (`from sklearn.feature_extraction import DictVectorizer`).

Traditional used in text processing pipelines by converting lists of feature mappings into vectors.

Need to convert DataFrame into a list of dictionary entries.

Exploratory data analysis

Before diving into the nitty gritty of pipelines and preprocessing, let's do some exploratory analysis of the original, unprocessed **Ames housing dataset**. When you worked with this data in previous chapters, we preprocessed it for you so you could focus on the core XGBoost concepts. In this chapter, you'll do the preprocessing yourself.

A smaller version of this original, unprocessed dataset has been pre-loaded into a `pandas` DataFrame called `df`. The larger purpose of this exercise is to understand the kinds of transformations you will need to perform in order to be able to use XGBoost.

Your task is to explore `df` in the Shell and pick the option that is **incorrect**.

```
In [1]: df.describe()
Out[1]:
```

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	...	HalfBath	BedroomAbvGr	Fireplaces	GarageArea	SalePrice
count	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	...	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000
mean	56.897260	70.049958	10516.828082	6.099315	5.575342	...	0.382877	2.866438	0.613014	472.980137	180921.195890
std	42.300571	24.284752	9981.264932	1.382997	1.112799	...	0.502885	0.815778	0.644666	213.804841	79442.502883
min	20.000000	21.000000	1300.000000	1.000000	1.000000	...	0.000000	0.000000	0.000000	0.000000	34900.000000
25%	20.000000	59.000000	7553.500000	5.000000	5.000000	...	0.000000	2.000000	0.000000	334.500000	129975.000000
50%	50.000000	69.000000	9478.500000	6.000000	5.000000	...	0.000000	3.000000	1.000000	480.000000	163000.000000
75%	70.000000	80.000000	11601.500000	7.000000	6.000000	...	1.000000	3.000000	1.000000	576.000000	214000.000000
max	190.000000	313.000000	215245.000000	10.000000	9.000000	...	2.000000	8.000000	3.000000	1418.000000	755000.000000

```
In [2]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 21 columns):
MSSubClass      1460 non-null int64
LotFrontage     1201 non-null float64
LotArea         1460 non-null int64
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
MSZoning        1460 non-null object
PavedDrive      1460 non-null object
Neighborhood    1460 non-null object
BldgType        1460 non-null object
HouseStyle      1460 non-null object
SalePrice       1460 non-null int64
dtypes: float64(1), int64(15), object(5)
memory usage: 239.6+ KB
```

Possible Answers

- The DataFrame has 21 columns and 1460 rows.
- The mean of the `LotArea` column is `10516.828082`.
- The DataFrame has missing values.
- The `LotFrontage` column has no missing values and its entries are of type `float64`.
- The standard deviation of `SalePrice` is `79442.502883`.

Well done! The `LotFrontage` column actually does have missing values: 259, to be precise. Additionally, notice how columns such as `MSZoning`, `PavedDrive`, and `HouseStyle` are categorical. These need to be encoded numerically before you can use XGBoost. This is what you'll do in the coming exercises.

Encoding categorical columns I: LabelEncoder

Now that you've seen what will need to be done to get the housing data ready for XGBoost, let's go through the process step-by-step.

First, you will need to fill in missing values - as you saw previously, the column `LotFrontage` has many missing values. Then, you will need to encode any categorical columns in the dataset using one-hot encoding so that they are encoded numerically. You can watch **this video** from **Supervised Learning with scikit-learn** for a refresher on the idea.

The data has five categorical columns: `MSZoning`, `PavedDrive`, `Neighborhood`, `BldgType`, and `HouseStyle`. Scikit-learn has a **LabelEncoder** function that converts the values in each categorical column into integers. You'll practice using this here.

```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

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

# 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())
```

```
<script.py> output:
      MSZoning PavedDrive Neighborhood BldgType HouseStyle
    0       RL          Y      CollgCr     1Fam     2Story
    1       RL          Y      Veenker     1Fam     1Story
    2       RL          Y      CollgCr     1Fam     2Story
    3       RL          Y      Crawfor     1Fam     2Story
    4       RL          Y      NoRidge     1Fam     2Story
       MSZoning  PavedDrive  Neighborhood  BldgType  HouseStyle
    0         3           2             5         0           5
    1         3           2            24         0           2
    2         3           2             5         0           5
    3         3           2             6         0           5
    4         3           2            15         0           5
```

Encoding categorical columns II: OneHotEncoder

Okay - so you have your categorical columns encoded numerically. Can you now move onto using pipelines and XGBoost? Not yet! In the categorical columns of this dataset, there is no natural ordering between the entries. As an example: Using `LabelEncoder`, the `CollgCr` `Neighborhood` was encoded as `5`, while the `Veenker` `Neighborhood` was encoded as `24`, and `Crawfor` as `6`. Is `Veenker` "greater" than `Crawfor` and `CollgCr`? No - and allowing the model to assume this natural ordering may result in poor performance.

As a result, there is another step needed: You have to apply a one-hot encoding to create binary, or "dummy" variables. You can do this using scikit-learn's **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)
```

```
<script.py> output:
    [[0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 1.000e+00 0.000e+00 0.000e+00 6.000e+01 6.500e+01 8.450e+03
      7.000e+00 5.000e+00 2.003e+03 0.000e+00 1.710e+03 1.000e+00 0.000e+00
      2.000e+00 1.000e+00 3.000e+00 0.000e+00 5.480e+02 2.085e+05]
     [0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 2.000e+01 8.000e+01 9.600e+03
      6.000e+00 8.000e+00 1.976e+03 0.000e+00 1.262e+03 0.000e+00 1.000e+00
      2.000e+00 0.000e+00 3.000e+00 1.000e+00 4.600e+02 1.815e+05]
     [0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 1.000e+00 0.000e+00 0.000e+00 6.000e+01 6.800e+01 1.125e+04
      7.000e+00 5.000e+00 2.001e+03 1.000e+00 1.786e+03 1.000e+00 0.000e+00
      2.000e+00 1.000e+00 3.000e+00 1.000e+00 6.080e+02 2.235e+05]
     [0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 1.000e+00 0.000e+00 0.000e+00 7.000e+01 6.000e+01 9.550e+03
      7.000e+00 5.000e+00 1.915e+03 1.000e+00 1.717e+03 1.000e+00 0.000e+00
      1.000e+00 0.000e+00 3.000e+00 1.000e+00 6.420e+02 1.400e+05]
     [0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 1.000e+00 0.000e+00 0.000e+00 6.000e+01 8.400e+01 1.426e+04
      8.000e+00 5.000e+00 2.000e+03 0.000e+00 2.198e+03 1.000e+00 0.000e+00
      2.000e+00 1.000e+00 4.000e+00 1.000e+00 8.360e+02 2.500e+05]]
    (1460, 21)
    (1460, 62)
```

As you can see, after one hot encoding, which creates binary variables out of the categorical variables, there are now 62 columns.

Encoding categorical columns III: DictVectorizer

Alright, one final trick before you dive into pipelines. The two step process you just went through - `LabelEncoder` followed by `OneHotEncoder` - can be simplified by using a **DictVectorizer**.

Using a `DictVectorizer` on a DataFrame that has been converted to a dictionary allows you to get label encoding as well as one-hot encoding in one go.

Your task is to work through this strategy in this exercise!

```
# 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(sparse=False)

# 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_)
```

```
<script.py> output:
    [[3.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 2.000e+00 5.480e+02 1.710e+03 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00
      8.450e+03 6.500e+01 6.000e+01 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 5.000e+00 7.000e+00
      0.000e+00 0.000e+00 1.000e+00 0.000e+00 2.085e+05 2.003e+03]
     [3.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 1.000e+00 2.000e+00 4.600e+02 1.262e+03 0.000e+00 0.000e+00
      0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      9.600e+03 8.000e+01 2.000e+01 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 8.000e+00 6.000e+00
      0.000e+00 0.000e+00 1.000e+00 0.000e+00 1.815e+05 1.976e+03]
     [3.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 1.000e+00 2.000e+00 6.080e+02 1.786e+03 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00
      1.125e+04 6.800e+01 6.000e+01 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 5.000e+00 7.000e+00
      0.000e+00 0.000e+00 1.000e+00 1.000e+00 2.235e+05 2.001e+03]
     [3.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 1.000e+00 1.000e+00 6.420e+02 1.717e+03 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00
      9.550e+03 6.000e+01 7.000e+01 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 5.000e+00 7.000e+00
      0.000e+00 0.000e+00 1.000e+00 1.000e+00 1.400e+05 1.915e+03]
     [4.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 1.000e+00 2.000e+00 8.360e+02 2.198e+03 1.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00
      1.426e+04 8.400e+01 6.000e+01 0.000e+00 0.000e+00 0.000e+00 1.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 1.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00
      0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 5.000e+00 8.000e+00
      0.000e+00 0.000e+00 1.000e+00 0.000e+00 2.500e+05 2.000e+03]]
    {'MSSubClass': 23, 'LotFrontage': 22, 'LotArea': 21, 'OverallQual': 55, 'OverallCond': 54, 'YearBuilt': 61, 'Remodeled': 59, 'GrLivArea': 11, 'BsmtFullBath': 6, 'BsmtHalfBath': 7,
       'FullBath': 9, 'HalfBath': 12, 'BedroomAbvGr': 0, 'Fireplaces': 8, 'GarageArea': 10, 'MSZoning=RL': 27, 'PavedDrive=Y': 58, 'Neighborhood=CollgCr': 34, 'BldgType=1Fam': 1, 'HouseStyle
       =2Story': 18, 'SalePrice': 60, 'Neighborhood=Veenker': 53, 'HouseStyle=1Story': 15, 'Neighborhood=Crawfor': 35, 'Neighborhood=NoRidge': 44, 'Neighborhood=Mitchel': 40, 'HouseStyle=1
       .5Fin': 13, 'Neighborhood=Somerst': 50, 'Neighborhood=NWAmes': 43, 'MSZoning=RM': 28, 'Neighborhood=OldTown': 46, 'Neighborhood=BrkSide': 32, 'BldgType=2fmCon': 2, 'HouseStyle=1.5Unf':
       14, 'Neighborhood=Sawyer': 48, 'Neighborhood=NridgHt': 45, 'Neighborhood=NAmes': 41, 'BldgType=Duplex': 3, 'Neighborhood=SawyerW': 49, 'PavedDrive=N': 56, 'Neighborhood=IDOTRR': 38,
       'Neighborhood=MeadowV': 39, 'BldgType=TwnhsE': 5, 'MSZoning=C (all)': 24, 'Neighborhood=Edwards': 36, 'PavedDrive=P': 57, 'Neighborhood=Timber': 52, 'HouseStyle=SFoyer': 19, 'MSZoning
       =FV': 25, 'Neighborhood=Gilbert': 37, 'HouseStyle=SLvl': 20, 'BldgType=Twnhs': 4, 'Neighborhood=StoneBr': 51, 'HouseStyle=2.5Unf': 17, 'Neighborhood=ClearCr': 33, 'Neighborhood=NPkVill'
       : 42, 'HouseStyle=2.5Fin': 16, 'Neighborhood=Blmngtn': 29, 'Neighborhood=BrDale': 31, 'Neighborhood=SWISU': 47, 'MSZoning=RH': 26, 'Neighborhood=Blueste': 30}
```

Besides simplifying the process into one step, `DictVectorizer` has useful attributes such as `vocabulary_` which maps the names of the features to their indices. With the data preprocessed, it's time to move onto pipelines!

Preprocessing within a pipeline

Now that you've seen what steps need to be taken individually to properly process the Ames housing data, let's use the much cleaner and more succinct `DictVectorizer` approach and put it alongside an `XGBoostRegressor` inside of a scikit-learn pipeline.

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

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

# 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)
```

```
<script.py> output:
    [03:36:50] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
```

Well done! It's now time to see what it takes to use XGBoost within pipelines.

Incorporating XGBoost into pipelines

Scikit-learn pipeline example with XGBoost

```
import pandas as pd
import xgboost as xgb
import numpy as np
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("boston_housing.csv",names=names)
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,
                         scoring="neg_mean_squared_error",cv=10)
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print("Final XGB RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.02719593323
```

This is lower than RandomForestRegressor RMSE 4.5453
(see example in Review of pipelines using sklearn)

Additional components introduced for pipelines:

sklearn_pandas: (bridge the gap for both scikit-learn and pandas to work seamlessly together)

```
from sklearn_pandas import DataFrameMapper
```

DataFrameMapper = interoperability between pandas and scikit-learn

```
from sklearn_pandas import CategoricalImputer
```

CategoricalImputer = allow for imputation of categorical variables without the need for conversion to integers

sklearn.preprocessing:

Imputer = native imptation of missing numerical columns in scikit-learn

sklearn.pipeline:

```
from sklearn.pipeline import FeatureUnion
```

FeatureUnion = combine multiple pipelines of features into a single pipeline of features, for example, a set of preprocessing steps on categorical features and another set of preprocessing steps on numerical features of a dataset

Cross-validating your XGBoost model

In this exercise, you'll go one step further by using the pipeline you've created to preprocess **and** cross-validate your model.

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

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

# 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))))
```

```
<script.py> output:
    [03:58:11] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:12] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:12] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:12] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:13] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:13] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:13] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:14] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:14] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    [03:58:14] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
    10-fold RMSE:  29867.603720688923
```

Kidney disease case study I: Categorical Imputer

You'll now continue your exploration of using pipelines with a dataset that requires significantly more wrangling. The **chronic kidney disease dataset** contains both categorical and numeric features, but contains lots of missing values. The goal here is to predict who has chronic kidney disease given various blood indicators as features.

As Sergey mentioned in the video, you'll be introduced to a new library, `sklearn_pandas`, that allows you to chain many more processing steps inside of a pipeline than are currently supported in scikit-learn. Specifically, you'll be able to impute missing categorical values directly using the `Categorical_Imputer()` class in `sklearn_pandas`, and the `DataFrameMapper()` class to apply any arbitrary sklearn-compatible transformer on DataFrame columns, where the resulting output can be either a NumPy array or DataFrame.

We've also created a transformer called a `Dictifier` that encapsulates converting a DataFrame using `.to_dict("records")` without you having to do it explicitly (and so that it works in a pipeline). Finally, we've also provided the list of feature names in `kidney_feature_names`, the target name in `kidney_target_name`, the features in `X`, and the target in `y`.

In this exercise, your task is to apply the `CategoricalImputer` to impute all of the categorical columns in the dataset. You can refer to how the numeric imputation mapper was created as a template. Notice the keyword arguments `input_df=True` and `df_out=True`? This is so that you can work with DataFrames instead of arrays. By default, the transformers are passed a `numpy` array of the selected columns as input, and as a result, the output of the DataFrame mapper is also an array. Scikit-learn transformers have historically been designed to work with `numpy` arrays, not `pandas` DataFrames, even though their basic indexing interfaces are similar.

```
# 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
                                               )
```

```
<script.py> output:
    age        9
    bp        12
    sg        47
    al        46
    su        49
    bgr       44
    bu        19
    sc        17
    sod       87
    pot       88
    hemo      52
    pcv       71
    wc       106
    rc       131
    rbc      152
    pc        65
    pcc        4
    ba         4
    htn        2
    dm         2
    cad        2
    appet      1
    pe         1
    ane        1
    dtype: int64
```

Kidney disease case study II: Feature Union

Having separately imputed numeric as well as categorical columns, your task is now to use scikit-learn's **FeatureUnion** to concatenate their results, which are contained in two separate transformer objects - `numeric_imputation_mapper`, and `categorical_imputation_mapper`, respectively.

You may have already encountered `FeatureUnion` in **Machine Learning with the Experts: School Budgets**. Just like with pipelines, you have to pass it a list of `(string, transformer)` tuples, where the first half of each tuple is the name of the transformer.

```
# 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)
                                         ])
```

Kidney disease case study III: Full pipeline

It's time to piece together all of the transforms along with an `XGBClassifier` to build the full pipeline!

Besides the `numeric_categorical_union` that you created in the previous exercise, there are two other transforms needed: the `Dictifier()` transform which we created for you, and the `DictVectorizer()`.

After creating the pipeline, your task is to cross-validate it to see how well it performs.

```
# 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, kidney_data, y, scoring="roc_auc", cv=3)

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

```
<script.py> output:
    3-fold AUC:  0.998637406769937
```

Tuning XGBoost hyperparameters

Automated hyperparameter tuning in a pipeline, for an XGBoost model

```
import pandas as pd
   ...: import xgboost as xgb
   ...: import numpy as np
   ...: 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("boston_housing.csv",names=names)
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,
   ...: param_distributions=gbm_param_grid, n_iter=10,
   ...: scoring='neg_mean_squared_error', cv=4)
randomized_neg_mse.fit(X, y)
```

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

```
Best rmse: 3.9966784203040677
```

```
print("Best model: ", randomized_neg_mse.best_estimator_)
```

```
Best model:  Pipeline(steps=[('st_scaler', StandardScaler(copy=True,
with_mean=True, with_std=True)),
('xgb_model', XGBRegressor(base_score=0.5, colsample_bylevel=1,
       colsample_bytree=0.95000000000000029, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=8, min_child_weight=1, missing=None,
       n_estimators=100, nthread=-1, objective='reg:linear', reg_alpha=0,
       reg_lambda=1, scale_pos_weight=1, seed=0, silent=True,
       subsample=0.90000000000000013))])
```

Bringing it all together

Alright, it's time to bring together everything you've learned so far! In this final exercise of the course, you will combine your work from the previous exercises into one end-to-end XGBoost pipeline to really cement your understanding of preprocessing and pipelines in XGBoost.

Your work from the previous 3 exercises, where you preprocessed the data and set up your pipeline, has been pre-loaded. Your job is to perform a randomized search and identify the best hyperparameters.

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

# Perform RandomizedSearchCV
randomized_roc_auc = RandomizedSearchCV(estimator=pipeline,
                                        param_distributions=gbm_param_grid,
                                        n_iter=2, scoring='roc_auc', cv=2, verbose=1)

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

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

```
<script.py> output:
    Fitting 2 folds for each of 2 candidates, totalling 4 fits
    0.9965333333333334
    Pipeline(memory=None,
             steps=[('featureunion',
                     FeatureUnion(n_jobs=None,
                                  transformer_list=[('num_mapper',
                                                     DataFrameMapper(default=False,
                                                                     df_out=True,
                                                                     features=[(['age'],
                                                                                Imputer(axis=0,
                                                                                        copy=True,
                                                                                        missing_values='NaN',
                                                                                        strategy='median',
                                                                                        verbose=0)),
                                                                               (['bp'],
                                                                                Imputer(axis=0,
                                                                                        copy=True,
                                                                                        missing_values='NaN',
                                                                                        strategy='median',
                                                                                        verbose=0)),
                                                                               (['sg'],
                                                                                Imputer(axis=0,
                                                                                        copy=...
                     XGBClassifier(base_score=0.5, booster='gbtree',
                                   colsample_bylevel=1, colsample_bynode=1,
                                   colsample_bytree=1, gamma=0,
                                   learning_rate=0.9500000000000001,
                                   max_delta_step=0, max_depth=4,
                                   min_child_weight=1, missing=None,
                                   n_estimators=100, n_jobs=1, nthread=None,
                                   objective='binary:logistic', random_state=0,
                                   reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
                                   seed=None, silent=None, subsample=1,
                                   verbosity=1))],
             verbose=False)
```

This type of pipelining is very common in real-world data science and you're well on your way towards mastering it.

Final Thoughts

What you have learnt in this course:

- Using XGBoost for classification tasks
- Using XGBoost for regression tasks
- Tuning XGBoost's most important hyperparameters
- Incorporating XGBoost into sklearn pipelines

Path forward, to explore:

- Using XGBoost for ranking and recommendation problems (Netflix/Amazon problem)
- Using more sophisticated hyperparameter tuning strategies for tuning XGBoost models (Bayesian Optimization)
- Using XBGoost as part of an ensemble of other models for regression/classification

---

Happy learning!