2.3. First ML Model

My first Machine Learning Model using scikit-learn's DecisionTreeRegressor

In [1]:
```
import pandas as pd

# Path of the file to read
iowa_file_path = 'input/home-data-for-ml-course/train.csv'

home_data = pd.read_csv(iowa_file_path)
```

In [2]:
```
# print the list of columns in the dataset to find the name of the prediction target
home_data.columns
```

Out[2]:
```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

Step 1: Specify Prediction Target

(You'll need to print a list of the columns to find the name of the column you need.)

You can pull out a variable with `dot-notation`. This single column is stored in a `Series`, which is broadly like a DataFrame with only a single column of data.

We'll use the dot notation to select the column we want to predict, which is called the `prediction target`. By convention, the prediction target is called `y`. So the code we need to save the sales prices in the home data is

In [3]:
```
y = home_data.SalePrice
print(y.head())
```

```
0    208500
1    181500
2    223500
3    140000
4    250000
Name: SalePrice, dtype: int64
```

Step 2: Choosing "Features"

The columns that are inputted into our model (and later used to make predictions) are called "features." In our case, those would be the columns used to determine the home price. Sometimes, you will use all columns except the target as features. Other times you'll be better off with fewer features.

For now, we'll build a model with only a few features. Later on you'll see how to iterate and compare models built with different features.

We select multiple features by providing a list of column names inside brackets. Each item in that list should be a string (with quotes).

In [4]:
```
# Create the list of features below
feature_names = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
```

By convention, this data is called `X`.

In [5]:
```
# Select data corresponding to features in feature_names
X = home_data[feature_names]
```

Let's quickly review the data we'll be using to predict house prices using the `describe` method and the `head` method, which shows the top few rows.

```
In [6]: X.describe()
```

Out[6]:

	LotArea	YearBuilt	1stFlrSF	2ndFlrSF	FullBath	BedroomAbvGr	TotRmsAbvGrd
count	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000
mean	10516.828082	1971.267808	1162.626712	346.992466	1.565068	2.866438	6.517808
std	9981.264932	30.202904	386.587738	436.528436	0.550916	0.815778	1.625393
min	1300.000000	1872.000000	334.000000	0.000000	0.000000	0.000000	2.000000
25%	7553.500000	1954.000000	882.000000	0.000000	1.000000	2.000000	5.000000
50%	9478.500000	1973.000000	1087.000000	0.000000	2.000000	3.000000	6.000000
75%	11601.500000	2000.000000	1391.250000	728.000000	2.000000	3.000000	7.000000
max	215245.000000	2010.000000	4692.000000	2065.000000	3.000000	8.000000	14.000000

```
In [7]: X.head()
```

Out[7]:

	LotArea	YearBuilt	1stFlrSF	2ndFlrSF	FullBath	BedroomAbvGr	TotRmsAbvGrd
0	8450	2003	856	854	2	3	8
1	9600	1976	1262	0	2	3	6
2	11250	2001	920	866	2	3	6
3	9550	1915	961	756	1	3	7
4	14260	2000	1145	1053	2	4	9

Step 3: Build Your Model

You will use the `scikit-learn` library to create your models. When coding, this library is written as `sklearn`, as you will see in the sample code. Scikit-learn is easily the most popular library for modeling the types of data typically stored in DataFrames.

The steps to building and using a model are:

`Define` : What type of model will it be? A decision tree? Some other type of model? Some other parameters of the model type are specified too.

`Fit` : Capture patterns from provided data (`X` and `y`). This is the heart of modeling.

`Predict` : Just what it sounds like.

`Evaluate` : Determine how accurate the model's predictions are.

Create a `DecisionTreeRegressor` and save it iowa_model. Ensure you've done the relevant import from sklearn to run this command.

```
In [8]: from sklearn.tree import DecisionTreeRegressor

        # Define model. Specify a number for random_state to ensure same results each run
        iowa_model = DecisionTreeRegressor(random_state=1)

        # Fit model
        iowa_model.fit(X, y)
```

```
Out[8]: DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                              max_features=None, max_leaf_nodes=None,
                              min_impurity_decrease=0.0, min_impurity_split=None,
                              min_samples_leaf=1, min_samples_split=2,
                              min_weight_fraction_leaf=0.0, presort='deprecated',
                              random_state=1, splitter='best')
```

Many machine learning models allow some randomness in model training. Specifying a number for random_state ensures you get the same results in each run. This is considered a good practice. You use any number, and model quality won't depend meaningfully on exactly what value you choose.

We now have a fitted model that we can use to make predictions.

In practice, you'll want to make predictions for new houses coming on the market rather than the houses we already have prices for. But we'll make predictions for the first few rows of the training data to see how the predict function works.

Step 4: Make Predictions

Make predictions with the model's `predict` command using `X` as the data. Save the results to a variable called `predictions` .

In [9]:
```
predictions = iowa_model.predict(X)
print(predictions)
```

```
[208500. 181500. 223500. ... 266500. 142125. 147500.]
```

Think About Your Results

Use the `head` method to compare the top few predictions to the actual home values (in `y`) for those same homes. Anything surprising?

In [10]:
```
print("Making predictions for the following 5 homes:")
print(y.head())
print("The predictions are")
print(iowa_model.predict(X.head()))
```

```
Making predictions for the following 5 homes:
0    208500
1    181500
2    223500
3    140000
4    250000
Name: SalePrice, dtype: int64
The predictions are
[208500. 181500. 223500. 140000. 250000.]
```

2.4. Model Validation

What is Model Validation

You'll want to evaluate almost every model you ever build. In most (though not all) applications, the relevant measure of model quality is predictive accuracy. In other words, will the model's predictions be close to what actually happens.

Many people make a huge mistake when measuring predictive accuracy. They make predictions with their training data and compare those predictions to the target values in the training data. You'll see the problem with this approach and how to solve it in a moment, but let's think about how we'd do this first.

You'd first need to summarize the model quality into an understandable way. If you compare predicted and actual home values for 10,000 houses, you'll likely find mix of good and bad predictions. Looking through a list of 10,000 predicted and actual values would be pointless. We need to summarize this into a single metric.

There are many metrics for summarizing model quality, but we'll start with one called `Mean Absolute Error` (also called `MAE`). Let's break down this metric starting with the last word, error.

The prediction error for each house is:

```
math
error=actual−predicted
```

So, if a house cost USD150,000 and you predicted it would cost USD100,000 the error is USD50,000.

With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors.

In [1]:
```python
import pandas as pd
from sklearn.tree import DecisionTreeRegressor

# Path of the file to read
iowa_file_path = 'input/home-data-for-ml-course/train.csv'

home_data = pd.read_csv(iowa_file_path)
y = home_data.SalePrice
feature_columns = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = home_data[feature_columns]

# Specify Model
iowa_model = DecisionTreeRegressor()
# Fit Model
iowa_model.fit(X, y)

print("First in-sample predictions:", iowa_model.predict(X.head()))
print("Actual target values for those homes:", y.head().tolist())
```

```
First in-sample predictions: [208500. 181500. 223500. 140000. 250000.]
Actual target values for those homes: [208500, 181500, 223500, 140000, 250000]
```

The scikit-learn library has a function **train_test_split** to break up the data into two pieces. We'll use some of that data as training data to fit the model, and we'll use the other data as validation data to calculate **mean_absolute_error**.

Here is the code:

Step 1: Split Your Data

Use the `train_test_split` function to split up your data.

Give it the argument `random_state=1` so the `check` functions know what to expect when verifying your code.

Recall, your features are loaded in the DataFrame **X** and your target is loaded in **y**.

In [2]:
```python
# Import the train_test_split function
from sklearn.model_selection import train_test_split

train_X, val_X, train_y, val_y = train_test_split(X, y, random_state=1)
```

Step 2: Build Model

Create a `DecisionTreeRegressor` model and fit it to the relevant data. Set `random_state` to 1 again when creating the model.

In [3]:
```
# Define model
iowa_model = DecisionTreeRegressor(random_state=1)

# Fit model with the training data.
iowa_model.fit(train_X, train_y)
```

Out[3]:
```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=1, splitter='best')
```

Step 3: Get Predicted Sale Prices with ALL Validation data

In [4]:
```
val_predictions = iowa_model.predict(val_X)
```

In [5]:
```
# print the top few validation predictions
print(iowa_model.predict(val_X.head()))
# print the top few actual prices from validation data
print(val_y.head())
```

```
[186500. 184000. 130000.  92000. 164500.]
258     231500
267     179500
288     122000
649      84500
1233    142000
Name: SalePrice, dtype: int64
```

What do you notice that is different from what you saw with in-sample predictions (which are printed after the top code cell in this page).

Do you remember why validation predictions differ from in-sample (or training) predictions? This is an important idea from the last lesson.

Step 4: Calculate the Mean Absolute Error in Validation Data

In [6]:
```
from sklearn.metrics import mean_absolute_error
val_mae = (mean_absolute_error(val_y, val_predictions))
print(val_mae)
```

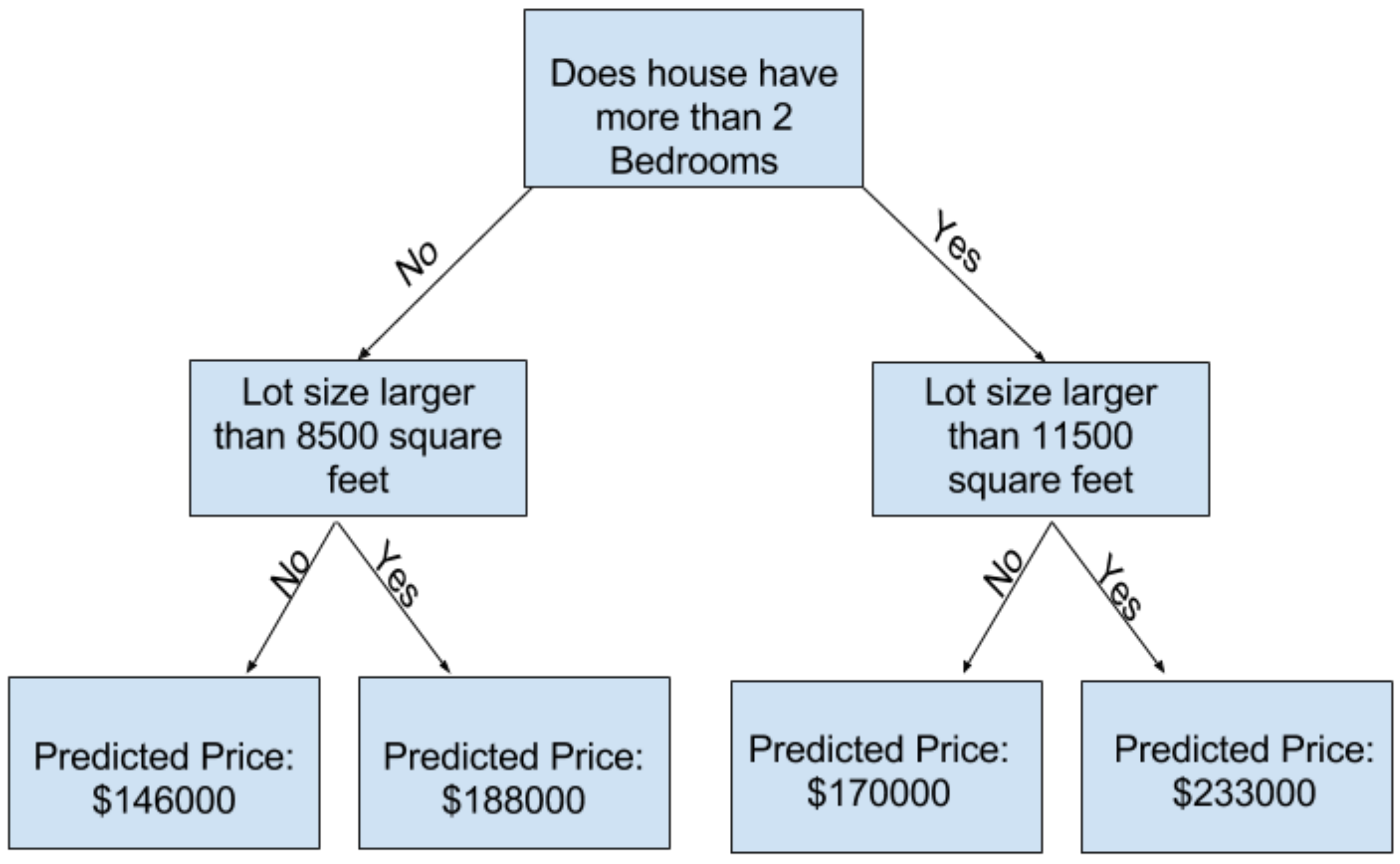
```
29652.931506849316
```

2.5. Underfitting and Overfitting

Experimenting With Different Models

Now that you have a reliable way to measure model accuracy, you can experiment with alternative models and see which gives the best predictions. But what alternatives do you have for models?

You can see in scikit-learn's documentation (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html) that the decision tree model has many options (more than you'll want or need for a long time). The most important options determine the tree's depth. Recall from the first lesson in this micro-course (https://www.kaggle.com/dansbecker/how-models-work) that a tree's depth is a measure of how many splits it makes before coming to a prediction. This is a relatively shallow tree


Does house have more than 2 Bedrooms
No
Yes
Lot size larger than 8500 square feet
Lot size larger than 11500 square feet
No
Yes
No
Yes
Predicted Price: $146000
Predicted Price: $188000
Predicted Price: $170000
Predicted Price: $233000

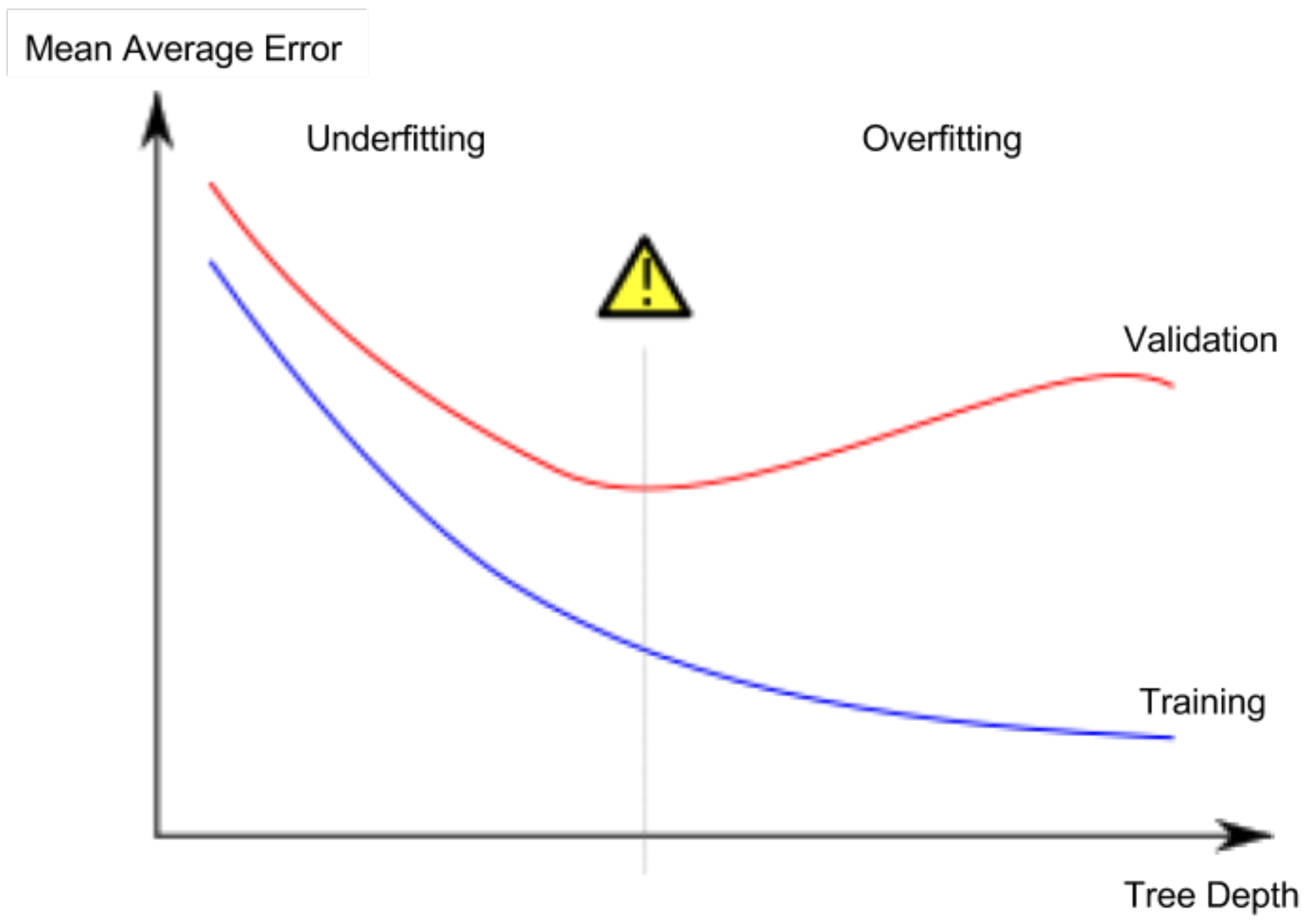
In practice, it's not uncommon for a tree to have 10 splits between the top level (all houses) and a leaf. As the tree gets deeper, the dataset gets sliced up into leaves with fewer houses. If a tree only had 1 split, it divides the data into 2 groups. If each group is split again, we would get 4 groups of houses. Splitting each of those again would create 8 groups. If we keep doubling the number of groups by adding more splits at each level, we'll have 2^10 groups of houses by the time we get to the 10th level. That's 1024 leaves.

When we divide the houses amongst many leaves, we also have fewer houses in each leaf. Leaves with very few houses will make predictions that are quite close to those homes' actual values, but they may make very unreliable predictions for new data (because each prediction is based on only a few houses).

This is a phenomenon called `overfitting`, where a model matches the training data almost perfectly, but does poorly in validation and other new data. On the flip side, if we make our tree very shallow, it doesn't divide up the houses into very distinct groups.

At an extreme, if a tree divides houses into only 2 or 4, each group still has a wide variety of houses. Resulting predictions may be far off for most houses, even in the training data (and it will be bad in validation too for the same reason). When a model fails to capture important distinctions and patterns in the data, so it performs poorly even in training data, that is called `underfitting`.

Since we care about accuracy on new data, which we estimate from our validation data, we want to find the sweet spot between underfitting and overfitting. Visually, we want the low point of the (red) validation curve in



Recap

You've built your first model, and now it's time to optimize the size of the tree to make better predictions. Run this cell to set up your coding environment where the previous step left off.

In [1]:
```
import pandas as pd
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor


# Path of the file to read
iowa_file_path = 'input/home-data-for-ml-course/train.csv'

home_data = pd.read_csv(iowa_file_path)
# Create target object and call it y
y = home_data.SalePrice
# Create X
features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = home_data[features]

# Split into validation and training data
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state=1)

# Specify Model
iowa_model = DecisionTreeRegressor(random_state=1)
# Fit Model
iowa_model.fit(train_X, train_y)

# Make validation predictions and calculate mean absolute error
val_predictions = iowa_model.predict(val_X)
val_mae = mean_absolute_error(val_predictions, val_y)
print("Validation MAE: {:,.0f}".format(val_mae))
```

```
Validation MAE: 29,653
```

Example

There are a few alternatives for controlling the tree depth, and many allow for some routes through the tree to have greater depth than other routes. But the `max_leaf_nodes` argument provides a very sensible way to control overfitting vs underfitting. The more leaves we allow the model to make, the more we move from the underfitting area in the above graph to the overfitting area.

We can use a utility function to help compare MAE scores from different values for `max_leaf_nodes` :

In [2]:
```
def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)
```

Step 1: Compare Different Tree Sizes

Write a loop that tries the following values for *max_leaf_nodes* from a set of possible values.

Call the *get_mae* function on each value of max_leaf_nodes. Store the output in some way that allows you to select the value of `max_leaf_nodes` that gives the most accurate model on your data.

In [3]:
```
candidate_max_leaf_nodes = [5, 25, 50, 100, 250, 500]
```

In [4]:
```
# Write loop to find the ideal tree size from candidate_max_leaf_nodes
# Take a short solution with a dict comprehension:
scores = {leaf_size: get_mae(leaf_size, train_X, val_X, train_y, val_y) for leaf_size in candidate_max_leaf_nodes}

# Store the best value of max_leaf_nodes (it will be either 5, 25, 50, 100, 250 or 500)
best_tree_size = min(scores, key=scores.get)
print("best_tree_size = %d \n" %(best_tree_size))


# Or with an explicit loop:
for max_leaf_nodes in candidate_max_leaf_nodes:
    my_mae = get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y)
    print("Max leaf nodes: %d  \t\t Mean Absolute Error:  %d" %(max_leaf_nodes, my_mae))

# Store the best value of max_leaf_nodes (it will be either 5, 25, 50, 100, 250 or 500)
# best_tree_size = 100
```

```
best_tree_size = 100

Max leaf nodes: 5                Mean Absolute Error:  35044
Max leaf nodes: 25               Mean Absolute Error:  29016
Max leaf nodes: 50               Mean Absolute Error:  27405
Max leaf nodes: 100              Mean Absolute Error:  27282
Max leaf nodes: 250              Mean Absolute Error:  27893
Max leaf nodes: 500              Mean Absolute Error:  29454
```

Conclusion

Here's the takeaway: Models can suffer from either:

`Overfitting` : capturing spurious patterns that won't recur in the future, leading to less accurate predictions, or

`Underfitting` : failing to capture relevant patterns, again leading to less accurate predictions.

We use `validation` data, which isn't used in model training, to measure a candidate model's accuracy. This lets us try many candidate models and keep the best one.

Step 2: Fit Model Using All Data

You know the best tree size. If you were going to deploy this model in practice, you would make it even more accurate by using all of the data and keeping that tree size. That is, you don't need to hold out the validation data now that you've made all your modeling decisions.

In [5]:
```
# Fill in argument to make optimal size and uncomment
final_model = DecisionTreeRegressor(max_leaf_nodes=best_tree_size, random_state=1)

# fit the final model and uncomment the next two lines
final_model.fit(X, y)
```

Out[5]:
```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                      max_features=None, max_leaf_nodes=100,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=1, splitter='best')
```

2.6. Random Forests

Decision trees leave you with a difficult decision. A deep tree with lots of leaves will overfit because each prediction is coming from historical data from only the few houses at its leaf. But a shallow tree with few leaves will perform poorly because it fails to capture as many distinctions in the raw data.

Even today's most sophisticated modeling techniques face this tension between underfitting and overfitting. But, many models have clever ideas that can lead to better performance. We'll look at the **random forest** as an example.

The random forest uses many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better predictive accuracy than a single decision tree and it works well with default parameters. If you keep modeling, you can learn more models with even better performance, but many of those are sensitive to getting the right parameters.

Racap

In [1]:
```
import pandas as pd
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor


# Path of the file to read
iowa_file_path = 'input/home-data-for-ml-course/train.csv'

home_data = pd.read_csv(iowa_file_path)
# Create target object and call it y
y = home_data.SalePrice
# Create X
features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = home_data[features]

# Split into validation and training data
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state=1)

# Specify Model
iowa_model = DecisionTreeRegressor(random_state=1)
# Fit Model
iowa_model.fit(train_X, train_y)

# Make validation predictions and calculate mean absolute error
val_predictions = iowa_model.predict(val_X)
val_mae = mean_absolute_error(val_predictions, val_y)
print("Validation MAE when not specifying max_leaf_nodes: {:,.0f}".format(val_mae))

# Using best value for max_leaf_nodes
iowa_model = DecisionTreeRegressor(max_leaf_nodes=100, random_state=1)
iowa_model.fit(train_X, train_y)
val_predictions = iowa_model.predict(val_X)
val_mae = mean_absolute_error(val_predictions, val_y)
print("Validation MAE for best value of max_leaf_nodes: {:,.0f}".format(val_mae))
```

```
Validation MAE when not specifying max_leaf_nodes: 29,653
Validation MAE for best value of max_leaf_nodes: 27,283
```

Step 1: Use a Random Forest

In [2]:
```
from sklearn.ensemble import RandomForestRegressor

# Define the model. Set random_state to 1
rf_model = RandomForestRegressor(random_state=1)

# fit your model
rf_model.fit(train_X, train_y)

# Calculate the mean absolute error of your Random Forest model on the validation data
rf_val_mae = mean_absolute_error(rf_model.predict(val_X), val_y)

print("Validation MAE for Random Forest Model: {}".format(rf_val_mae))
```

```
Validation MAE for Random Forest Model: 21857.15912981083
```

2.7. Exercise: Machine Learning Competitions

In this exercise, you will create and submit predictions for a Kaggle competition. You can then improve your model (e.g. by adding features) to improve and see how you stack up to others taking this micro-course.

The steps in this notebook are:

Build a Random Forest model with all of your data (X and y) Read in the "test" data, which doesn't include values for the target. Predict home values in the test data with your Random Forest model. Submit those predictions to the competition and see your score. Optionally, come back to see if you can improve your model by adding features or changing your model. Then you can resubmit to see how that stacks up on the competition leaderboard.

Recap ¶

In [1]:
```python
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

iowa_file_path = 'input/home-data-for-ml-course/train.csv'

home_data = pd.read_csv(iowa_file_path)
# Create target object and call it y
y = home_data.SalePrice
# Create X
features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = home_data[features]

# Split into validation and training data
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state=1)

# Specify Model
iowa_model = DecisionTreeRegressor(random_state=1)
# Fit Model
iowa_model.fit(train_X, train_y)

# Make validation predictions and calculate mean absolute error
val_predictions = iowa_model.predict(val_X)
val_mae = mean_absolute_error(val_predictions, val_y)
print("Validation MAE when not specifying max_leaf_nodes: {:,.0f}".format(val_mae))

# Using best value for max_leaf_nodes
iowa_model = DecisionTreeRegressor(max_leaf_nodes=100, random_state=1)
iowa_model.fit(train_X, train_y)
val_predictions = iowa_model.predict(val_X)
val_mae = mean_absolute_error(val_predictions, val_y)
print("Validation MAE for best value of max_leaf_nodes: {:,.0f}".format(val_mae))

# Define the model. Set random_state to 1
rf_model = RandomForestRegressor(random_state=1)
rf_model.fit(train_X, train_y)
rf_val_predictions = rf_model.predict(val_X)
rf_val_mae = mean_absolute_error(rf_val_predictions, val_y)

print("Validation MAE for Random Forest Model: {:,.0f}".format(rf_val_mae))
```

```
Validation MAE when not specifying max_leaf_nodes: 29,653
Validation MAE for best value of max_leaf_nodes: 27,283
Validation MAE for Random Forest Model: 21,857
```

Build a Random Forest model and train it on all of X and y.

In [2]:
```python
# To improve accuracy, create a new Random Forest model which you will train on all training data
rf_model_on_full_data = RandomForestRegressor(random_state=1)

# fit rf_model_on_full_data on all data from the training data
rf_model_on_full_data.fit(X, y)
```

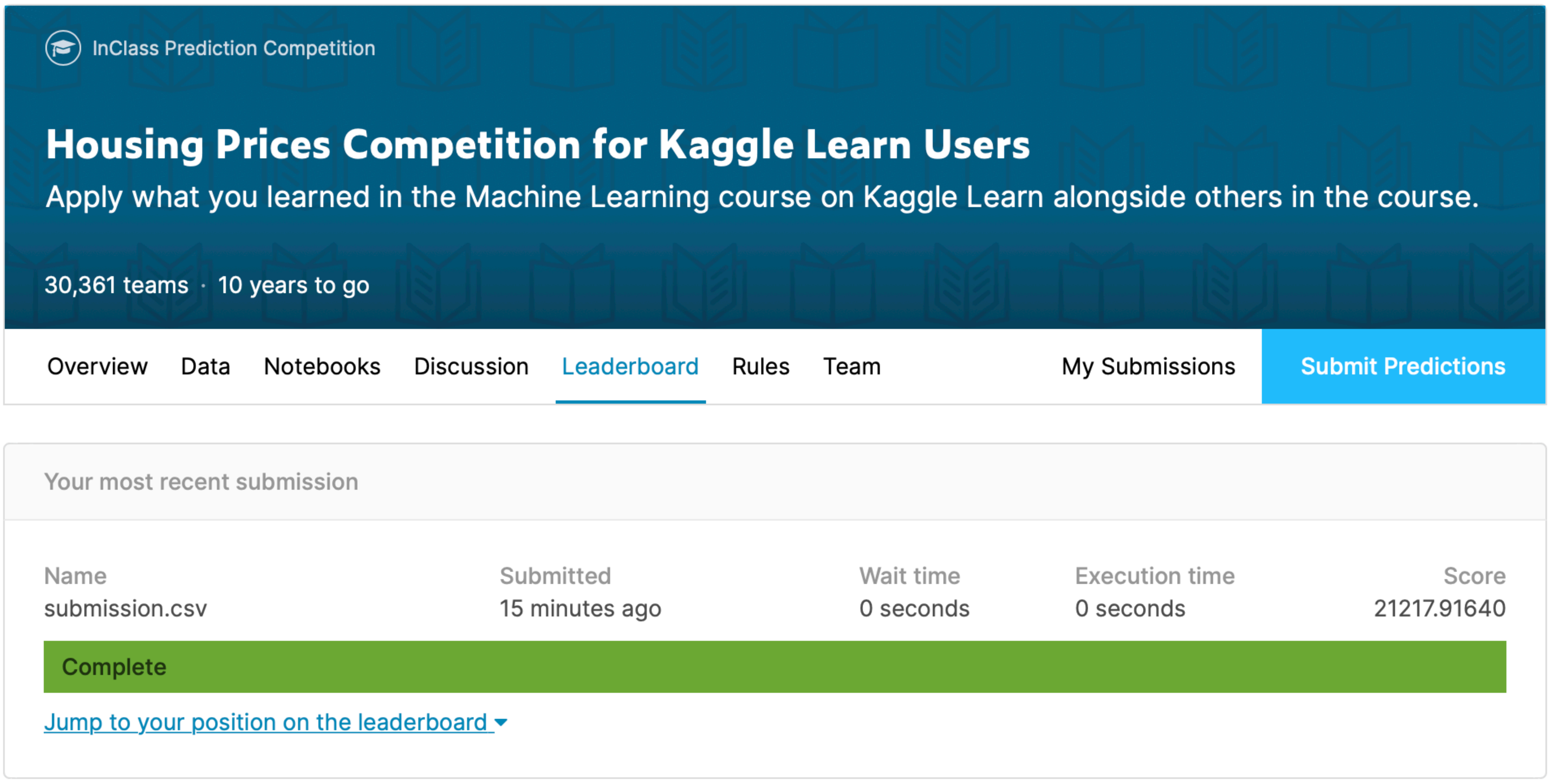
Out[2]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=1, verbose=0, warm_start=False)
```

Make Predictions

Read the file of "test" data. And apply your model to make predictions

In [3]:
```
# path to file you will use for predictions
test_data_path = 'input/home-data-for-ml-course/test.csv'

# read test data file using pandas
test_data = pd.read_csv(test_data_path)

# create test_X which comes from test_data but includes only the columns you used for prediction.
# The list of columns is stored in a variable called features
test_X = test_data[features]

# make predictions which we will submit.
test_preds = rf_model_on_full_data.predict(test_X)

# The lines below shows how to save predictions in format used for competition scoring
# Just uncomment them.

output = pd.DataFrame({'Id': test_data.Id,
                       'SalePrice': test_preds})
output.to_csv('submission.csv', index=False)
```

My scores (MAE): 21217.91640



Housing Prices Competition for Kaggle Learn Users

Apply what you learned in the Machine Learning course on Kaggle Learn alongside others in the course.

30,361 teams · 10 years to go

Your most recent submission

Name	Submitted	Wait time	Execution time	Score
submission.csv	15 minutes ago	0 seconds	0 seconds	21217.91640

Complete

Continuing Your Progress

There are many ways to improve your model, and **experimenting is a great way to learn at this point.**

The best way to improve your model is to add features. Look at the list of columns and think about what might affect home prices. Some features will cause errors because of issues like missing values or non-numeric data types.

The **Intermediate Machine Learning (https://www.kaggle.com/learn/intermediate-machine-learning)** micro-course will teach you how to handle these types of features. You will also learn to use **xgboost**, a technique giving even better accuracy than Random Forest.

Other Micro-Courses

The **Pandas (https://kaggle.com/Learn/Pandas)** micro-course will give you the data manipulation skills to quickly go from conceptual idea to implementation in your data science projects.

You are also ready for the **Deep Learning (https://kaggle.com/Learn/Deep-Learning)** micro-course, where you will build models with better-than-human level performance at computer vision tasks.