In [1]:

```
import pandas as pd
url = (
    "https://archive.ics.uci.edu/ml/machine-learning-databases"
    "/abalone/abalone.data"
)
abalone = pd.read_csv(url, header=None)
```

In [2]:

```
abalone.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

In [3]:

```
abalone.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       4177 non-null   object 
 1   1       4177 non-null   float64
 2   2       4177 non-null   float64
 3   3       4177 non-null   float64
 4   4       4177 non-null   float64
 5   5       4177 non-null   float64
 6   6       4177 non-null   float64
 7   7       4177 non-null   float64
 8   8       4177 non-null   int64  
dtypes: float64(7), int64(1), object(1)
memory usage: 293.8+ KB
```

In [4]:

```
abalone.shape
```

Out[4]:

```
(4177, 9)
```

You can see that the column names are still missing.

You can find those names in the abalone.names file on the UCI machine learning repository.

You can add them to your DataFrame as follows:

In [5]:

```
abalone.columns = [
    "Sex",
    "Length",
    "Diameter",
    "Height",
    "Whole weight",
    "Shucked weight",
    "Viscera weight",
    "Shell weight",
    "Rings",
]
```

In [6]:

```
abalone.head()
```

Out[6]:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

The imported data should now be more understandable.

But there's one other thing that you should do: You should remove the Sex column.

The goal of the current exercise is to use physical measurements to predict the age of the abalone.

Since sex is not a purely physical measure, you should remove it from the dataset.

You can delete the Sex column using .drop:

In [7]:

```
abalone = abalone.drop("Sex", axis=1)
```

In [8]:

```
1 abalone.head()
```

Out[8]:

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Descriptive Statistics From the Abalone Dataset

When working on machine learning, you need to have an idea of the data you're working with.

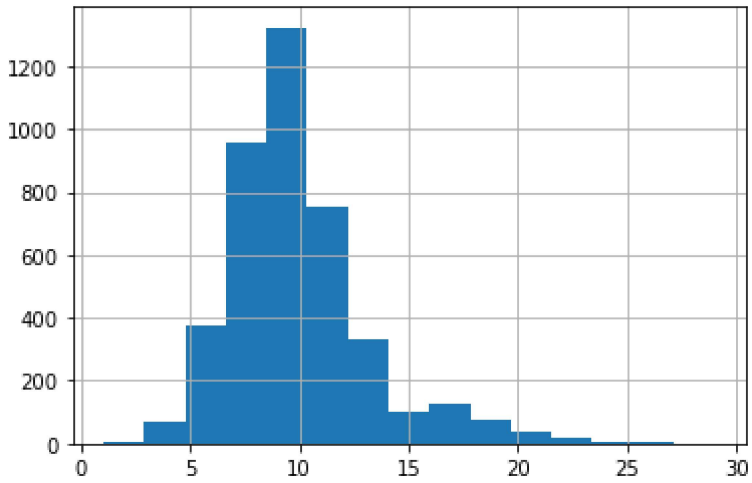
Without going into too much depth, here's a look at some exploratory statistics and graphs.

The target variable of this exercise is Rings, so you can start with that.

A histogram will give you a quick and useful overview of the age ranges that you can expect:

In [9]:

```
1 import matplotlib.pyplot as plt
2 abalone["Rings"].hist(bins=15)
3 plt.show()
```



The code below uses the pandas plotting functionality to generate a histogram with fifteen bins.

The decision to use fifteen bins is based on a few trials.

When defining the number of bins, you generally try to have neither too many observations per bin nor too few.

Too few bins can hide certain patterns, while too many bins can make the histogram lack smoothness.

You can see the histogram in the following graph:

```
The histogram shows that most abalones in the dataset have between five and fifteen rings, but that it's possible to get up to twenty-five rings.
The older abalones are underrepresented in this dataset.
This seems intuitive, as age distributions are generally skewed like this due to natural processes.

A second relevant exploration is to find out which of the variables, if any, have a strong correlation with the age.
A strong correlation between an independent variable and your goal variable would be a good sign, as this would confirm that physical measurements and age are related.
```

You can observe the complete correlation matrix in correlation_matrix.

The most important correlations are the ones with the target variable Rings.

You can get those correlations like this:

In [10]:

```
correlation_matrix = abalone.corr()
correlation_matrix["Rings"]
```

Out[10]:

```
Length            0.556720
Diameter          0.574660
Height            0.557467
Whole weight      0.540390
Shucked weight    0.420884
Viscera weight    0.503819
Shell weight      0.627574
Rings             1.000000
Name: Rings, dtype: float64
```

```
Now look at the correlation coefficients for Rings with the other variables.
The closer they are to 1, the more correlation there is.

You can conclude that there's at least some correlation between physical measurements of adult abalones and their age, yet it's also not very high.
Very high correlations mean that you can expect a straightforward modeling process.
In this case, you'll have to try and see what results you can obtain using the kNN algorithm.
```

A Step-by-Step kNN From Scratch in Python

Plain English Walkthrough of the kNN Algorithm

```
The kNN algorithm is a little bit atypical as compared to other machine learning algorithms.
As you saw earlier, each machine learning model has its specific formula that needs to be estimated.
The specificity of the k-Nearest Neighbors algorithm is that this formula is computed not at the moment of fitting but rather at the moment of prediction.
This isn't the case for most other models.
```

```
When a new data point arrives, the kNN algorithm, as the name indicates, will start by finding the nearest neighbors of this new data point.
Then it takes the values of those neighbors and uses them as a prediction for the new data point.
```

```
As an intuitive example of why this works, think of your neighbors.
Your neighbors are often relatively similar to you. They're probably in the same socioeconomic class as you.
Maybe they have the same type of work as you, maybe their children go to the same school as yours, and so on.
But for some tasks, this kind of approach is not as useful.
For instance, it wouldn't make any sense to look at your neighbor's favorite color to predict yours.
```

```
The kNN algorithm is based on the notion that you can predict the features of a data point based on the features of its neighbors. In some cases, this method of prediction may be successful, while in other cases it may not.
Next, you'll look at the mathematical description of "nearest" for data points and the methods to combine multiple neighbors into one prediction.
```

Define "Nearest" Using a Mathematical Definition of Distance

To find the data points that are closest to the point that you need to predict, you can use a mathematical definition of distance called Euclidean distance.

To get to this definition, you should first understand what is meant by the difference of two vectors.

You must understand that your data points are actually vectors.

You can then compute the distance between them by computing the norm of the difference vector.

You can compute this in Python using linalg.norm() from NumPy.

```
In this case, you should compute the norm of the difference vector c to obtain the distance between the data points.
```

In [11]:

```
# Example :
import numpy as np
a = np.array([2, 2])
b = np.array([4, 4])
np.linalg.norm(a - b)
```

Out[11]:

```
2.8284271247461903
```

```
In this code block, you define your data points as vectors.
You then compute norm() on the difference between two data points.
This way, you directly obtain the distance between two multidimensional points.
Even though the points are multidimensional, the distance between them is still a scalar, or a single value.
```

```
In case you want to get more details on the math, you can have a look at the Pythagorean theorem to understand how the Euclidean distance formula is derived.
```

Find the k Nearest Neighbors

Now that you have a way to compute the distance from any point to any point, you can use this to find the nearest neighbors of a point on which you want to make a prediction.

You need to find a number of neighbors, and that number is given by k.

The minimum value of k is 1. This means using only one neighbor for the prediction.

The maximum is the number of data points that you have. This means using all neighbors.

The value of k is something that the user defines.

Optimization tools can help you with this, as you'll see in the last part of this tutorial.

```
Now, to find the nearest neighbors in NumPy, go back to the Abalone Dataset.
As you've seen, you need to define distances on the vectors of the independent
variables, so you should first get your pandas DataFrame into a NumPy array using the
.values attribute:
```

In [12]:

```
X = abalone.drop("Rings", axis=1)
X = X.values
y = abalone["Rings"]
y = y.values
```

```
This code block generates two objects that now contain your data: X and y.
X is the independent variables and y is the dependent variable of your model.
Note that you use a capital letter for X but a lowercase letter for y.
This is often done in machine learning code because mathematical notation generally
uses a capital letter for matrices and a lowercase letter for vectors.
```

In [13]:

```
# You can create the NumPy array for this data point as follows:
new_data_point = np.array([
    0.569552,
    0.446407,
    0.154437,
    1.016849,
    0.439051,
    0.222526,
    0.291208,
])
```

In [14]:

```
# The next step is to compute the distances between this new data point and each of the
distances = np.linalg.norm(X - new_data_point, axis=1)
```

In [15]:

```
print(distances)
```

```
[0.59739395 0.9518455  0.40573594 ... 0.20397872 0.14342627 1.10583307]
```

You now have a vector of distances, and you need to find out which are the three closest neighbors.

To do this, you need to find the IDs of the minimum distances.

You can use a method called .argsort() to sort the array from lowest to highest, and you can take the first k elements to obtain the indices of the k nearest neighbors:

In [16]:

```
k = 3
nearest_neighbor_ids = distances.argsort()[:k]
nearest_neighbor_ids
```

Out[16]:

```
array([4045, 1902, 1644], dtype=int64)
```

```
This tells you which three neighbors are closest to your new_data_point.
In the next paragraph, you'll see how to convert those neighbors in an estimation.
```

Voting or Averaging of Multiple Neighbors

Having identified the indices of the three nearest neighbors of your abalone of unknown age, you now need to combine those neighbors into a prediction for your new data point.

As a first step, you need to find the ground truths for those three neighbors:

In [17]:

```
nearest_neighbor_rings = y[nearest_neighbor_ids]
nearest_neighbor_rings
```

Out[17]:

```
array([ 9, 11, 10], dtype=int64)
```

Now that you have the values for those three neighbors, you'll combine them into a prediction for your new data point.

Combining the neighbors into a prediction works differently for regression and classification.

Average for Regression

In regression problems, the target variable is numeric.

You combine multiple neighbors into one prediction by taking the average of their values of the target variable.

You can do this as follows:

In [18]:

```
prediction = nearest_neighbor_rings.mean()
```

```
You'll get a value of 10 for prediction. This means that the 3-Nearest Neighbor prediction for your new data point is 10.
You could do the same for any number of new abalones that you want.
```

Mode for Classification

```
In classification problems, the target variable is categorical.
As discussed before, you can't take averages on categorical variables.
For example, what would be the average of three predicted car brands? That would be impossible to say.
You can't apply an average on class predictions.
```

```
Instead, in the case of classification, you take the mode.
The mode is the value that occurs most often.
This means that you count the classes of all the neighbors, and you retain the most common class.
The prediction is the value that occurs most often among the neighbors.
```

```
If there are multiple modes, there are multiple possible solutions.
You could select a final winner randomly from the winners.
You could also make the final decision based on the distances of the neighbors, in which case the mode of the closest neighbors would be retained.
```

```
You can compute the mode using the SciPy mode() function.
As the abalone example is not a case of classification, the following code shows how you can compute the mode for a toy example:
```

In [19]:

```
import scipy.stats
class_neighbors = np.array(["A", "B", "B", "C"])
scipy.stats.mode(class_neighbors)
```

Out[19]:

```
ModeResult(mode=array(['B'], dtype='<U1'), count=array([2]))
```

As you can see, the mode in this example is "B" because it's the value that occurs most often in the input data.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

Fit kNN in Python Using scikit-learn

```
While coding an algorithm from scratch is great for learning purposes, it’s usually not very practical when working on a machine learning task.
In this section, you’ll explore the implementation of the kNN algorithm used in scikit-learn, one of the most comprehensive machine learning packages in Python.
```

Splitting Data Into Training and Test Sets for Model Evaluation

```
In this section, you’ll evaluate the quality of your abalone kNN model. In the previous sections, you had a technical focus, but you’re now going to have a more pragmatic and results-oriented point of view.
```

```
There are multiple ways of evaluating models, but the most common one is the train-test split. When using a train-test split for model evaluation, you split the dataset into two parts:

1- Training data is used to fit the model. For kNN, this means that the training data will be used as neighbors.
2- Test data is used to evaluate the model. It means that you’ll make predictions for the number of rings of each of the abalones in the test data and compare those results to the known true number of rings.
```

In [20]:

```
# You can split the data into training and test sets in Python using scikit-learn’s bui
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=12345
)
```

```
The test_size refers to the number of observations that you want to put in the training data and the test data.
If you specify a test_size of 0.2, your test_size will be 20 percent of the original data, therefore leaving the other 80 percent as training data.

The random_state is a parameter that allows you to obtain the same results every time the code is run.
train_test_split() makes a random split in the data, which is problematic for reproducing the results.
Therefore, it’s common to use random_state. The choice of value in random_state is arbitrary.

In the above code, you separate the data into training and test data.
This is needed for objective model evaluation.
You can now proceed to fit a kNN model on the training data using scikit-learn.
```

Fitting a kNN Regression in scikit-learn to the Abalone Dataset

```
To fit a model from scikit-learn, you start by creating a model of the correct class.
At this point, you also need to choose the values for your hyperparameters.
For the kNN algorithm, you need to choose the value for k, which is called
n_neighbors in the scikit-learn implementation. Here's how you can do this in Python:
```

In [21]:

```
from sklearn.neighbors import KNeighborsRegressor
knn_model = KNeighborsRegressor(n_neighbors=3)
```

```
You create an unfitted model with knn_model.
This model will use the three nearest neighbors to predict the value of a future data
point.
To get the data into the model, you can then fit the model on the training dataset:
```

In [22]:

```
knn_model.fit(X_train, y_train)
```

Out[22]:

```
KNeighborsRegressor(n_neighbors=3)
```

```
Using .fit(), you let the model learn from the data.
At this point, knn_model contains everything that's needed to make predictions on new
abalone data points.
That's all the code you need for fitting a kNN regression using Python!
```

Using scikit-learn to Inspect Model Fit

Fitting a model, however, isn't enough.

In this section, you'll look at some functions that you can use to evaluate the fit.

There are many evaluation metrics available for regression, but you'll use one of the most common ones, the root-mean-square error (RMSE). The RMSE of a prediction is computed as follows:

1- Compute the difference between each data point's actual value and predicted value.

2- For each difference, take the square of this difference.

3- Sum all the squared differences.

4- Take the square root of the summed value.

```
To start, you can evaluate the prediction error on the training data.
This means that you use the training data for prediction, so you know that the result
should be relatively good.
You can use the following code to obtain the RMSE:
```

In [23]:

```
from sklearn.metrics import mean_squared_error
from math import sqrt
train_preds = knn_model.predict(X_train)
mse = mean_squared_error(y_train, train_preds)
rmse = sqrt(mse)
print(rmse)
```

```
1.653705966446084
```

In this code, you compute the RMSE using the knn_model that you fitted in the previous code block.

You compute the RMSE on the training data for now. For a more realistic result, you should evaluate the performances on data that aren't included in the model.

This is why you kept the test set separate for now.

You can evaluate the predictive performances on the test set with the same function as before:

In [24]:

```
test_preds = knn_model.predict(X_test)
mse = mean_squared_error(y_test, test_preds)
rmse = sqrt(mse)
print(rmse)
```

```
2.375417924000521
```

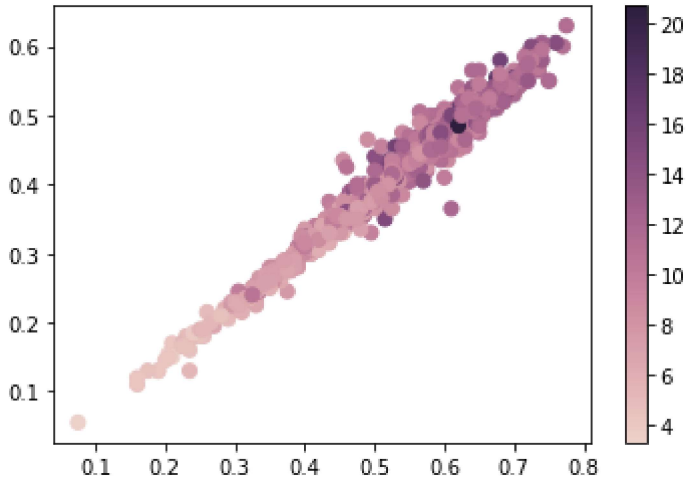
```
In this code block, you evaluate the error on data that wasn't yet known by the
model. This more-realistic RMSE is slightly higher than before. The RMSE measures the
average error of the predicted age, so you can interpret this as having, on average,
an error of 1.65 years. Whether an improvement from 2.37 years to 1.65 years is good
is case specific. At least you're getting closer to correctly estimating the age.

Until now, you've only used the scikit-learn kNN algorithm out of the box. You
haven't yet done any tuning of hyperparameters and a random choice for k. You can
observe a relatively large difference between the RMSE on the training data and the
RMSE on the test data. This means that the model suffers from overfitting on the
training data: It does not generalize well.

This is nothing to worry about at this point. In the next part, you'll see how to
optimize the prediction error or test error using various tuning methods.
```

Plotting the Fit of Your Model

```
A last thing to look at before starting to improve the model is the actual fit of
your model.
To understand what the model has learned, you can visualize how your predictions have
been made using Matplotlib:
```

In [25]:

```
import seaborn as sns
cmap = sns.cubehelix_palette(as_cmap=True)
f, ax = plt.subplots()
points = ax.scatter(
    X_test[:, 0], X_test[:, 1], c=test_preds, s=50, cmap=cmap
)
f.colorbar(points)
plt.show()
```



In this code block, you use Seaborn to create a scatter plot of the first and second columns of X_test by subsetting the arrays X_test[:,0] and X_test[:,1].

Remember from before that the first two columns are Length and Diameter.

They are strongly correlated, as you've seen in the correlations table.

You use c to specify that the predicted values (test_preds) should be used as a colorbar.

The argument s is used to specify the size of the points in the scatter plot.

You use cmap to specify the cubehelix_palette color map.

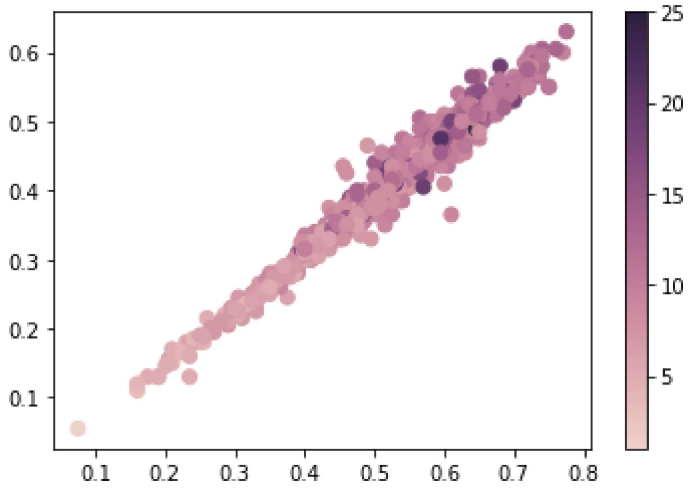
To learn more about plotting with Matplotlib, check out Python Plotting With Matplotlib.

```
On this graph, each point is an abalone from the test set, with its actual length and
actual diameter on the X- and Y-axis, respectively. The color of the point reflects
the predicted age.
You can see that the longer and larger an abalone is, the higher its predicted age.
This is logical, and it's a positive sign.
It means that your model is learning something that seems correct.
```

To confirm whether this trend exists in actual abalone data, you can do the same for the actual values by simply replacing the variable that is used for c:

In [26]:

```
cmap = sns.cubehelix_palette(as_cmap=True)
f, ax = plt.subplots()
points = ax.scatter(
    X_test[:, 0], X_test[:, 1], c=y_test, s=50, cmap=cmap
)
f.colorbar(points)
plt.show()

# This code uses Seaborn to create a scatterplot with a colorbar. It produces the follo
```




```
This confirms that the trend your model is learning does indeed make sense.

You could extract a visualization for each combination of the seven independent
variables.
For this tutorial, that would be too long, but don't hesitate to try it out.
The only thing to change is the columns that are specified in the scatter.

These visualizations are two-dimensional views of a seven-dimensional dataset.
If you play around with them, it will give you a great understanding of what the
model is learning and, maybe, what it's not learning or is learning wrong.
```

Tune and Optimize kNN in Python Using scikit-learn

```
There are numerous ways you can improve your predictive score.
Some improvements could be made by working on the input data using data wrangling,
but in this tutorial, the focus is on the kNN algorithm.
Next, you'll look at ways to improve the algorithm part of the modeling pipeline.
```

Improving kNN Performances in scikit-learn Using GridSearchCV

Until now, you've always worked with k=3 in the kNN algorithm, but the best value for k is something that you need to find empirically for each dataset.

```
When you use few neighbors, you have a prediction that will be much more variable
than when you use more neighbors:

```

```
* If you use one neighbor only, the prediction can strongly change from one point to the other.
When you think about your own neighbors, one may be quite different from the others.
If you lived next to an outlier, your 1-NN prediction would be wrong.

* If you have multiple data points, the impact of one extremely different neighbor will be much less.

* If you use too many neighbors, the prediction of each point risks being very close.
Let's say that you use all neighbors for a prediction.
In that case, every prediction would be the same.
```

```
To find the best value for k, you're going to use a tool called GridSearchCV.
This is a tool that is often used for tuning hyperparameters of machine learning models.
In your case, it will help by automatically finding the best value of k for your dataset.
```

GridSearchCV is available in scikit-learn, and it has the benefit of being used in almost the exact same way as the scikit-learn models:

In [27]:

```
from sklearn.model_selection import GridSearchCV
parameters = {"n_neighbors": range(1, 50)}
gridsearch = GridSearchCV(KNeighborsRegressor(), parameters)
gridsearch.fit(X_train, y_train)
```

Out[27]:

```
GridSearchCV(estimator=KNeighborsRegressor(),
             param_grid={'n_neighbors': range(1, 50)})
```

```
Here, you use GridSearchCV to fit the model.
In short, GridSearchCV repeatedly fits kNN regressors on a part of the data and tests the performances on the remaining part of the data.
Doing this repeatedly will yield a reliable estimate of the predictive performance of each of the values for k.
In this example, you test the values from 1 to 50.
```

In the end, it will retain the best performing value of k, which you can access with .best_params_:

In [28]:

```
gridsearch.best_params_
```

Out[28]:

```
{'n_neighbors': 25}
```

In this code, you print the parameters that have the lowest error score.

With .best_params_, you can see that choosing 25 as value for k will yield the best predictive performance

```
Now that you know what the best value of k is, you can see how it affects your train and test performances:
```

In [29]:

```
train_preds_grid = gridsearch.predict(X_train)
train_mse = mean_squared_error(y_train, train_preds_grid)
train_rmse = sqrt(train_mse)
test_preds_grid = gridsearch.predict(X_test)
test_mse = mean_squared_error(y_test, test_preds_grid)
test_rmse = sqrt(test_mse)
print(train_rmse)

print(test_rmse)
```

```
2.0731180327543384
2.1700197339962175
```

```
With this code, you fit the model on the training data and evaluate the test data.
You can see that the training error is worse than before, but the test error is
better than before.
This means that your model fits less closely to the training data.
Using GridSearchCV to find a value for k has reduced the problem of overfitting on
the training data.
```

Adding Weighted Average of Neighbors Based on Distance

```
Using GridSearchCV, you reduced the test RMSE from 2.37 to 2.17.
In this section, you'll see how to improve the performances even more.

Below, you'll test whether the performance of your model will be any better when
predicting using a weighted average instead of a regular average.
This means that neighbors that are further away will less strongly influence the
prediction.
```

You can do this by setting the weights hyperparameter to the value of "distance".

However, setting this weighted average could have an impact on the optimal value of k.

Therefore, you'll again use GridSearchCV to tell you which type of averaging you should use:

In [30]:

```
parameters = {
    "n_neighbors": range(1, 50),
    "weights": ["uniform", "distance"],
}
gridsearch = GridSearchCV(KNeighborsRegressor(), parameters)
gridsearch.fit(X_train, y_train)



gridsearch.best_params_

test_preds_grid = gridsearch.predict(X_test)
test_mse = mean_squared_error(y_test, test_preds_grid)
test_rmse = sqrt(test_mse)
print(test_rmse)
```

```
2.1634265584947485
```

Here, you test whether it makes sense to use a different weighing using your GridSearchCV.

Applying a weighted average rather than a regular average has reduced the prediction error from 2.17 to 2.1634.

Although this isn't a huge improvement, it's still better, which makes it worth it.

Further Improving on kNN in scikit-learn With Bagging

```
As a third step for kNN tuning, you can use bagging. Bagging is an ensemble method,
or a method that takes a relatively straightforward machine learning model and fits a
large number of those models with slight variations in each fit.
Bagging often uses decision trees, but kNN works perfectly as well.

Ensemble methods are often more performant than single models. One model can be wrong
from time to time, but the average of a hundred models should be wrong less often.
The errors of different individual models are likely to average each other out, and
the resulting prediction will be less variable.
```

You can use scikit-learn to apply bagging to your kNN regression using the following steps.

First, create the KNeighborsRegressor with the best choices for k and weights that you got from GridSearchCV:

In [41]:

```
best_k = gridsearch.best_params_["n_neighbors"]
best_weights = gridsearch.best_params_["weights"]
bagged_knn = KNeighborsRegressor(n_neighbors=best_k, weights=best_weights)
```

Then import the BaggingRegressor class from scikit-learn and create a new instance with 100 estimators using the bagged_knn model:

In [54]:

```
from sklearn.ensemble import BaggingRegressor
bagging_model = BaggingRegressor(bagged_knn, n_estimators=100)
bagging_model.fit(X_train, y_train)
```

Out[54]:

```
BaggingRegressor(base_estimator=KNeighborsRegressor(n_neighbors=25,
                                                    weights='distance'),
                 n_estimators=100)
```

In [55]:

```
# Now you can make a prediction and calculate the RMSE to see if it improved:
test_preds_grid = bagging_model.predict(X_test)
test_mse = mean_squared_error(y_test, test_preds_grid)
test_rmse = sqrt(test_mse)
test_rmse
```

Out[55]:

```
2.164753415586377
```

The prediction error on the bagged kNN is 2.1616, which is slightly smaller than the previous error that you

obtained. It does take a little more time to execute, but for this example, that's not problematic.

In [ ]:

```
1
```