# Understanding the Scikit-Learn API

## Overview

In this lab you’ll use the scikit-learn API to perform supervised learning on sample data about California house prices.

You’ll load the California sample data from scikit-learn, familiarize yourself with the shape of the data, and then use linear regression to find a line of best fit for California house prices.

## Source folders

Student folder : PythonML\Student\03-ScikitLearnApi

Solution folder: PythonML\Solutions\03-ScikitLearnApi

## Roadmap

1. Loading the California dataset from scikit-learn
2. Converting the feature matrix into a Pandas DataFrame
3. Splitting the dataset into “training” data and “testing” data
4. Creating a linear regression model and predicting labels for new data
5. (If time permits) Plotting predicted vs. actual house prices
6. (If time permits) Determining the root mean squared error

## Exercise 1: Loading the California dataset from scikit-learn

Create a new Python script file in the *student* folder. Give the file a name such as predictCaliforniaHousePrices.py.

In the script file, add code to do the following:

* From sklearn.datasets, import the fetch\_california\_housing function.
* Call the fetch\_california\_housing() function, to load the California house price dataset. Store the dataset in a variable named housing.

Python objects have a keys() function, which tells you the names of the properties defined in the object. Add the following code, to print the keys for the housing dataset object:

print(housing.keys())

Add code as follows, to investigate the shape of the feature matrix and the target array:

print("\nDetails about the feature matrix")

print(housing.data.shape)

print(housing.feature\_names)

print(housing.data)

print("\nDetails about the target array")

print(housing.target.shape)

print(housing.target)

You should find that the housing dataset had more than 20,000 rows, which is realistic enough to get started with machine learning concepts.

**Exercise 2: Converting the feature matrix into a Pandas** **DataFrame**

As you saw in the previous exercise, the data in the California dataset (i.e. housing.data) is a 2D array. There are approximately 20,000 rows (each row represents a house) and 8 columns (representing the 8 features for each house). Generally, it’s beneficial to convert the data into a pandas DataFrame, which is easier to work with than a 2D array. You can do this as follows:

import pandas as pd

X = pd.DataFrame(housing.data)

Add the following statement, to print the first 5 rows in the DataFrame:

print(X.head())

Note the data columns are just called 0, 1, 2, 3, …. 7.

If you want more meaningful column names (and who wouldn’t 😊), you can assign the feature names as column names as follows. The print() statement confirms the column names are now meaningful:

X.columns = housing.feature\_names

print(X.head())

One last step in this exercise… assign the housing target array (i.e. the house prices) to a variable named y for readability and according to convention:

y = housing.target

**Exercise 3: Splitting the dataset into “training” data and “testing” data**

A common approach in machine learning is to split the samples in a dataset into two portions:

* A relatively large portion of samples (e.g. 80%) that can be used to *train* the model.
* The remainder of the samples (e.g. 20%) that can be used to *test* the quality of the values predicted by the model.

This is such a common task in machine learning that scikit-learn has an off-the-shelf function called train\_test\_split() to do it. Add the following code to your script:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X,

y,

train\_size = 0.80)

Here’s a quick explanation of the train\_test\_split() function:

* The function takes any number of arrays of the same size.
* The train\_size parameter indicates the portion of the data you want to treat as “training” data. Here, 80% will be treated as “training” data, and 20% as “testing” data.
* The function returns a bunch of arrays, according to the specified split. In our example, X\_train and X\_test will hold the “training” and “testing” portions of X, and y\_train and y\_test will hold the “training” and “test” portions of y.

Add the following code, to show the shape of the resultant arrays:

print(X\_train.shape)

print(X\_test.shape)

print(y\_train.shape)

print(y\_test.shape)

**Exercise 4: Creating a linear regression model and predicting labels for new data**

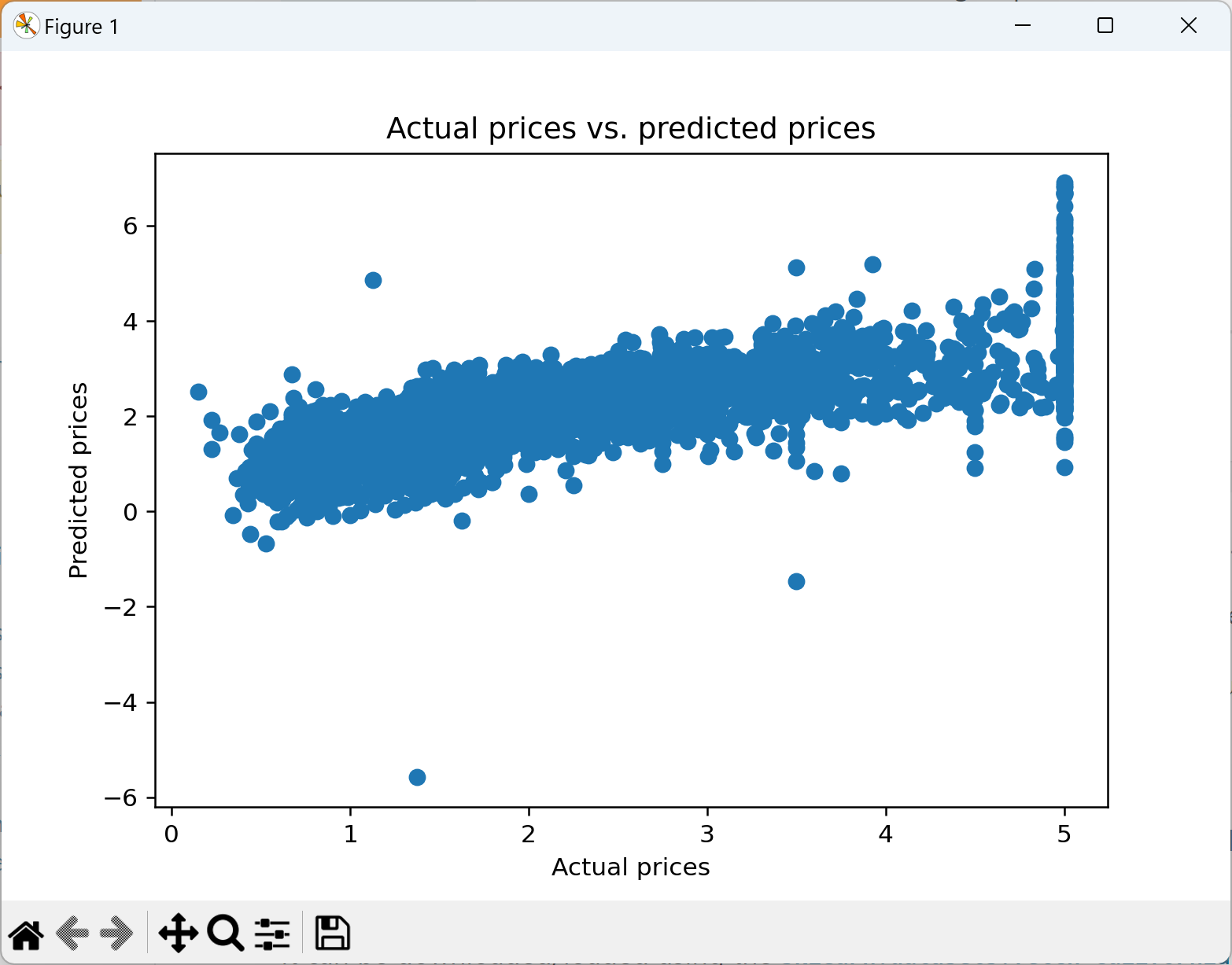
In this exercise you’ll use scikit-learn to create a linear regression model, fit it to the training data about California house prices, and then use the model to predict the prices of other houses in California.

Follow these steps (refer to the chapter notes for more info, if necessary):

* Import the LinearRegression model class.
* Create a LinearRegression model object, and fit it to the "training" dataset.
* Use the model to predict labels (i.e., house prices) for the "testing" dataset. To do this, pass the X\_test data as a parameter to the predict() function. The function returns predicted house prices for this test data.
* Print the predicted house prices for the test data, alongside the actual house prices for the test data. This will give you an inkling over the quality of the model – the better the model, the closer the predicted and actual prices will be. What do you find, and why…?

**Exercise 5 (If time permits): Plotting predicted vs. actual house prices**

Using matplotlib, draw a scatterplot graph that shows predicted vs. actual house prices. Here’s the graph we obtained when we ran the code in the solution script. In a perfect model, the dots would form a completely straight line because the predicted and actual values would always match exactly…



**Exercise 6 (If time permits): Determining the root mean squared error**

In statistics, the root mean squared error is a measurement of the quality of predicted vs. actual results:

* You pump in a series of predicted and actual results, and it calculates the square of the difference (i.e., error) between predicted and actual values. It uses *squares* to always get positive deltas.
* It then calculates the mean (average) of the squared errors.
* You can then take the square root, to obtain the average error in the same units as the data itself (e.g. $ for California house prices).

This is such a common technique in machine learning that scikit-learn has a standard function called to mean\_squared\_error() to calculate the mean squared error. The following code shows how to use it:

from sklearn.metrics import mean\_squared\_error

import math

mse = mean\_squared\_error(y\_test, Y\_pred)

rmse = math.sqrt(mse)

print("Root mean squared error %f" % rmse)