Report 7 - Intermediate Python and Data Analysis in Python

May 31, 2020

1 Intermediate Python

In this document we will cover some basic and intermediate concepts in Python. We will be using Python version 3.7.3. This can be checked quite easily by running `python --version` in the terminal. The beginning of this document closely follows the Intermediate Python course offered by the ACRC at the University of Bristol, and the second half of this document follows the Applied Data Analysis in Python course. The courses can be found here and here.

1.1 Functions

We begin with discussing functions. When we encounter an order of operations which we may frequently want to repeat (perhaps with different arguments), it may be best to put the code into a *function*.

As a toy example, suppose we routinely want to add two arrays element-wise. We could easily write a function to do this for us. The code can be found below.

[10]:
```
def add_array(x, y):
    z = []
    for x_elem, y_elem in zip(x,y):
        z.append(x_elem + y_elem)
    return z
```

We can easily test the above code by simply observing a set of inputs and the funtion's output.

[11]:
```
add_array([1,2,3], [4,5,6])
```

[11]:
```
[5, 7, 9]
```

Looking at the above function demonstrates how we define a function in Python. We must define the function name and its arguments using `def` and follow this with a colon `:`.

It's also clear that in order to call a function we must use its name and pass it its arguments. Without supplying the needed arguments (or if we provide too many), the function will not run as demonstrated below.

[12]:
```
add_array()
```

```
␣
↪---------------------------------------------------------------------------
```

```
        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-12-b51761f97c43> in <module>
    ----> 1 add_array()


        TypeError: add_array() missing 2 required positional arguments: 'x' and␣
↪'y'
```

[13]:
```
add_array([1,2],[2,3],[3,4])
```

```
      ␣
↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-13-0ff9f97973bb> in <module>
    ----> 1 add_array([1,2],[2,3],[3,4])


        TypeError: add_array() takes 2 positional arguments but 3 were given
```

Note that we can also assign the output of a function to a variable, as you might expect.

[14]:
```
test = add_array([1,2], [3,4])
print(test)
```

```
[4, 6]
```

A useful feature in Python which is not offered by R, is the ability to assign multiple variables at once. This is best demonstrated by an example.

[17]:
```
a, b = 1, 2
print(a)
print(b)
```

```
1
2
```

This can also be used in confunction with functions! We can easily assign the multiple outputs of a function to appropriate variables when necessary.

[18]:
```
def multiple_output(a, b):
    return a, b, 2*a, 2*b
```

[20]:
```
a, b, two_a, two_b = multiple_output(a, b)
print(a, b, two_a, two_b)
```

```
1 2 2 4
```

1.2 Modules

Modules provide a natural progression from functions. In order to reuse code across multiple scripts, we would need to copy and paste functions into each script. One way to overcome this problem in Python is to use *modules*.

Modules in Python allow us to import a set of functions from a script. Often, certain functionality is packaged into a module. For example, we may have a script called `arrays.py` and `tensors.py` — one will provide a set of functions for arrays, the other for tensors.

We will continue with the example used in the above section. We will add our function `add_array` to the module (file) `arrays.py`, and import this module below.

[22]:
```
import arrays

arrays.add_array([1,2], [3,4])
```

[22]:
```
[4, 6]
```

We see that in order to use functions within the `arrays` module, we must tell Python that we are calling functions from the `arrays` module by `arrays.<module_function>`.

If we want to avoid this, we can instead import the module using `from arrays import *`. This works because `from <module> import <function>` directly imports a function from a module into the local namespace.

[28]:
```
from arrays import *

add_array([1,2], [3, 4])
```

[28]:
```
[4, 6]
```

1.3 Classes

Python also provides a simple-to-use implementation of object-oriented programming. This again allows us to easily reuse functionality. Classes simplify code by combining functions/code with data. For example, when analyzing data we may define a linear regression class which will take our data (and perhaps a model formula or design matrix) as an argument and create a linear model object. The class could initialize the object by computing coefficient estimates and predictions over the test set, and could have a class-specific function such as `.predict()` which would take a datapoint as an argument and output a prediction.

Below we demonstrate the use of classes by implementing a linear regression model.

[2]:
```
import numpy as np

class LinearRegression:
    def __init__(self, X, y):
        self._X = X
        self._y = y
```

```
        self.betahat = (np.linalg.inv(np.matmul(np.transpose(X),X)) @ (np.
↪transpose(X) @ y))

    def predict(self, xstar):
        return np.matmul(xstar, self.betahat)
```

Let's simulate some data and check that the above code works. We simulate $X \sim N(0, 0.1^2)$, and generate y using $y = 1 + 1.5X + \epsilon$, where the noise $\epsilon \sim N(0, 0.3^2)$.

[3]:
```
mu, sigma = 0, 0.1
X = np.column_stack((np.repeat(1,100), np.random.normal(mu, sigma, 100)))
y = X[:,0] + 1.5*X[:,1] + np.random.normal(0,0.3,100)
```

[4]:
```
model = LinearRegression(X, y)
```

Let's check the coefficient estimates produced.

[5]:
```
model.betahat
```

[5]:
```
array([0.97220882, 1.70190582])
```

We can also check that the predict function works as expected.

[6]:
```
xstar=np.array([[1, 0], [1,1.5], [1,3]])
model.predict(xstar)
```

[6]:
```
array([0.97220882, 3.52506755, 6.07792627])
```

Given that we know the model formulation, the predictions seem logical.

1.3.1 Constructing classes

Let's discuss exactly how to construct classes.

To begin constructing a class we must use the `class` keyword. Within the class we must define `__init__` which is known as the constructor. `__init__` is frequently referred to as "dunder init" (where dunder is short for double underscore). The constructor is required in all classes and specifies how to create and intialize an object within the class. In the example above the constructor stores the dataset (X, y) and computes the parameter estimates $\hat{\beta}$ which it stores under `.betahat`. Thus, when we create an object of class `LinearRegression`, the coefficient estimates are immediately computed in the background and attached to the object.

Note that the variables are attached to `self` which is also the first argument of functions of the class. `self` is a special variable only available to functions of the class, and provides access to all of the object's variables e.g. in the function `predict` we use `betahat` which is a variable belonging to `self`.

Note that we do not need to pass `self` as an argument to the function or class, Python does this for us. We also use underscores to denote variables which are private to the class and not necessarily for users of the class/function. You may notice that an underscore is not used for the definition of `betahat`, as it is very likely the user of the class would like access to the variable.

2 Data Analysis in Python

In this section we will introduce libraries for data analysis in Python. We will use scikit-learn for the analysis, and we will use the dataframes offered by pandas. In order to simulate data we will

use NumPy.

We begin with something we are very familiar with: linear regression.

2.1 Linear Regression

In this subsection I will assume that we are very familiar with linear regression, and will only discuss how to implement linear regression in Python. First we must generate some data.

We will generate the data using Numpy, but before generating data we must set the seed for reproducibility. This can be done easily using `np.random.RandomState(seed)`.

[26]:
```
import numpy as np

rng = np.random.RandomState(123)

# dataset size
n = 100

x = rng.uniform(-1,1,n)
y = 0.5 + 2*x + rng.normal(0,0.2,n)
```

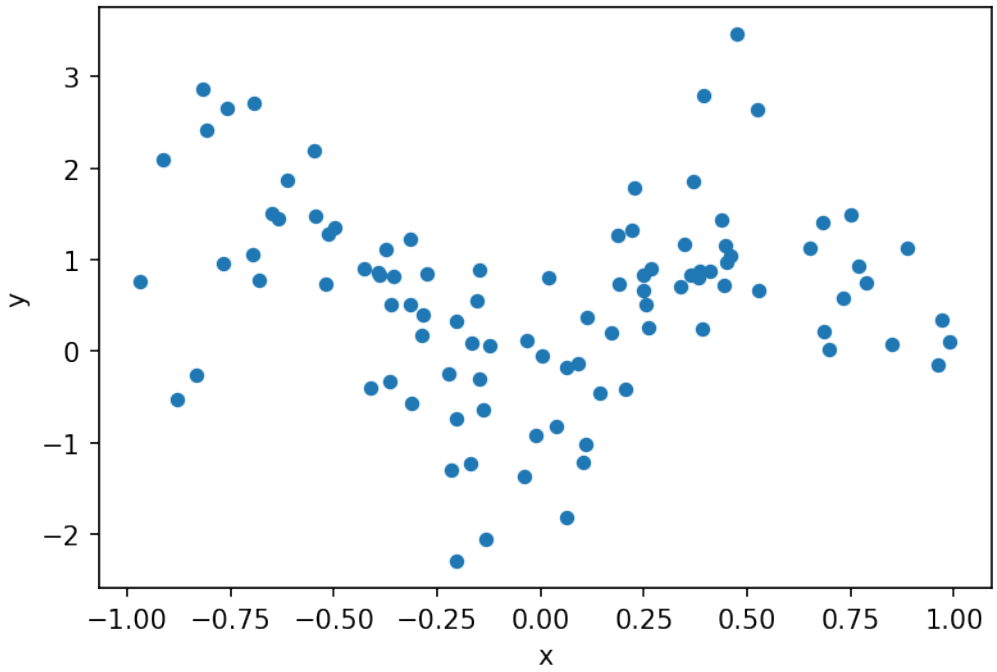
We can visualize the data relatively easily using a pandas dataframe and matplotlib.

[84]:
```
%matplotlib inline
plt.rcParams['figure.dpi']= 150
from pandas import DataFrame

data = DataFrame({"x": x,
                  "y": y})

data.plot.scatter("x", "y")
```

[84]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb7902ecd30>
```



Now in order to fit a linear model to the data we will create a linear regression object using scikit-learn. scikit-learn provides many other methods for modeling data which you can investigate in the documentation. For regression problems some popular methods provided are (kernel) ridge regression, generalized linear models, and Gaussian process regression.

Each model provided by scikit-learn is a Python class, and an implementation can be seen below.

[10]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression(fit_intercept=True)
```

In the above code we have *created* our model and specified its hyperparameters, but we have not provided it with data to fit. In order to train the model we must call the method `fit()` on the object. scikit-learn methods take input data in the following format 1. The input must have dimensions `[n_samples, n_features]` (i.e. the standard $n \times p$ dimensional model matrix). 2. The response must have dimension `[n_samples]`.

Note that even though we only have one feature, the input must still have dimensions `[100, 1]`. We have stored our data in a pandas DataFrame and thus to extract the data in this format we must use `data[["x"]]`. If we use `data["x"]` the data will be extracted as a pandas `Series` object with shape `[n_samples]`. Thus, in order to extract our response from the DataFrame we will use `data["y"]`.

[11]:
```
model.fit(data[["x"]], data["y"])
```

[11]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

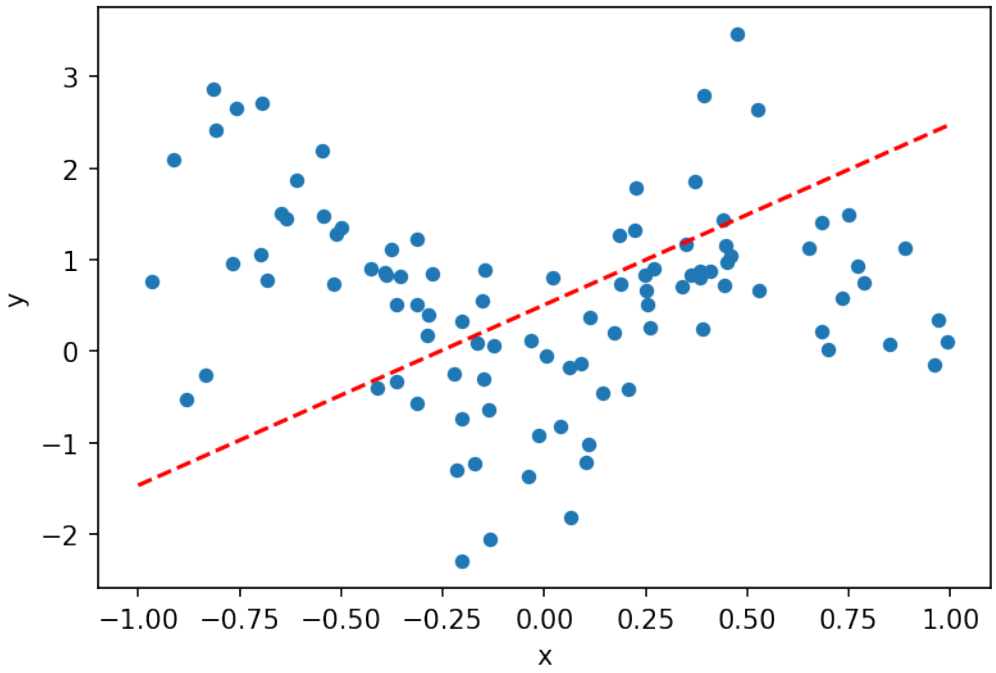
In order to make predictions we call `.predict()` on our trained model. We will make predictions for all values of x from -1 to 1 and plot the predicted values using matplotlib. Recall that

scikit-learn requires our input to have two dimensions, and so we use `np.newaxis` to add a new dimension to an existing object.

[89]:
```
xfit = np.linspace(-1,1)
yfit = model.predict(xfit[:, np.newaxis])

ax = data.plot.scatter("x", "y")
ax.plot(xfit, yfit, linestyle="--", color="red")
```

[89]:
```
[<matplotlib.lines.Line2D at 0x7fb7830da588>]
```



The model fit to the data has the following parameters:

[20]:
```
print(f"Model coefficient: {model.coef_[0]}")
print(f"Model intercept: {model.intercept_}")
```

```
Model coefficient: 1.9730953513408522
Model intercept: 0.5070848414292028
```

2.2 Kernel Ridge Regression

As mentioned above, it is possible to implement much more interesting models using scikit-learn. Below we will implement a kernel ridge regression using a Gaussian kernel (often called the RBF function/kernel). In this scenario the bandwidth (or length-scale) of the kernel is a hyperparameter to be tuned; we will choose this parameter by using a cross-validation procedure implemented in scikit-learn.

[75]:
```
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import GridSearchCV
from sklearn.gaussian_process.kernels import RBF

# generate data
rng = np.random.RandomState(123)
X = 10*rng.rand(100, 1)
y = 0.5 + np.sin(X).ravel() + rng.normal(0,0.8,100)

# fit a kernel ridge regression
## perform grid-search cv procedure to tune bandwidth
param_grid = {"kernel": [RBF(length_scale) for length_scale in np.logspace(-2,␣
 ↪5, 100)]}
krr = GridSearchCV(KernelRidge(), param_grid=param_grid, cv=5)
krr.fit(X, y)
```

[75]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=KernelRidge(alpha=1, coef0=1, degree=3, gamma=None,
                                   kernel='linear', kernel_params=None),
             iid='warn', n_jobs=None,
             param_grid={'kernel': [RBF(length_scale=0.01),
                                    RBF(length_scale=0.0118),
                                    RBF(length_scale=0.0138),
                                    RBF(length_scale=0.0163),
                                    RBF(length_scale=0.0192),
                                    RBF(length_scale=0.0226),
                                    RBF(length_sc...
                                    RBF(length_scale=0.221),
                                    RBF(length_scale=0.26),
                                    RBF(length_scale=0.305),
                                    RBF(length_scale=0.359),
                                    RBF(length_scale=0.423),
                                    RBF(length_scale=0.498),
                                    RBF(length_scale=0.586),
                                    RBF(length_scale=0.689),
                                    RBF(length_scale=0.811),
                                    RBF(length_scale=0.955),
                                    RBF(length_scale=1.12), ...]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

In the above code we have generated some non-linear data that we want to fit a model to. We have also specified a grid of length-scale parameters for the grid search procedure to minimize the CV score over, using `GridSearchCV`. In this case we have used `np.logscale` to generate possible parameter values on a logscale, as it's likely that the optimal length-scale is small (in this case less than two or three, but this is subjective). In this case we have chosen to implement 5-fold cross-validation by setting `cv=5`.

The grid-search procedure does not run until we provide the model with data using the

`fit` method. Having fit the model we can output the model parameters using the model's `best_params_` method explained in the scikit-learn documentation.

[87]:
```
krr.best_params_
```

[87]:
```
{'kernel': RBF(length_scale=1.83)}
```

Below we produce a plot of the predictions provided by a kernel ridge regression fit to the generated data.

[85]:
```
import matplotlib.pyplot as plt

# produce predictions for plot
X_fit = np.linspace(0,10, 1000)[:, None]
y_fit = krr.predict(X_fit)

# plot
plt.scatter(X, y, label="data", c="darkgrey")
plt.plot(X_fit, 0.5 + np.sin(X_fit), c = "black", label="true", lw=2)
plt.plot(X_fit, y_fit, label=f"KRR fit {krr.best_params_}", lw=2)
plt.legend(loc="best")
```

[85]:
```
<matplotlib.legend.Legend at 0x7fb782fc3eb8>
```