Ironhack Student Portal

preview.my.ironhack.com/lms/courses/course-

Lesson Goals

In this lesson you will learn all about object oriented programming and how it relates to data science.

Introduction

In this course so far, we have emphasized the importance of functional programming as the programming paradigm of choice in data science. However, we neglect to mention how object oriented programming can be helpful to us as data scientists. Object oriented programming is the practice of writing programs that are centered around objects. These objects contain methods and properties all bundled together.

You have been using objects all along without knowing. For example, a dataframe is an object. It contains many methods bundled into the dataframe like the `isna()` function or the `shape()` method.

The Linear Regression Example

Let's start with a relevant example. In the prework module we have learned about linear regression. While we did not introduce the algorithm in great detail, we learned that linear regression is a method to model the relationship between the predictor (or independent) variables and the response (or dependent variable). To find the best model between the predictor and response variables, we must optimize the sum of squared errors. This means that we find the difference between each observation and the predicted value, square these differences and add them up. The smallest such sum will give us the best linear model.

To find this optimal model, let's write our own regression function.

```
import numpy as np

def ordinary_least_squares(X, y):
    # This function returns the regression coefficients of a linear model
    # input: X - predictor variables and y - response variables
    # output: regression coefficients

    xtx = np.dot(X.T, X) ## x-transpose times x
    xtx_inv = np.linalg.inv(xtx) ## inverse of x-transpose times x
    xty = np.dot(X.T, y) ## x-transpose times y
    return np.dot(xtx_inv, xty)
```

Our function returns the regression coefficients. However, what if we want to make a prediction? In this case we will multiply the observed values by the coefficients. So to make a prediction, we will first call our ordinary least squares function and then multiply the observed values by the coefficients.

```
coefficients = ordinary_least_squares(X, y)
predicted = np.dot(X, coefficients)
```

Additionally, what if we want to find the r squared of the model? We will have to first compute the coefficients and then compute the r squared of the model.

To avoid doing these computations and storing them in variables and passing them around in functions, we are better off creating a linear regression object.

Creating an Object

We'll start off by creating a linear regression object called a class. A class has two main interesting features.

The `self` variable

`self` is a variable that is accessible to all other variables and methods inside the class. Using `self` inside an object helps us pass information around without having to recompute it every time.

The `__init__` function

The `__init__` function is typically the first function in an object. This function defines all the actions that need to be performed when we create a new object. The reason we have two underscores before and after the function name is to indicate that this function is internal to the object and should not be called from outside the object.

Now that we have defined a few basics, let's create a linear regression object. The naming convention for classes is upper camel case (this means that the first letter of every word in the name is capitalized).

```
class LinearReg:

    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept
        self.coefficients = None
        self.intercept = None
```

What we did in the code block above is set a default value of `True` for `fit_intercept`. This means that we would like to create a linear model with an intercept by default. We then assign this value to `self`. We also defined a number of attributes of `self` that do not have an initial value. We will compute them later, so for now they are initialized to `None`. After finding the regression model, we will be able to assign the values of the coefficients and the intercept.

Constructing the Class

Let's construct the entire class. We will use our `__init__` function as well as add more functions to fill in all values of `self`.

```
class LinearReg:

    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept
        self.coefficients = None
        self.intercept = None

    def fit(self, X, y):
        # This function returns the fitted linear model
        # input: X - predictor variables and y - response variables
        # output: regression coefficients

        xtx = np.dot(X.T, X) ## x-transpose times x
        xtx_inv = np.linalg.inv(xtx) ## inverse of x-transpose times x
        xty = np.dot(X.T, y) ## x-transpose times y
        coefficients = np.dot(xtx_inv, xty)

        if self.fit_intercept:
            self.intercept = coefficients[0]
            self.coefficients = coefficients[1:]
        else:
            self.intercept = 0
            self.coefficients = coefficients


    def predict(self, X):
        # This function returns the predicted values
        # input: array of dependent variables
        # output: predicted values
        if len(X.shape) == 1:
            X = X.reshape(-1,1)
            return self.intercept_ + np.dot(X, self.coef_)
```

Instantiating the Class

Using the `__init__` function, we can create an instance of our `LinearReg` class. We can make as many instances of our class as we like. We create an instance of our class like this:

```
linreg = LinearReg()
```

Since there are no required parameters to initialize, we leave the parentheses empty. However, we could initialize to `fit_intercept=False` if we wanted a model without an intercept.

We can now use `linreg` as an access point to the methods in the class. For example, here we use `fit` :

```
linreg.fit(X, y)
```

After fitting the model, we should have values populated for the coefficients and intercept since classes are not immutable.

Class Inheritance

We can use class inheritance when we would like to create a new class that will take on the attributes of another class. The new child class inherits all the methods of the parent class. However, we can override the methods of the parent class in the child class.

Summary

This lesson showed us that while we should prefer functional programming as data science, we have actually been using object oriented-programming all this time. Object oriented programming also very important to our work as data scientists. we have learned how to create classes. We learned about the `self` variable and the `__init__` function in classes. We have also learned how to assign values inside the class. We learned how to instantiate objects and how to use the methods inside them. This will provide us with a greater understanding of pandas and numpy as well as scikit-learn in the future.