Midterm Project

GRA4152 - Object Oriented Programming with Python

Fall 2022

Student numbers:

S2218689

S2213161

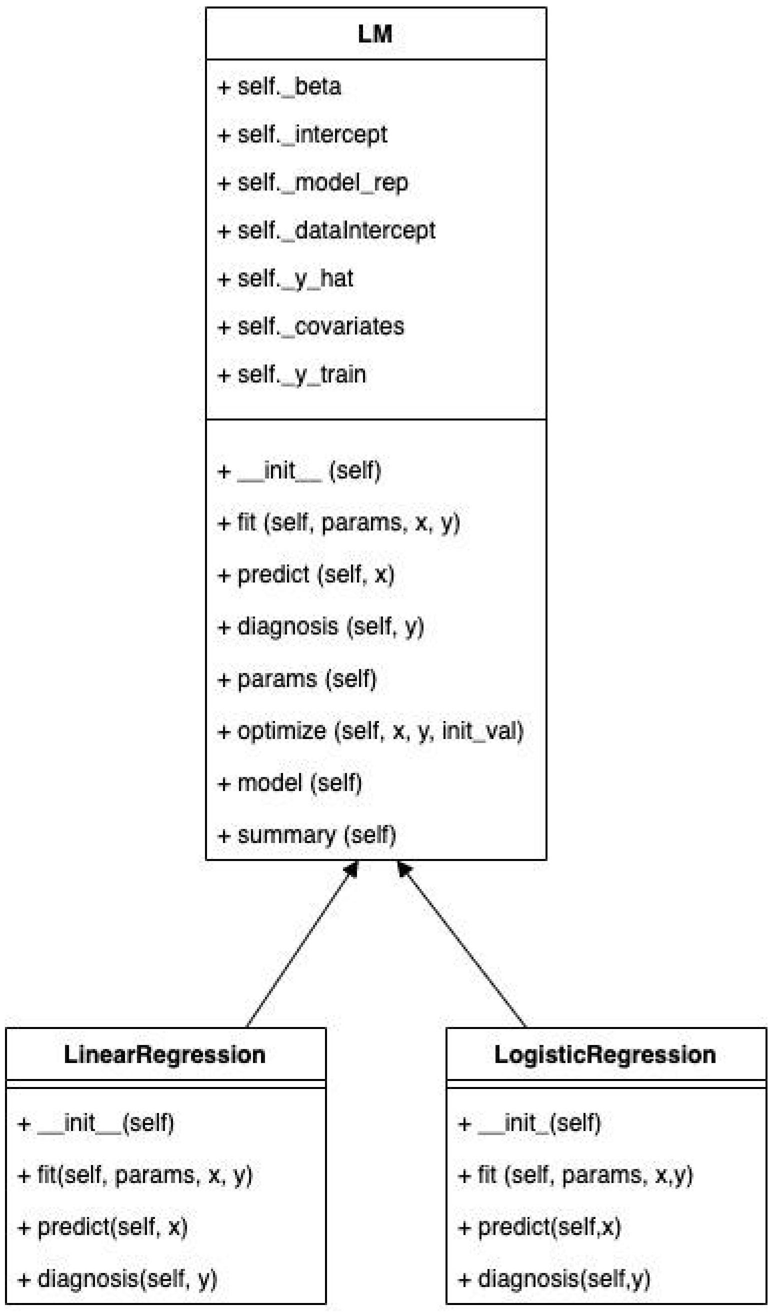
S2213401

1 Linear Model Classes

1.1) Python Classes

Please view linearmodels.py

1.2) UML and Public Interface



LM

The public interface of LM consists of the following methods

- __init__ (self)*

- linearModel(self, model_rep)
  Mutator method that takes in model_rep as an argument that specifies the independent and dependent variables we want the model to use in a string format.

- fit (self, params, x, y)
  Abstract method that raises NotImplementedError

- predict(self,x)
  Abstract method that raises NotImplementedError

- diagnosis(self,y)
  Abstract method that raises NotImplementedError

- params(self)
  Accesor method that returns the parameters that are optimized by scipy minimize function.

- optimize(self, x, y, dataIntercept, init_val)
  Mutator method that uses scipy minimize function to find the optimum parameters for our betas and intercept.
  dataIntercept is a boolean argument that is altered by the user to tell the method if the first column of the data contains integer 1 for calculating the intercept. Standard value is set False. This must be set to True by the user if the user has added intercept values to the data.
  init_val is an argument that sets the initial value for the scipy minimization of the parameters. standard value is set to 1

- model(self)
  Accessor method that returns the string model representation with dependent and independent variables.

- summary(self, y_test)
  summary method that prints the model summary with information about the model representation, fitted parameters and the scoring of the model.

*The instance variables in the constructor __init__(self) are:

- self._beta is the beta parameters for the model
- self._intercept is whether or not the model has an intercept added
- self._model_rep is the model as a string
- self._dataIntercept is whether or not the dataset has an intercept added to it
- self._y_hat is the predicted y
- self._covariates is a list of indexes for the x based on the model

LinearRegression

LinearRegression is a subclass om LM, which inherits:

- linearModel(self, model_rep)
- params(self)
- optimize(self, x, y, dataIntercept = False, init_val = 1)
- model(self), __repr__(self)
- summary(self, y_test)

The constructor inherits the LM constructor, and initializes the LogisticRegression.

The abstract methods from LM are implemented in LinearRegression as mutator methods:

- fit(self, params, x, y):
  Calculates and returns the total deviance of the model.
  x is a NumPy-array with the independent variables.
  y is a NumPy-array with the dependent variable.
  Comment: dot product has been used instead matmul, this means that there is no need for transposing the independent variables.

- predict(self, x):
  Estimates y based on the independent variables and returns the predicted y
  x is a NumPy-array with the independent variables.

Comment: dot product has been used instead matmul, this means that there is no need for transposing the independent variables.

- diagnosis(self, y):
  Calculates R^2 based on SST (sum of squares of residuals) and SSE (total sum of squares), and returns it, rounded to four decimals
  y is a NumPy-array with the dependent variable

LogisticRegression

LogisticRegression is a subclass of LM, which inherits:

- linearModel(self, model_rep)
- params(self)
- optimize(self, x, y, dataIntercept = False, init_val = 1)
- model(self), __repr__(self)
- summary(self, y_test).

The constructor inherits the LM constructor, and initializes the LogisticRegression.

The abstract methods from LM are implemented in LogisticRegression as mutator methods:

- fit(self, params, x, y):
  Calculates and returns the total deviance of the model.
  x is a NumPy-array with the independent variables.
  y is a NumPy-array with the dependent variable.
  Comment: dot product has been used instead matmul, this means that there is no need for transposing the independent variables.

- predict(self, x):
  Estimates y based on the independent variables, and returns the predicted y
  x is a NumPy-array with the independent variables.
  Comment: dot product has been used instead matmul, this means that there is no need for transposing the independent variables.

- diagnosis(self, y):
  Calculates the area under the ROC-curve based on y and the predicted y, and returns the area under the curve (AUC), rounded to four decimals

is a NumPy-array with the dependent variable

1.3 - 1.4) Regression Classes - Linear Models Class

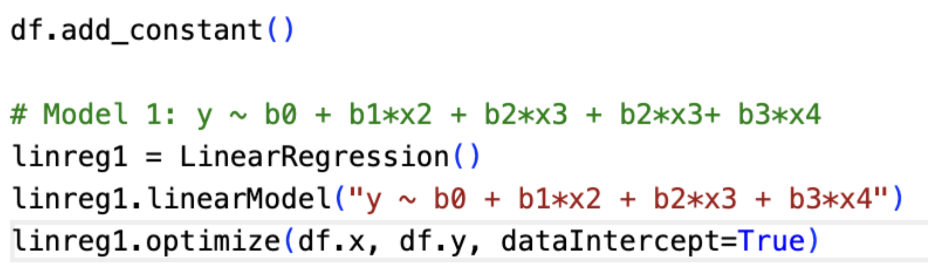
Please view LM, LinearRegression and LogisticRegression in linearmodels.py

The following five scenarios should compile:

- The dataset has an intercept constant and no model representation
- The dataset has an intercept constant and a model representation with an intercept
- The dataset has an intercept constant and a model representation without an intercept
- The dataset has an intercept constant and no model representation
- The dataset does not have an intercept constant and no model representation
- The dataset does not have an intercept constant and a model representation without an intercept

This means, if the dataIntercept parameter of the optimize method in the regression is set to False and add_constant() has been called an error will occur. However, dataIntercept can be set to True if either a constant has been added to the dataset prior to loading it in or add_constant() has been called first. An example of the latter is below:

Figure 1: Example of added constant and dataIntercept = True



```
df.add_constant()

# Model 1: y ~ b0 + b1*x2 + b2*x3 + b2*x3+ b3*x4
linreg1 = LinearRegression()
linreg1.linearModel("y ~ b0 + b1*x2 + b2*x3 + b3*x4")
linreg1.optimize(df.x, df.y, dataIntercept=True)
```

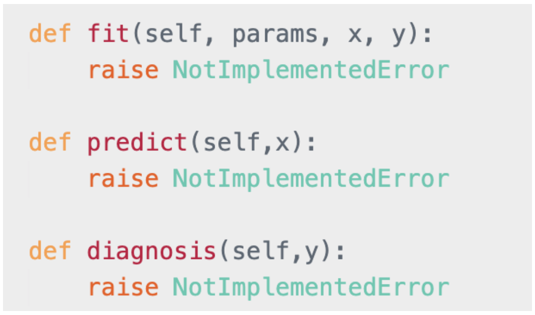
1.5) OOP Concepts

The class LM, which represents linear models, is a superclass. LinearRegression and LogisticRegression are subclasses of LM and therefore inherit from it. Even though the names might suggest otherwise, both linear and logistic regression are linear models. The latter is a linear model since its input combines one or more linear variables.

Since both subclasses are linear models, they have the same way of representing the model, for instance, “y ~ b0 + b1*x1”. This means these classes inherit LM’s methods linearModel, params, optimize, model, __repr__and summary. Hence, these have not been implemented in the subclasses. Additionally, both subclasses inherit from the constructor in their constructor as they both have a beta, intercept, model_rep, dataIntercept, y_hat and covariates, which are the instance variables of LM.

However, the methods fit, predict, and diagnosis of the two regression types are different. Therefore, the LM class has a NotImplementedError in the body of such methods. This makes sure both subclasses have an implementation of these. Since the methods fit, predict and diagnosis are not implemented in the LM class, these are abstract methods, which also makes LM an abstract class. This is shown in the figure below.

Figure 2: The abstract methods of LM



```
def fit(self, params, x, y):
    raise NotImplementedError

def predict(self,x):
    raise NotImplementedError

def diagnosis(self,y):
    raise NotImplementedError
```

Both subclasses have a class variable containing a string with the type of regression specified. For the LinearRegression class, this is: _modelName = “Linear Regression”. This means the summary method of LM can also print information about which linear model has been used. Hence, polymorphism, in addition to the class variables of the subclasses, is used to allow the summary method to specify the linear model in the print function. This is shown in the figure below.

Figure 3: summary method of LM and LinearRegression with class variable

```
    def summary(self, y_test):
        print(f"------------------------- Model summary {self.__class__._modelName} ------------------------- ")
        print("Model:", self)
        print("Fitted parameters:",self.params)
        print("Model accuracy:" ,self.diagnosis(y_test))
        print("-----------------------------------------------------------------------------------")


class LinearRegression(LM):
    _modelName = "Linear Regression"
```

2 DataSet Class

2.1 - 2.2) DataSet superclass and csvDataSet subclass

Please view DataSet and CsvDataSet in dataset.py.

In the DataSet class, self._hasIntercept is added as an instance variable. It is a boolean for whether an intercept is added to the dataset or not. If the add_constant() method is used, the self._hasIntercept is set to True. Additionally, to transpose data, transpose(self, data) has been added as a method as well, to separate the operation from the constructor.

3 DiagnosticPlot Class

3.1-3.2) diagnosticPlot class and Class architecture

Please view DiagnosticPlot in diagnostic_plot.py

The DiagnosticPlot class in task 3.1 has a constructor that takes in an object and checks if it is an instance of LogisticRegression or LinearRegression. If it is not, it raises an Exception. The class consists of one method called plot. This method takes two NumPy arrays comprised of the y values from the data set and the ŷ, representing the predicted y values from the model. The method plots y against ŷ in a scatter plot for objects of the class LinearRegression and plots the ROC curve for objects of the LogisticRegression class.

One could also implement a class for DiagnosticPlot without checking which class the object is an instance of in the constructor. But instead, only check in the plot method. This way, you would eliminate the need to check the object's instance twice. Another way to implement plots for the LinearRegression and LogisticRegression would be to drop the whole DiagnosticClass and instead implement a plot method in each of the linear model's classes. This would be an excellent way to implement more polymorphism since we could call the plot method on an object, not knowing if it is an instance of LogisticRegression or LinearRegression but still getting the correct plot.

4 Testing your Code

4.1) testerLogistic.py

Please view testerLogistic.py.

4.2 testerLinear.py

Please view testerLinear.py