Midterm Project

GRA-4152 OOP with Python

Autumn 2022

1. Linear model classes

1.1) Python Classes

We created LM (linear models) as a superclass.

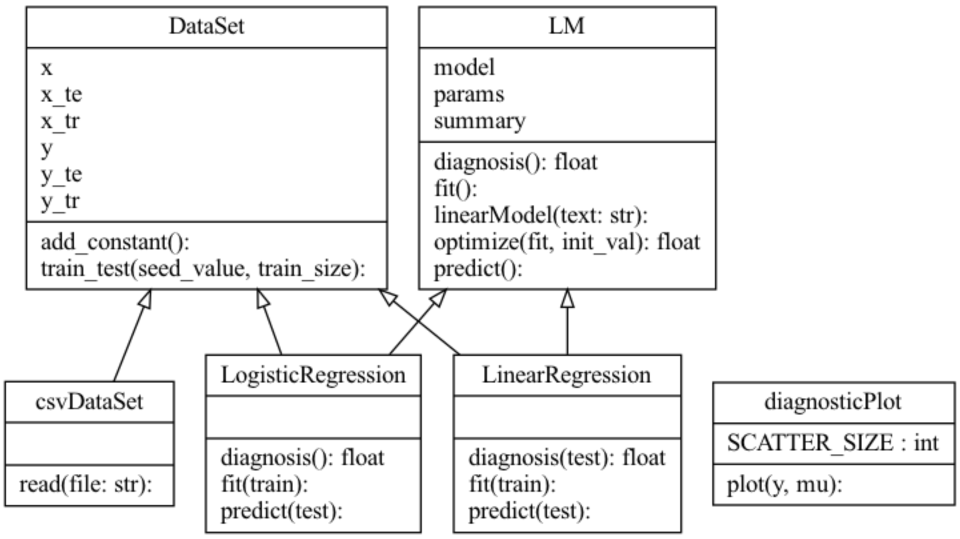
We chose to implement LinearRegression and LogisticRegression as subclasses of LM, meaning they both inherit from LM.

Class	
LM	Superclass
LinearRegression	Subclass
LogisticRegression	Subclass

1.2) UML and Public Interface

UML diagram

We used pylint to generate the UML diagram seen below.



Public interface

LM – Superclass

def __init__(self) -> None | constructor

def linearModel(self, text: str) -> np.ndarray | method for specifying the model
@param text the model as a string

def fit(self) -> None | method to fit the specified model | abstract method

def predict(self) -> None | method to predict using the fitted coefficients | abstract method

def params(self) -> np.ndarray | accessor method to get fitted parameters using @property

def optimize(self, fit, init_val = 1) -> float | method to minimize the objective
@param fit the objective function
@param init_val the initial guess for the coefficients

def model(self) -> str | accessor method to return the specified model using @property

def diagnosis(self) -> float | method to get the accuracy of the model | abstract method

def __repr__(self) -> str | method to return a string with fitted parameters

def summary(self) -> None | method to print a summary of the fitted model using @property

LinearRegression – Subclass

def __init__(self, x: np.ndarray, y: np.ndarray, scale_x = False, transposed = False) -> None: | constructor
@param x the explanatory variables
@param y the dependent variable
@param scale_x the scaling option, default is False
@param transposed True if data transposed, False if not

def fit(self, train = False) -> None: | method to fit the specified model
@param train whether to use training set or not

def predict(self, test = False) -> np.ndarray | method to predict using the fitted coefficients |
@param test whether to predict on full data, testing set, or training set

def diagnosis(self, test = False) -> float | method to get the accuracy of the model (R2) |
@param test False if training set, True if test set

LogisticRegression – Subclass

def __init__(self, x: np.ndarray, y: np.ndarray, scale_x = False, transposed = False) -> None: | constructor
@param x the explanatory variables
@param y the dependent variable
@param scale_x the scaling option, default is False
@param transposed True if data transposed, False if not

def fit(self, train = False) -> None: | method to fit the specified model
@param train whether to use training set or not

def predict(self, test = “all”) -> np.ndarray | method to predict using the fitted coefficients | @param test whether to predict on full data, testing set, or training set

def diagnosis(self, test = False) -> float | method to get the accuracy of the model (AUC) | @param test False if training set, True if test set

1.3) Regression Classes

We created LinearRegression and LogisticRegression as subclasses which inherits from both LM and DataSet. We decided to inherit DataSet as well (multiple inheritance), so it is easier to modify the data without creating another object just for data modification.

Fit: We decided to add an input to fit which decide if you want to fit train data, by default it fits all data. Fit calls the optimize method from LM to calculate the parameters with minimal deviance.

Predict: Here we also added an input to decide if you want to predict all data, train data or test data. The optimize method from LM creates the instance variable self._params which is used to calculate predictions. This method is initialized when fit is run. Predict and several other methods does not work before Fit has been run, thus we added tests to check this in Predict and other methods.

Diagnosis: Added an input to choose test or all data. Overrides an abstract method from LM and calculates R^2 and AUC respectively. For R^2 we chose to implement it like statsmodels, where we check if there is an intercept in the model. If there is not, we use uncentered R^2, which is calculated as: $Uncentered\ R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i y_i^2}$, and if there is one, we implemented it like your formula.

We did not add any additional methods, inherited methods from superclass were sufficient.

1.4) Linear Models Class

Instance variables:

_model: x and y stacked as np.ndarray

_fitted: result from optimizer

_params: coefficients minimizing the objective function

_text: specified model as a string

_indices: indices for coefficient numbers

_mui: predicted mean vector

Our linearModel method reads a string and returns the model (data) specified using some logic (if else statements combined with loops).

Convention is to write:
"dependent variable ~ intercept + coefficients*covariates"
y: dependent variable
b0: intercept
b1, b2, ..., bn: coefficients
x1, x2, ..., xn: covariates
Examples:

With intercept: "y ~ b0 + b1*x1 + b2*x2"

Without intercept: "y ~ b1*x3 + b2*x5"

linearModel can also read strings such as: “y ~ b0 + x1 + x2 + x3” correctly since it is also common to specify regressions without writing the betas for the x variables.

It returns a np.ndarray (the data), but also stores the indices/numbers of the coefficients (e.g. from b2*x5 or similarly x5 it stores the number 5), such that we could use it for the __repr__() method later and return a string with fitted parameters.

Our fit, predict, and diagnosis methods do nothing as they are simply abstract methods. In a development setting one can think of these as a reminder to develop the methods in the subclasses, but in a production setting one would probably want to remove them from the superclass.

As written in 1.2), the methods params and model are accessor methods for returning the fitted parameters and model specified.

The optimize method does numerical minimization with scipy. It contains an initial value with default 1 and creates a vector of the initial value repeated n times (number of x coefficients including the potential intercept). By polymorphism (the way we implemented the code) the scipy.optimize.minimize method knows which objective function to minimize (either the deviance for the linear regression model or the deviance for the logistic regression). It minimizes the objective function and returns the final function value (minimized deviance).

The __repr__ method contains a combination of logic and loops to obtain a string representation of the fitted model.

The summary method contains f-strings printing the specified model, fitted parameters, and accuracy of the model. To do this we invoked the other methods, model, __repr__(), diagnosis(). The accuracy is printed as R-squared if the object is LinearRegression and AUC if the object is LogisticRegression.

1.5) OOP Concepts

Inheritance: LinearRegression and LogisticRegression inherit everything from DataSet and LM. As they have many overlapping elements, having LM as a superclass that they can inherit from makes sense. As examples, they both inherit the linearModel and __repr__ methods, which they both need later. We decided it would be much simpler to have the subclasses inherit from DataSet as well, so we could modify the data without creating a new DataSet object each time, and then initialize a new Linear or Logistic Object.

csvDataSet subclass inherits from DataSet and is used to read csv files specifically.

Abstract methods: fit, predict and diagnosis are abstract methods in LM. This is since these 3 are entirely different in linear and logistic, thus it is abstract in LM so it can be overridden in the respective subclasses.

Polymorphism: As seen in our summary method in LM, the self variable calling the method diagnosis indicates that it “knows” which object it is being called from, and then use the corresponding method from that class.

Class Variables: The diagnosticPlot class has a class variable (constant): SCATTER_SIZE = 10. It contains a value for the size of the scatterplot markers which belongs to the class and not to an instance of the class. It is possible to mutate the value by writing diagnosticPlot.SCATTER_SIZE = newValue. This can be handy if a user wants to change the size of the markers since the plot method has no size parameter.

2. DataSet Class

2.1) DataSet superclass

The DataSet class is a superclass.
Takes data as numpy arrays.
Option to do MinMaxScaling on x.
@param x the explanatory variables
@param y the dependent variable
@param scale_x the scaling option, default is False

Constructor:

def __init__(self, x: np.ndarray, y: np.ndarray, scale_x = False, transposed = False) -> None:

By using the isinstance function we check if x and y are numpy arrays, and otherwise raise an exception. We create the instance variables x and y with underscores. The constructor then checks if the data is transposed or not, whether the user wants the x to be scaled or not, and then apply the appropriate transformation and scaling on the data.

The add_constant() method takes no arguments. It checks if there already is a vector of ones in the x data, and if not, it stacks a vector of ones to the x data, and if there is it raises an exception.

The next method is train_test(self, seed_value = None, train_size = 0.70) -> None:

It takes a seed and a training set size as arguments. The program raises an exception if not $0 < \text{training size} < 1$. The seed is set to the value of the seed specified. We then calculate

$n = \text{train\_size} * \text{length of data}$ and rounds it to the nearest integer, extract n random indices for training data, meaning $\text{length of data} - n$ random indices go to the testing data. We then used these indices to obtain training and test data for both x and y.

The rest of the methods are simple accessor methods using decorators: x (get x), y (get y), x_tr (get x training), y_tr (get y training), x_te (get x testing), y_te (get y testing).

2.2) csvDataSet subclass

A simple subclass with one method, read. It is used to read csv files, and takes a string as the file name argument, then returns a np.ndarray with numpy.float32 as default dtype.

3. DiagnosticPlot Class

3.1) diagnosticPlot class

The constructor takes the object as an argument and checks which class it is an instance of.

The plot method takes y (the dependent variable vector) and mu (the vector with predictions) as arguments. Based on the check in the constructor it creates a scatter plot if it is a LinearRegression and a ROC curve if it is a LogisticRegression.

3.2) Class architecture

If this class only is made to plot either a logistic or linear regression, we believe it rather should just have been an abstract method in LM, which is then overwritten in linear and logistic subclasses to plot scatter and ROC respectively.

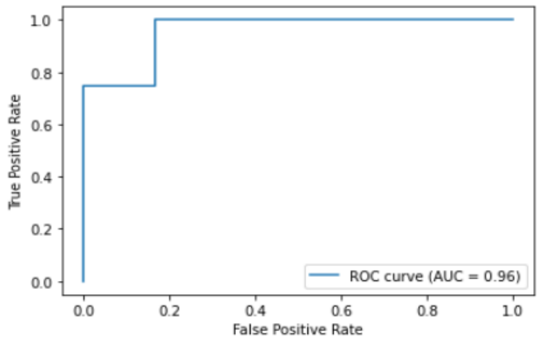
If diagnosticPlot should be used as a general plotting tool, it can be its own class. But then you should be able to choose the type of plot you want and add more plot options than scatter and ROC.

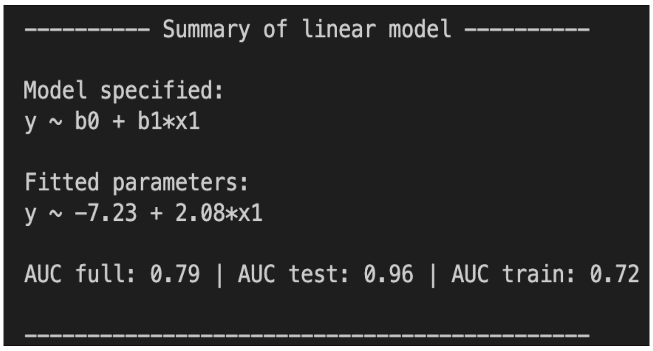
4. Testing your code

4.1) testerLogistic.py

We fitted the model on the training data, and then used those predictions for the plot on the test set. We calculated AUC on all parts of the data with training parameters.

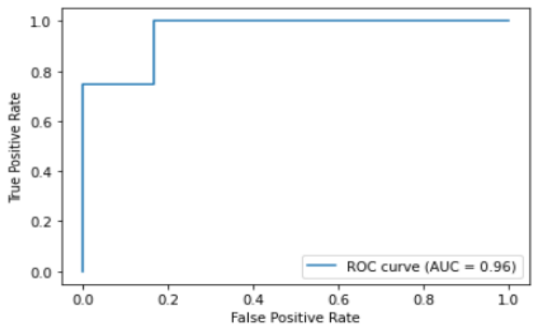
$y \sim b0 + b1 * x1$:

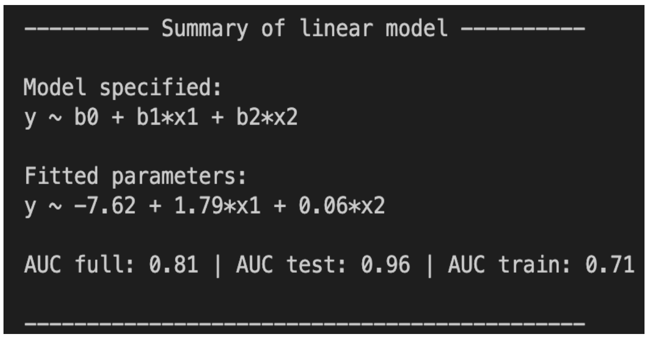




```
---------- Summary of linear model ----------

Model specified:
y ~ b0 + b1*x1

Fitted parameters:
y ~ -7.23 + 2.08*x1

AUC full: 0.79 | AUC test: 0.96 | AUC train: 0.72

---------------------------------------------
```

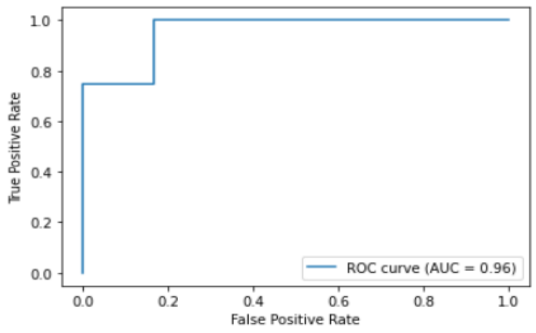
$y \sim b0 + b1 * x1 + b2 * x2$:

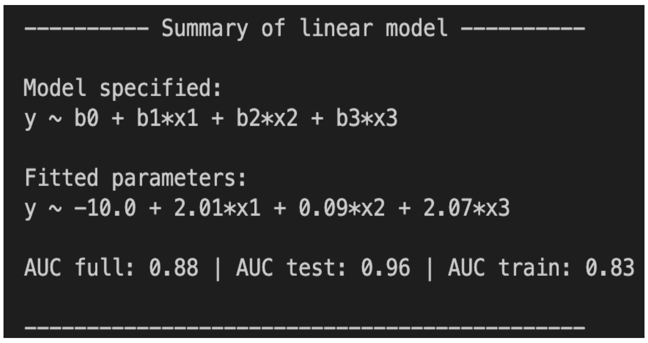




```
---------- Summary of linear model ----------

Model specified:
y ~ b0 + b1*x1 + b2*x2

Fitted parameters:
y ~ -7.62 + 1.79*x1 + 0.06*x2

AUC full: 0.81 | AUC test: 0.96 | AUC train: 0.71

---------------------------------------------
```

$y \sim b0 + \ b1 * x1 + \ b2 * x2 + b3 * x3$:



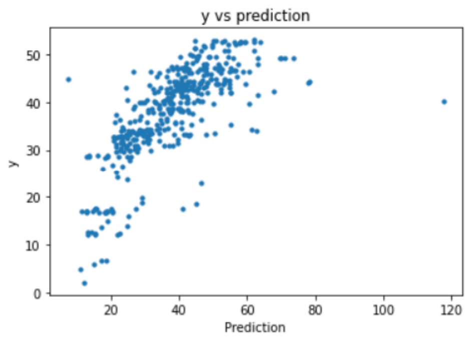


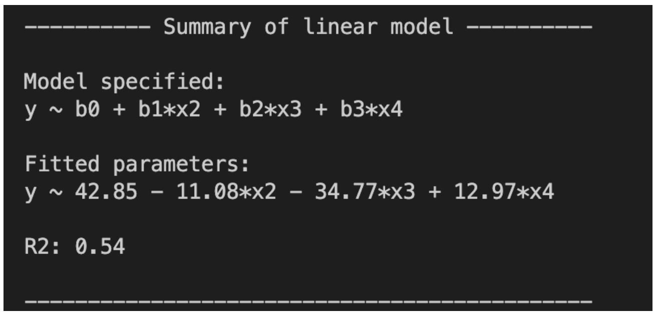
```
---------- Summary of linear model ----------

Model specified:
y ~ b0 + b1*x1 + b2*x2 + b3*x3

Fitted parameters:
y ~ -10.0 + 2.01*x1 + 0.09*x2 + 2.07*x3

AUC full: 0.88 | AUC test: 0.96 | AUC train: 0.83

---------------------------------------------
```

4.1) testerLinear.py

We fitted the model on the full data set with scaled covariates and used this to create the plots and calculate the R^2 for all models.

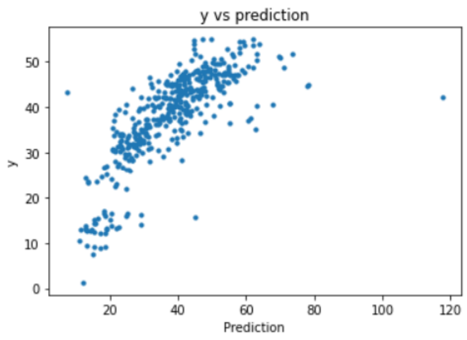
$y \sim b0 + b1 * x2 + b2 * x3 + b3 * x4$:

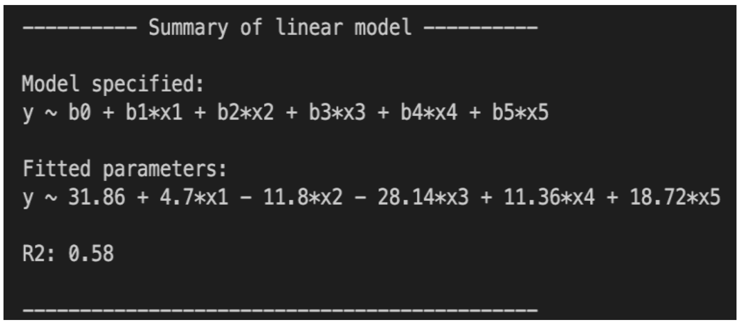




```
---------- Summary of linear model ----------

Model specified:
y ~ b0 + b1*x2 + b2*x3 + b3*x4

Fitted parameters:
y ~ 42.85 − 11.08*x2 − 34.77*x3 + 12.97*x4

R2: 0.54

---------------------------------------------
```

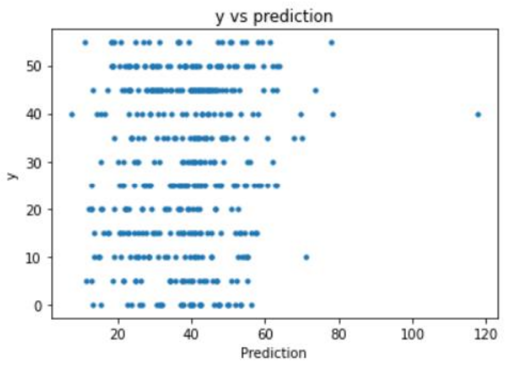
$y \sim b0 + \ b1 * x1 + b2 * x2 + b3 * x3 + b4 * x4 + b5 * x5$:

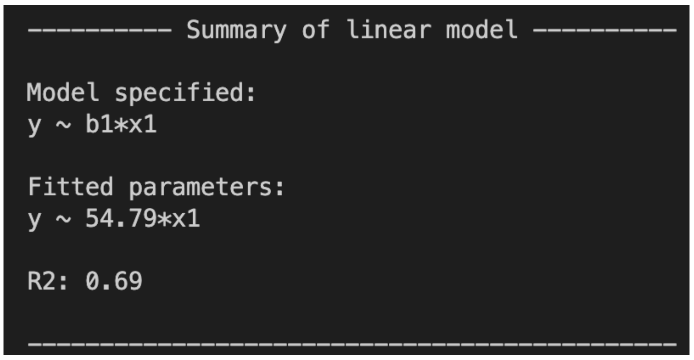




```
---------- Summary of linear model ----------

Model specified:
y ~ b0 + b1*x1 + b2*x2 + b3*x3 + b4*x4 + b5*x5

Fitted parameters:
y ~ 31.86 + 4.7*x1 − 11.8*x2 − 28.14*x3 + 11.36*x4 + 18.72*x5

R2: 0.58

---------------------------------------------
```

$y \sim b1 * x1$:





```
----------- Summary of linear model -----------

Model specified:
y ~ b1*x1

Fitted parameters:
y ~ 54.79*x1

R2: 0.69

-----------------------------------------------
```