OBJECT ORIENTED PROGRAMMING

MIDTERM PROJECT

Overview

In this project, we create a Linear Models superclass (LM) that provides a shell in which regression models can be estimated, used to predict, and create plots. The two regression models are defined as subclasses: Linear Regression and Logistic Regression. In this report we provide a concise overview of each class's implementation, their public interface, and testing results.

1. Linear Model Classes

1.1. Python Classes

First, we define the LM (Linear Models) superclass. Given that Linear Regression and Logistic Regression are types of Linear Models, their classes are defined as follows, allowing them to inherit methods defined in the LM superclass.

```
class LM : # Superclass
...
class LinearRegression(LM) : # Subclass
...
class LogisticRegression(LM) : # Subclass
...
```

1.2. UML and Public Interface

The first component of the Public Interface is the commentary used to describe the methods. An example is showed below.

```
#####
## Numerically minimizes the fit function/deviance
# @param init_val is the optional starting value for parameters (betas) in the optimization.
#
def optimize(self, init_val = 1) :
...
```

Appropriate descriptions are made, allowing other users to fully understand the code.

The second component of the public interface are the class's methods. The UML chart below gives a graphical representation of these, as well as whether they are inherited or overridden.

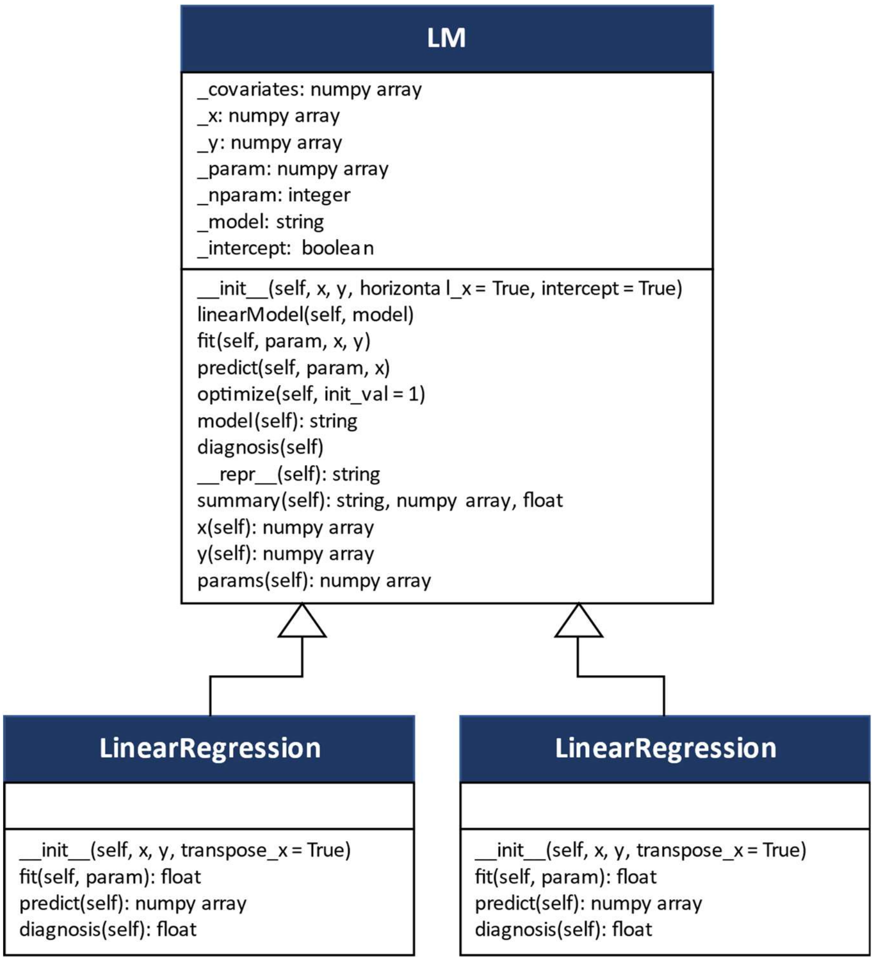


Figure 1. UML Diagram.

1.3. Regression Classes

In addition to inheriting methods from superclass Linear Models, the Linear Regression and Logistic Regression subclasses override each of the *fit, predict*, and *diagnosis* methods. These methods were purposely left as abstract methods in the superclass, as their unique definition determines each model type.

- ***fit:*** Computes the deviance of the model
- ***predict:*** Predicts μ given a set of covariates and parameters.
- ***diagnosis:*** Computes metrics to evaluate the model performance.

These three methods are polymorphic, i.e., they have the same name but execute different tasks.

1.3.1. Linear Regression

The LinearRegression subclass is composed by the following methods:

fit: Estimates the loss function (model deviance) given by:

$$dev = \sum_{i=1}^{n} (\boldsymbol{y}_i - \boldsymbol{x}_i^T \boldsymbol{\beta})^2.$$

```
def fit(self, params, x, y):
    # Estimate y_hat
    miu  = self.predict(params, x)
    # Deviance
    dev = (self.y - miu)**2
    return np.sum(dev)
```

predict: Receives a set of parameters and an x array to predict μ. The model prediction is given by the following expression:

$$\mu_i = \boldsymbol{x}_i^T \boldsymbol{\beta}.$$

```
def predict(self, params, x):
  # Predict using the estimated parameters (betas)
  miu = np.matmul(np.transpose(x), params)
  return miu
```

diagnosis: This method returns the R^2 of the linear regression:

$$R^2 = 1 - \frac{D}{D_0} = \frac{\sum_{i=1}^{n} (\boldsymbol{y}_i - \boldsymbol{x}_i^T \boldsymbol{\beta})^2}{\sum_{i=1}^{n} (\boldsymbol{y}_i - \bar{y})^2}$$

```
def diagnosis(self):
  D0 = np.sum((self.y - np.mean(self.y))**2)
  R2 = 1 - (self.fit(self.params, self.x, self.y) / D0)
  return round(R2,3)
```

1.3.2. Logistic Regression

The LogisticRegression subclass is composed by the following methods:

fit: Estimates the loss function (model deviance) given by:

$$dev = \sum_{i=1}^{n} [\log(1 + \exp(\boldsymbol{x}_i^T \boldsymbol{\beta})) - \boldsymbol{y}_i * \boldsymbol{x}_i^T \boldsymbol{\beta}]$$

```
def fit(self, params, x, y):
 # Calculate deviance
 t1 = np.log(1+np.exp(np.matmul(np.transpose(x), params)))
 t2 = np.matmul(np.transpose(x), params)
 dev = 0
 for i in range(len(y)):
   dev += t1[i]-y[i]*t2[i]
 return np.sum(dev)
```

predict: Receives a set of parameters and an x array to predict μ. The model prediction is given by the following expression:

$$\boldsymbol{\mu}_i = \frac{\exp\,(\boldsymbol{x}_i^T\boldsymbol{\beta})}{1+\exp\,(\boldsymbol{x}_i^T\boldsymbol{\beta})}$$

```
def predict(self, params, x):
   # Predict using the estimated parameters (betas)
   miu = np.exp(np.matmul(np.transpose(x), params))/(1+np.exp(np.matmul
   (np.transpose(x) , params)))
   return miu
```

diagnosis: This method returns the area under the ROC curve:

```
def diagnosis(self):
   auc = roc_auc_score(self.y, self.predict(self.params, self.x))
   return round(auc,3)
```

1.4. Linear Models Class

The LM superclass defines methods shared by all linear models and inherited by the Linear Regression and Logistic Regression classes.

Constructor: The constructor takes the data in the form of two *x* and *y* numpy arrays. In addition, we specify whether *x* contains row vectors and an intercept vector.

```
def __init__(self, x, y, horizontal_x = True, intercept = True):
...
```

linearModel: This method receives a string from the user and specifies the model to be estimated.

```
def linearModel(self, model):
...
```

Abstract Methods: The *fit, predict,* and *diagnosis* methods were defined as abstract methods to force their implementation in the subclass specifications.

```
def fit(self, param, x, y) :
   # To be specified
   raise NotImplementedError
def predict(self, param, x):
   # To be specified
   raise NotImplementedError
def diagnosis(self):
   # To be specified
   raise NotImplementedError
```

optimize: This method is responsible for minimizing the respective loss function. It uses the minimize method from the scipy.optimize library and stores the optimal parameters in an array.

```
def optimize(self, init_val = 1) :
   # Define starting parameters for the minimize function
   init_params = self._param + init_val
   # Minimize deviance
   results = minimize(self.fit, init_params, args = (self.x, self.y))
   # Store optimal parameters
   self._param = results["x"]
```

model: This method prints out the model with the current parameter estimates as a string.

```
def model(self):
...
```

summary: This method prints out the current model, the parameter estimates, and the diagnosis performance metric.

```
def summary(self):
...
```

__repr__: The *repr* method defines what is returned when the class object is printed. For the LM superclass, it prints a string with the model and the fitted parameters or a model with 0s as parameters in case these have not been fitted.

```
def __repr__(self):
...
```

Accessors: These methods allow the user to access the instance variables. For the LM superclass the instance variables include the following arrays: x, y, and params. To improve the usability and syntaxis of these methods, we have used the in-built decorator @property.

```
@property
def x(self):
…
@property
def y(self):
…
@property
def params(self):
…
```

1.5. OOP Concepts

As explained for the previous classes and their methods, we have used several OOP concepts.

Firstly, we used inheritance to define the relation between the LinearRegression and LogisticRegression subclasses with the LM superclass. This architecture allowed us to reuse methods in the definition of the subclasses.

Secondly, since the LinearRegression and LogisticRegression subclasses execute the same method differently, we used the concept of polymorphism. This allowed us to override the methods defined in the LM superclass and adapt them to the specific behavior of the subclasses.

Thirdly, we used abstract classes to force the implementation of these on a subclass level.

With regards to class variables, we didn't use any of these as we didn't find any static/constant variables or parameters that had to be private to ensure that methods from other classes wouldn't change their values.

2. DataSet Class

2.1. DataSet Superclass

The DataSet class builds objects using two numpy arrays (x and y) and allows the user to manipulate them so that they can be processed by a linear model. The following are the methods of this class:

Constructor: Allows the user to enter the x and y arrays and to specify whether the x array should be transposed (horizontal_x) using np.transpose or be scaled using the scaler.fit_transform method from the MinMaxScaler class in the scikitlearn library.

```
def __init__(self, x, y, horizontal_x = False, scale = False) :
...
```

add_constant: This method generates a vector of 1s and appends it in the first row of the x array using np.append.

```
def add_constant(self) :
...
```

train_test: Randomly splits the y and x arrays into train and test arrays in accordance with the trainSize parameter (percentage of train data) and the random seed specified by the user. The trainSize parameter has a default value of 0.7 (70%).

```
def train_test(self, trainSize = 0.7, randomSeed = 1234) :
...
```

Accessors and Setters: These methods allow the user to access or modify the instance variables. For the DataSet class the instance variables include the following arrays: x, y, x_tr, y_tr, x_te, and y_te. To improve the usability and syntaxis of these methods, we have used in-built decorators (@property). An example of an accessor and a setter is:

```
@property
def x(self) :
...
@x.setter
def x(self) :
...
```

2.2. csvDataSet Subclass

We extend the DataSet superclass to include a csvDataSet subclass that can process csv files. Since the main objective of this new class is to process csv files, we use the reader method from the csv library to process the file and obtain x and y numpy arrays. It is important to remember to specify in the constructor inputs whether the csv file contains headers, so that no row with relevant data is lost.

Once we have the x and y numpy arrays, they can be passed on to the superclass constructor together with the other parameters specified by the user. By doing this, we can reuse all the functionalities implemented in the superclass including those in the constructor.

```
def __init__(self, filename, horizontal_x = True, scale = False, headers = False) :
...
            # Use the Superclass constructor
            super().__init__(x, y, horizontal_x, scale)
```

3. DiagnosticPlot Class

3.1. DiagnosticPlot Class

The DiagnosticPlot class is a class whose objective is to generate diagnostic plots for linear models. As indicated in for the project, this class has only a constructor and a plot method.

The constructor of this class receives as input an instance of a LinearRegression or a LogisticRegression object and after validating that the input's class is correct it stores the model.

```
def __init__(self, linearModel) :
...
```

The plot method identifies the model's class and depending on this, it plots a scatterplot of y vs. μ (linear regression) or a ROC curve (logistic regression) using the RocCurveDisplay method from matplotlib.

```
def plot(self, y, miu) :
...
```

3.2. Class Architecture

The approach taken for the implementation of the DiagnosticPlot class in 3.1 has a big flaw. Every time the user wants to generate a diagnostic plot for an object, the class must identify the object type to generate the appropriate plot. Given the indications which allowed us to have only one class with two methods, this identification step was implemented using if statements.

This approach implies that every time we want to extend the plot method to a new linear model class, we must update the plot method adding a new if statement. This can become cumbersome and confusing the more classes we add.

A better approach to this problem would make use of the concept of polymorphism which makes code easily extensible. Thus, instead of identifying the object within the plot method, we could have created individual plot methods within each linear model subclass. In doing this, every time we create a new linear model class, we can create a specific plot method without having the need to modify our previous plot methods.

Another alternative in case we want a diagnosticPlot class that uses the same concept, would be to create a specific plot method for each linear model, e.g., plotLinearRegression and plotLogisticRegression.

4. Testing your Code

We generate the following programs to test all the classes and methods implemented.

4.1. testerLogistic.py

First, the spector dataset is loaded from the statsmodels package, then we load this into the DataSet class and perform a train-test split using a train set size of 70% and the seed value 12345.

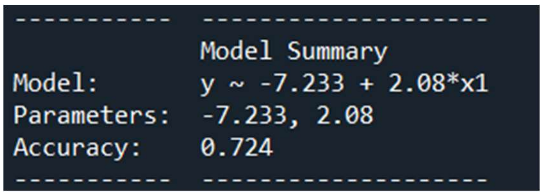
```
dataTest = DataSet(x, y, horizontal_x = False, scale = False)
dataTest.add_constant()
dataTest.train_test(trainSize = 0.7, randomSeed = 12345)
```

Having loaded and prepared the data, we now fit the first model on the training set:

$$y = b_0 + b_1 * x_1$$

```
logRegression_1 = LogisticRegression(dataTest.x_tr, dataTest.y_tr, horizontal_x = True)
logRegression_1.linearModel("y ~ b0 + b1*x1")
```

Then we optimize the parameters and call the summary method. The results obtained are:



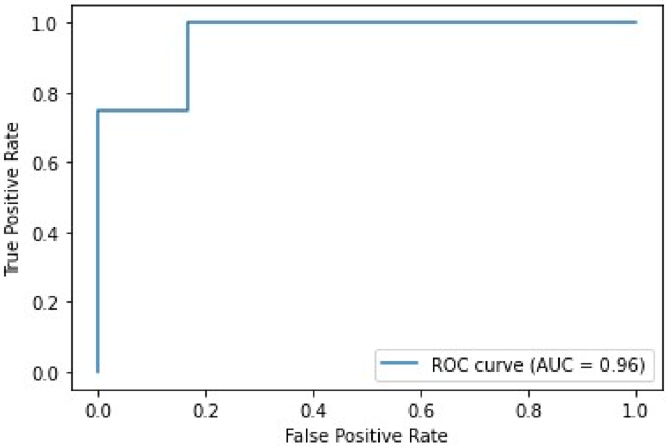
```
-----------  --------------------
             Model Summary
Model:       y ~ -7.233 + 2.08*x1
Parameters:  -7.233, 2.08
Accuracy:    0.724
-----------  --------------------
```

Figure 2. Summary of Logistic Regression Model 1.

Lastly, we use the plot method from the diagnosticPlot class and produce a ROC curve plot on the test set.

```
logRegression_1.optimize(init_val=1)

logRegression_1.summary()
```

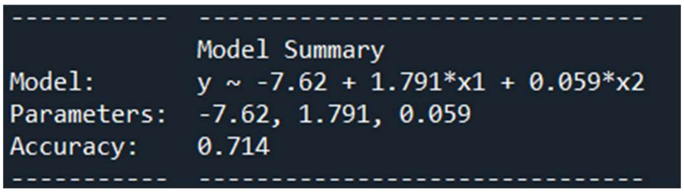


Graph 1. ROC curve of Model 1.

We now fit the second model on the training set:

$$y = b_0 + b_1 * x_1 + b_2 * x_2.$$

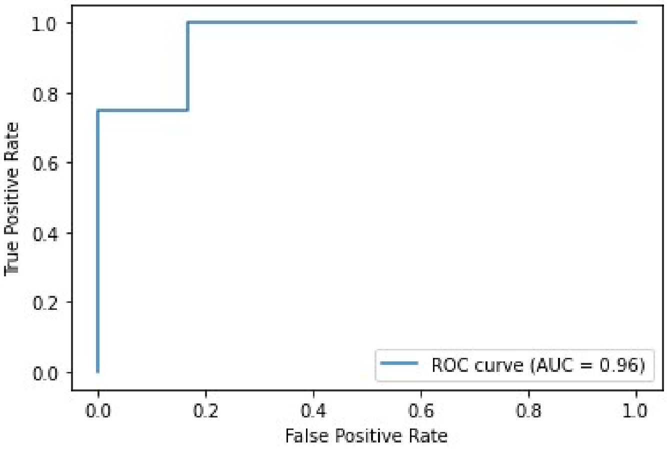
Then we optimize the parameters and call the summary method.



```
-----------  ------------------------------
             Model Summary
Model:       y ~ -7.62 + 1.791*x1 + 0.059*x2
Parameters:  -7.62, 1.791, 0.059
Accuracy:    0.714
-----------  ------------------------------
```

Figure 3. Summary of Logistic Regression Model 2.

Lastly, we use the plot method from the diagnosticPlot class and produce a ROC curve plot on the test set.

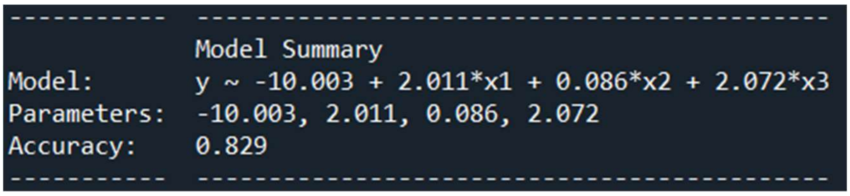


Graph 2. ROC curve of Model 2.

We now fit the third model on the training set.

$$y = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3$$

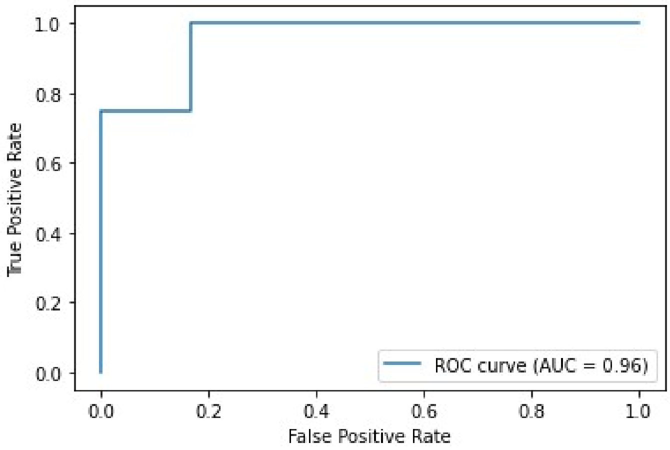
Then we optimize the parameters and call the summary method.



```
-----------  ------------------------------------------
             Model Summary
Model:       y ~ -10.003 + 2.011*x1 + 0.086*x2 + 2.072*x3
Parameters:  -10.003, 2.011, 0.086, 2.072
Accuracy:    0.829
-----------  ------------------------------------------
```

Figure 4. Summary of Logistic Regression Model 3.

Lastly, we use the plot method from the diagnosticPlot class and produce a ROC curve plot on the test set.



Graph 3. ROC curve of Model 3.

4.2. testerLinear.py

First, we load and scale the real_estate.csv dataset using the csvDataSet subclass. We add a constant using the add_constant method:

```
dataTest = csvDataSet("real_estate.csv", horizontal_x = False, scale = True, headers = False)
dataTest.add_constant()
```

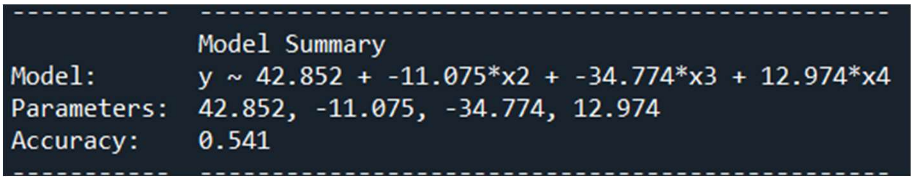
Having uploaded and transformed the dataset, we fit the first model:

$$y = b_0 + b_1 * x_2 + b_2 * x_3 + b_3 * x_4$$

```
linearRegression_1 = LinearRegression(dataTest.x, dataTest.y, horizontal_x = True)
linearRegression_1.linearModel("y ~ b0 + b1*x2 + b2*x3 + b3*x4")
```

Then we optimize the parameters and call the summary method.

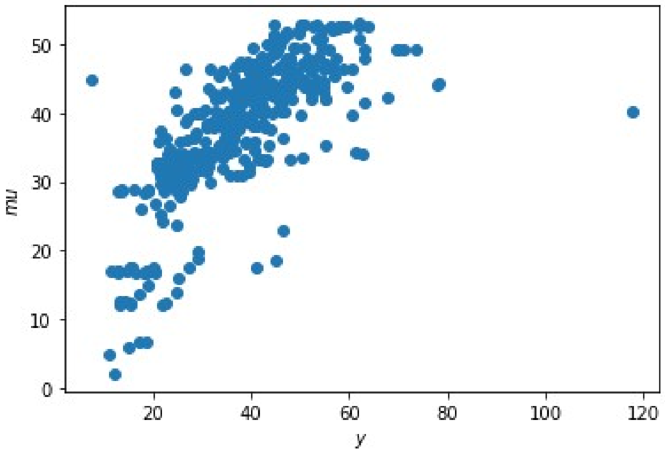
```
linearRegression_1.optimize()
linearRegression_1.summary()
```



```
-----------  -----------------------------------------------
             Model Summary
Model:       y ~ 42.852 + -11.075*x2 + -34.774*x3 + 12.974*x4
Parameters:  42.852, -11.075, -34.774, 12.974
Accuracy:    0.541
-----------  -----------------------------------------------
```

Figure 5. Summary of Linear Regression Model 1.

Lastly, we use the plot method from the diagnosticPlot class and produce a scatter plot of y vs. mu.

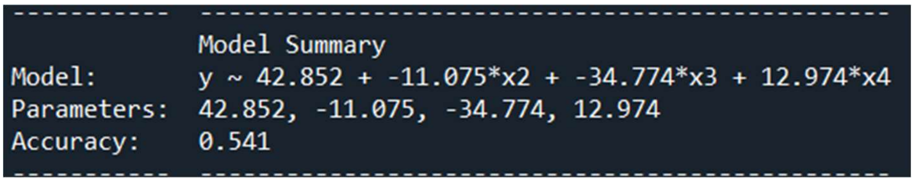


Graph 4. Scatterplot of Linear Regression Model 1.

We repeat the same procedure and fit the second model:

$$y = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3 + b_4 * x_4 + b_5 * x_5$$

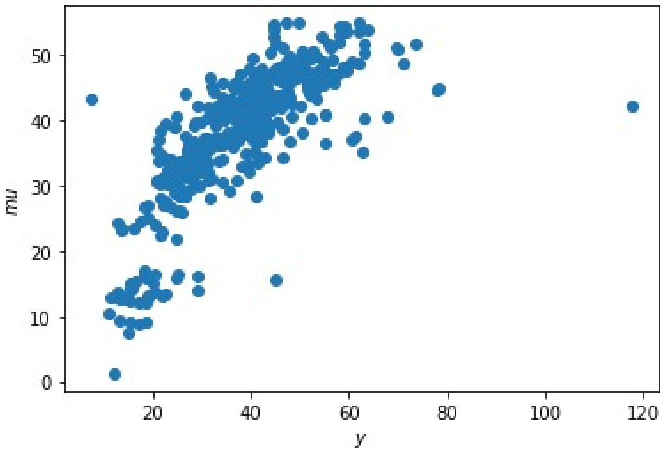
Then we optimize the parameters and call the summary method.



```
-----------  ------------------------------------------------
             Model Summary
Model:       y ~ 42.852 + -11.075*x2 + -34.774*x3 + 12.974*x4
Parameters:  42.852, -11.075, -34.774, 12.974
Accuracy:    0.541
-----------  ------------------------------------------------
```

Figure 6. Summary of Linear Regression Model 2.

Lastly, we use the plot method from the diagnosticPlot class and produce a scatter plot of y vs. mu.



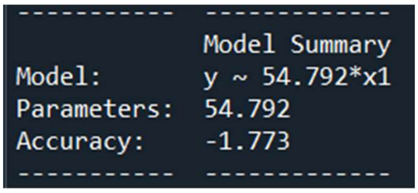
Graph 5. Scatterplot of Linear Regression Model 2.

Finally, we fit the third model:

$$y = \; b_1 * x_1$$

To do this, we call the csvDataSet subclass without adding an intercept. Then we optimize the parameters and call the summary method.

```
dataTest = csvDataSet("real_estate.csv", horizontal_x = False, scale = True, headers = False)
linearRegression_3 = LinearRegression(dataTest.x, dataTest.y, horizontal_x = True, intercept = False)
linearRegression_3.linearModel("y ~ b1*x1")
linearRegression_3.optimize()
linearRegression_3.summary()
```

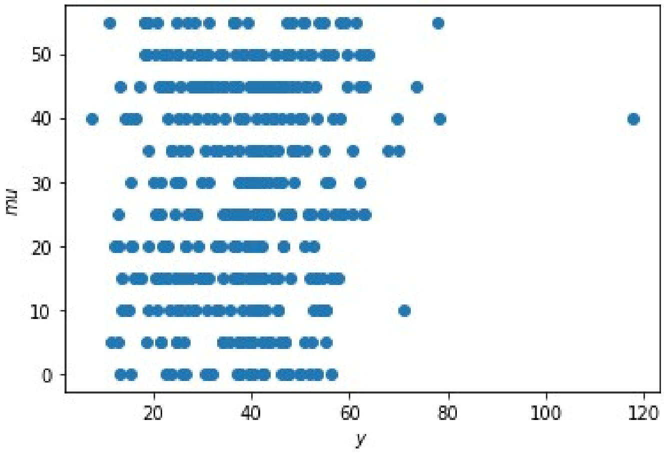


```
-----------  -------------
             Model Summary
Model:       y ~ 54.792*x1
Parameters:  54.792
Accuracy:    -1.773
-----------  -------------
```

Figure 7. Summary of Linear Regression Model 3.

Note: the R2 without an intercept falls outside the [0,1] bounds and should not be used for statistical inference.

Lastly, we use the plot method from the diagnosticPlot class and produce a scatter plot of y vs. mu.



Graph 6. Scatterplot of Linear Regression Model 3.