Midterm Project GRA 4152

1. Linear Model Class

1.1) Python Classes

In task 1.1, we create three classes: the superclass ***LM*** and the two subclasses ***LinearRegression*** and ***LogisticRegression***. These classes will be used to run two types of regressions depending on the invoked class and will be further described in sections 1.3 and 1.4.

1.2) UML and Public Interface

The visualization of the Linear Model Class using UML is presented below:

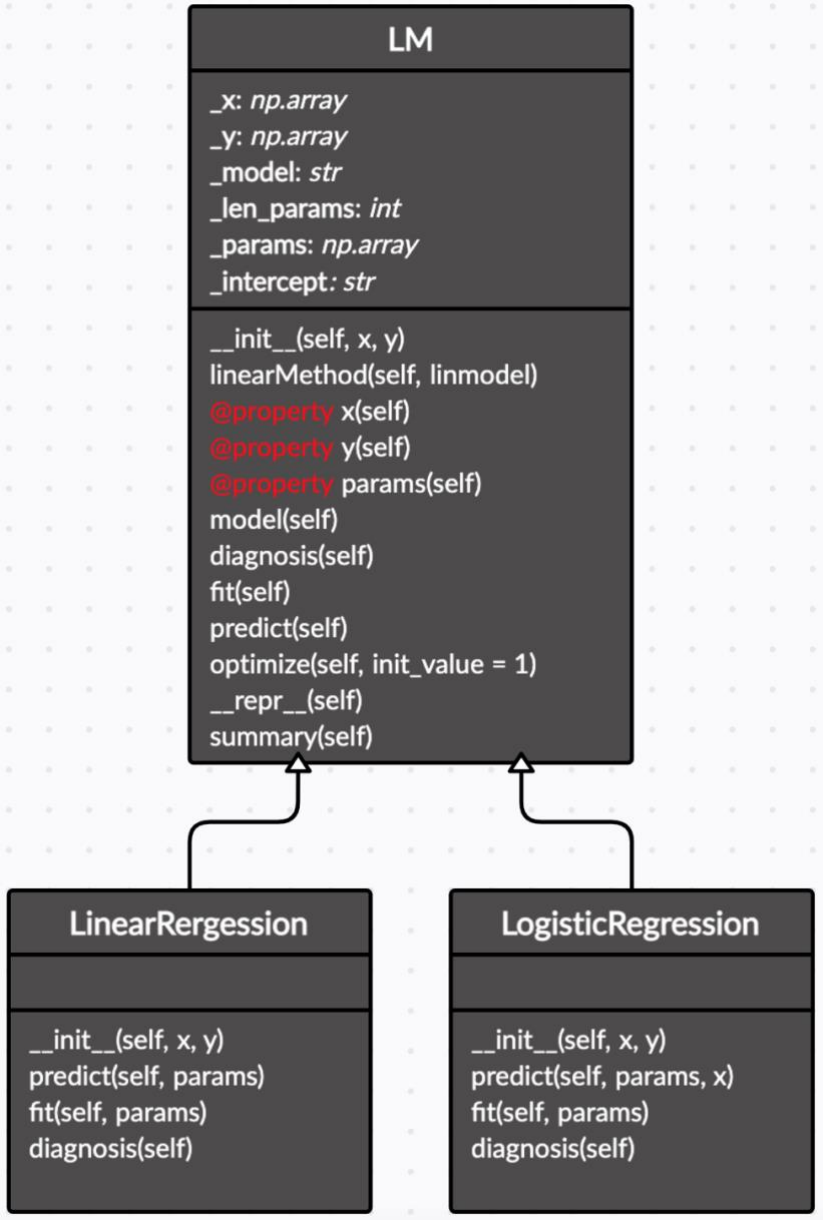

Figure 1.1 – *LM class visualization using UML.*

The public interfaces of each class are described in detail in the code. Here we provide only a partial example of the public interface of the LM superclass:

```
1.  #  This module defines the LM (Linear Models) superclass which works as a base for
    regression models
2.  #
3.
4.  class LM():
5.      ## Initializes with dependent variable(endogenous) y and independent(exogenous)
    variables x
6.      #  @param x numpy array of independent/covariates x specified by user
7.      #  @param y numpy array of dependent variable y specified by user
8.      #
9.      def __init__(self, x, y):
10.         ...
11.
12.     ## Construct a string which recognizes the dependent variables based on a
    string format.
13.     #  @param linmodel user specified string for regression model, e.g. "y ~ b0 +
    b1*x1"
14.     #
15.     def linearMethod(self, linmodel):
16.         ...
```

1.3) Regression Classes.

Two subclasses represent the regression classes: LinearRegression and LogisticRegression. In the first class, we have the following methods:

```
1.  ## Method for modelling estimate for mu (y_hat)
2.      #  @param params are beta parameters
3.      #  @return returns the modelled mu (y_hat)
4.      def predict(self, params):
5.          import numpy as np
6.          mu  = np.dot(np.transpose(self.x), params)
7.          return mu
```

The predict method is constructed such that we can use a different set of beta-coefficients (parameters) to predict/estimate μ. The same approach is used for the fit method. It is constructed to provide the user flexibility when calculating the model deviance by using a different set of parameters:

```
1.  ## Method for modelling the deviance
2.      #  @param params are beta parameters
3.      #  @return returns the model deviance
4.      def fit(self, params):
5.          import numpy as np
6.          mu  = self.predict(params)
7.          dev = (self.y - mu)**2
8.          return np.sum(dev)
```

The diagnosis method has the fit method (line 6 below) to calculate model deviance.

```
1.  ## Method for calculating model performance/accuracy
2.      #  @return returns R-Squared
3.      def diagnosis(self):
4.          import numpy as np
5.          D0 = np.sum((self.y - np.mean(self.y))**2)
6.          D  = self.fit(self.params)
7.          R2 = 1 - D/D0
8.          return R2
```

The LogisticRegression class has the same methods but is constructed differently:

```
1.  ## Method for modelling estimate for mu (y_hat)
2.      #  @param params are beta parameters
3.      #  @return returns the modelled mu (y_hat)
4.      def predict(self, params, x):
5.          import numpy as np
6.          mu = np.exp(np.dot(np.transpose(x), params)) / ( 1 +
    np.exp(np.dot(np.transpose(x), params))) # added x and replaced self.x with x
7.          return mu
```

There are some adjustments made to the predict method in this class. In addition to a different approach to calculate μ, we adjust the code such that x is now an input variable. This will enable us to use the predict method on the test data in task 4.1.
Otherwise, the methods fit, and diagnosis use the same logic as those methods in the LinearRegression class. The difference lies in the formulas specified in the guidance for the task paper:

```
1.  ## Method for modelling the deviance
2.      #  @param params are beta parameters
3.      #  @return returns the model deviance
4.      def fit(self, params):
5.          import numpy as np
6.          dev = np.log( 1 + np.exp(np.dot(np.transpose(self.x), params))) - self.y *
    np.dot(np.transpose(self.x), params)
7.          return np.sum(dev)
```

It is reasonable to measure the performance on a fitted model. Hence, we constructed the diagnosis method such that it uses x, y, and fitted parameters from the optimize method when the method was invoked.

```
1.  ## Method for calculating model performance/accuracy
2.      #  @return returns the Area Under the ROC Curve (auc)
3.      def diagnosis(self):
4.          from sklearn.metrics import roc_auc_score
5.          auc = roc_auc_score(self.y, self.predict(self.params, self.x))
6.          return auc
```

1.4) Linear Models Class.

LM is the superclass from which the subclasses described above inherits from. We use the class variable _intercept to define the string representation of the constant term used in linear models. The linearMethod method uses the variable to verify whether the constant is introduced in the model. The LM class has the following methods:

```
1.  ## Construct a string which recognizes the dependent variables based on a string
    format.
2.      #  @param linmodel user specified string for regression model, e.g. "y ~ b0 +
    b1*x1"
3.      #
4.      def linearMethod(self, linmodel):
5.          self._model = linmodel
6.
7.          # Splits string based on "x" and retrieve covariates corresponding to
    string
8.          model_stripped    = linmodel.split("x")[1:]
9.          for i in range(len(model_stripped)):
```

```
            model_stripped[i-1] = int(model_stripped[i-1].split(" ")[0])

        #intercept = "b0"
        if LM._intercept in self._model:
            model_stripped.insert(0, 0)

        # Update covariates according to specified model
        self._x = self._x[model_stripped, :]

        # Saves length of parameters for use in optimize method
        self._len_params = len(model_stripped)

```

The linearMethod method takes the string representation of the model specifications (linmodel) and examines the number of covariates (independent variables, x), and verifies whether the input model has an intercept (checks for "b0" in the string). The results are stored in an array which will be used as indices to extract the correct rows for covariates from the dataset and store the number of parameters we need to estimate.

We have three abstract methods (*fit, predict, diagnosis*) in this class which are coded in a similar fashion:

```
## Abstract class. Method for modelling the deviance
    #  @raise return to be specified in subclasses. Model specific
    def fit(self):
        raise NotImplementedError
```

Furthermore, we have several decorators (including additional accessor methods x and y).

```
## Accessor method. User can retrieve beta parameters
    #  @return numpy array of beta parameters
    #  @property is decorator
    @property
    def params(self):
        return self._params
```

We construct decorators in the accessor methods (x and y) to access the covariates and the dependent variable. These accessors can be used as inputs in other methods and for convenience when we want to access their values (e.g., examine covariates after specifying the string representation of the model).

The optimize method omits arguments for x and y as we want the method to run on the dependent variable y and the prepared covariates. The initial value for the parameters is set to one as default.

```
## Numerical minimization using spicy package
    #  @param init_val initial values for beta parameters (default set to 1)
    #  @return returns minimized beta parameters
    def optimize(self, init_val = 1):
        from scipy.optimize import minimize
        import numpy as np

        init_params  = np.repeat(init_val , self._len_params)
        results      = minimize(self.fit, init_params)
        self._params = results['x']
```

Furthermore, we use the *_len_params* variable obtained in the linearMethod method that specifies the number of parameters (beta-coefficients). Next, we create a method __repr__ to convert the outcome of different scenarios into a readable string.

```
## Prints out string with fitted parameters
    #  @return if model is not specified, it prints: "I am a LinearModel"
    #  @return if model is specified but not fitted it prints: specified model with 0s in all parameters
    #  @return if model is specified and fitted it prints: specified model with fitted parameters
    def __repr__(self):
        if self._model == "":
            return str("I am a LinearModel")
        elif hasattr(self, "params") == False and self._model != "":
            string_stripped    = self._model.split("b")[1:]
            "".join(string_stripped)
            for i in range(len(string_stripped)):
                string_stripped[i] = string_stripped[i].replace(string_stripped[i][0], "0")
            string = "".join(string_stripped)
            string = "".join(("y ~ ", string))
            return string
        else:
            import numpy as np
            string_stripped    = self._model.split("b")[1:]
            "".join(string_stripped)
            for i in range(len(string_stripped)):
                string_stripped[i] = string_stripped[i].replace(string_stripped[i][0], str(np.round(self._params[i], decimals = 4)))
            string = "".join(string_stripped)
            string = "".join(("y ~ ", string))
            return string

```

This method uses an if statement to check the different conditions. The first if-statement checks whether linearModel was invoked and “elif” to check whether the object has the params method (appears if the optimize method has been invoked). The last else condition will run if the model is specified, and coefficients are fitted. Inside the loop the string that specified the model is broken down into parts to retrieve the position of each beta coefficient. It then substitutes them with fitted coefficients or zeros and merges all elements to a string. The summary method prints out the string representation of the specified model and the results of optimization, namely, parameters and model accuracy (from diagnosis)

```
## Prints out the model specified in LM, the fitted parameters, and the model accuracy
    #
    def summary(self):
        import numpy as np
        print("                          LINEAR MODEL                          ")
        print("================================================================")
        print("Model Specified:   ", self._model)
        print("Fitted Parameters: ", np.round(self._params,4))
        print("Model Accuracy:    ", np.round(self.diagnosis(),4))
        print("================================================================")
```

1.5) OOP Concepts

Inheritance

Classes LinearRegression and LogisticRegression are subclasses that inherit from the superclass LM. While both of these subclasses share several methods (behave similarly), they are constructed as subclasses of LM using inheritance due to deviations in a few methods.

Abstract Methods

Methods diagnosis, fit, and predict are abstract methods in the LM superclass. These three methods are calculated differently for our subclasses and cannot be generalized for both regression models. Therefore, they are raised as "NotImplementedError."

Polymorphism

We use polymorphism on methods diagnosis, fit, and predict in subclasses LinearRegression and LogisticRegression. The method is created in the superclass but given that they are calculated differently for both subclasses, we extend the methods into our subclasses and calculate the model-specific methods. This is an example of a polymorphism concept.

Class Variables

We define one class variable in superclass LM. This is _intercept = "b0" which is used in linearMethod to recognize whether the model includes the intercept and needs to be used in the following analysis.

2. DataSet Class

2.1) DataSet superclass

The DataSet class is a superclass that shares behavior with its subclass csvDataSet. The only difference being on how data is imported. It is essential to verify that the class uses np.array as input. Therefore, we use the *isinstance* function to check this condition in the constructor.

```
1.  ##
2.  #  This module defines the DataSet class which examines the data and returns the
    correct format
3.  #
4.  class DataSet():
5.      ## Initializes with array x and array y.
6.      #  @param x numpy array of dependent x variables
7.      #  @param y numpy array of independent y variable
8.      #  @param transposed user input if data is already transposed (Default set to
    False)
9.      #  @param scaled user input if dependent x variables should be scaled or not
    (Default set to False)
10.     #  @return returns properly transposed data and scaled if specified
11.     def __init__(self, x, y, transposed = False ,scaled = False) :          # we
    don't use transposed as it here now any place
12.         import numpy as np
13.         if (not isinstance(x, np.ndarray) or
14.             not isinstance(y, np.ndarray)) :
15.             raise TypeError("Wrong format of x and y! Please, pass variables
    formatted as np.array.")
16.
17.         question_tr = input("Is the input data transposed?: (True/False):")
18.
19.         #
20.         if question_tr == "True":
21.             self._transposed = bool(True)
```

```
        elif question_tr == "False":
            self._transposed = bool(False)
        else:
            raise TypeError("Please, do answer True or False.")

        #
        if scaled == True:
            from sklearn.preprocessing import MinMaxScaler
            scaler = MinMaxScaler()

            if self._transposed == False:
                x = scaler.fit_transform(x)
            else:
                x = np.transpose(scaler.fit_transform(np.transpose(x)))

        #
        if self._transposed == False :
            self._x = np.transpose(x)
            self._y = np.transpose(y)
        elif self._transposed == True :
            self._x = x
            self._y = y
        else:
            raise TypeError("Please do answer True or False to the question.")
```

The method is coded to the ask user whether the input data is transposed. Furthermore, the data will not be scaled by default, and the user must specify “scaled = True” if the covariates should be scaled. This results in a dataset consisting of transposed (and scaled if set to True) covariates.

The method add_constant inserts a row of ones on top of the covariates dataset. The code assumes that the user will always invoke the method. The data will be manipulated correctly in the linearModel method of the LM class. If a user forgets to specify this, the code will still work correctly if the model does not include an intercept.

```
## Mutator method which appends a vector of 1s in the first row of x
    #  @return returns x with a new row of ones.
    def add_constant(self) :
        import numpy as np
        self._x = np.vstack([np.array(np.repeat(1.0, self._x.shape[1]), dtype =
np.float32), self._x])
```

Since we work with numpy arrays, we use the library “random” to define the indices to create a randomly sampled train and test data. We also set a seed (default set to 12345) to replicate results as described in the task:

```
## Method which splits the dataset into a train set and test set according to user
inputs
    #  @param train_set defines the weight to be given to the train set. Default is
set to 0.7 (70%)
    #  @param seed defines the random seed which lets the user replicate results.
Default is set to 12345
    def train_test(self, train_set = 0.7, seed = 12345) :
        import numpy as np
        import random
        random.seed(seed)

        ## Since random cannot be implied on np.array we create index list that
will be used to extract specific observations for training and test
```

```
        #
        index_tr = random.sample(list(range(self._x.shape[1])) , round(train_set*(self._x.shape[1])))
        index_te = np.where(np.in1d(list(range(self._x.shape[1])), index_tr, invert = True))[0]

        ## Split the data into test and train
        #
        self._xtr = self._x[:, index_tr]
        self._ytr = self._y[index_tr]
        self._xte = self._x[:, index_te]
        self._yte = self._y[index_te]
```

Lastly, we create 6 accessor methods using decorators to access the data samples. E.g., for the x train data:

```
## Accessor method. Retrieves test sample for x
    #  @return returns x test set as specified to be split in method train_test
    #  @property is decorator
    @property
    def x_te(self) :
        return self._xte
```

2.2) csvDataSet subclass

This subclass consists only of a constructor and uses the library "csv" to import the dataset. The constructor differs from its superclass in which the input is now a full dataset consisting of both independent and dependent variables:

```
##
#  This module defines the csvDataSet subclass which lets the user read a csv file and transform it to a numpy array
#
class csvDataSet(DataSet) :

    ## Initializes with name of csv-file
    #  @param path name of csv-file in ""
    #  @param transposed if x-array is in transposed format (default is set to False)
    #  @param scaled if x-array should be scaled or not (default is set to False)
    #  @param header if dataset includes header or not
    #  @return dataset in correct format and scaled if specified. x and y variables
    #  are retrieved with accessor methods in superclass (x, y, x_tr, y_tr, x_te, y_te)
    def __init__(self, path, transposed = False, scaled = False, header = None) :
        self._transposed = transposed
        import csv
        import numpy as np
        rows = []
        file = open(path)
        csvreader = csv.reader(file)
        if header == True:
            header = []
            header = next(csvreader)
        for row in csvreader:
                rows.append(row)
        self._data = np.array(rows, dtype = np.float32)
        file.close()

        if self._transposed == True:
            x = self._data[1:,:]
            y = self._data[0,:]
        else:
            x = self._data[:,1:]
```

```
33.             y = self._data[:,0]
34.
35.         super().__init__(x, y, transposed, scaled)
```

3. DiagnosticPlot Class

3.1) diagnosticPlot class

The diagnosticPlot only contains an initializer and a plot method. It initializes the class where the linearmodel object is recognized as linear regression or logistic regression. The code recognizes the type of object using isinstance function in python. The code snippet from the initializer is shown below:

```
1.  if isinstance(linearmodel, LinearRegression) == True:
2.      self._method = "LinearRegression"
3.      Print("Object is an instance of Linear Regression")
4.  elif ……..……
```

The plot method contains input variables "y" and "mu". "y" is a numpy array of the dependent variable y, whereas "mu" is numpy array of predicted μ variables. Based on the type of model which we fed into diagnosticPlot the methods plots different graphs.

```
1.  if self._method == "LinearRegression":
2.  .. .. ..
3.  elif self._method == "LogisticRegression":
4.  .. .. ..
```

Suppose the method recognizes the type of the model to be linear regression. In that case, it will construct a scatterplot of y against μ. If the model recognizes the model type as logistic regression, it will plot a ROC curve.

3.2) Class architecture

The approach in (3.1) requires us to use input variables y and μ in the plot method, where μ is calculated in the predict method in the LM subclasses LinearRegression and LogisticRegression. Instead, we could implement an abstract method for a plot method in LM. This would remove the need to identify the type of model. We could extend the plot method into the subclasses using polymorphism and compute a scatterplot for LinearRegression and ROC curve for LogisticRegression. Using this approach, we could keep y and μ as input variables. However, we could also plot directly by specifying the y and μ inside the method.

4. Testing your Code

4.1) testerLogistic.py

Approach

In testerLogistic.py we start by importing all our classes (LinearModels, diagnosticPlot, and DataSet). We then load the dataset spector from statsmodels and feed the dependent and independent variables into our DataSet class. Finally, we add a constant term to the independent variables and split our data into a test set (30%) and a train set (70%) using a random seed of 12345 to be able to replicate the results.

```
# Load dataset spector
spector = sm.datasets.spector.load_pandas()

# Feed into Dataset class
data = DataSet(spector.exog.values, spector.endog.values, scaled=False)
data.add_constant()

# Test and train sets
data.train_test(train_set = 0.7, seed = 12345)
```

The train and test samples for the dependent and independent variables are retrieved and used in the logistic regression. For all three model specifications, we use the same procedure. First, we run the LogisticRegression class with the given x_tr and y_tr (train sets) arrays. We then defined the model using linearMethod, and optimize our parameters. The code snippet for the first regression is shown below.

```
reg_1 = LogisticRegression(x_tr, y_tr)
reg_1.linearMethod("y ~ b0 + b1*x1")
reg_1.optimize()
```

For comparability we run the same model through statsmodel using the Logit function.

Results

```
                    LINEAR MODEL – Logistic Regression 1
=============================================================================================
Model Specified:    y ~ b0 + b1*x1
Fitted Parameters:  [-7.2335  2.0802]
Model Accuracy:     0.7238

Expected params: [-7.2335  2.0802]
=============================================================================================


                    LINEAR MODEL – Logistic Regression 2
=============================================================================================
Model Specified:    y ~ b0 + b1*x1 + b2*x2
Fitted Parameters:  [-7.6201  1.7909  0.0589]
Model Accuracy:     0.7143

Expected params: [-7.6201  1.7909  0.0589]
=============================================================================================


                    LINEAR MODEL – Logistic Regression 3
=============================================================================================
Model Specified:    y ~ b0 + b1*x1 + b2*x2 + b3*x3
Fitted Parameters:  [-10.003    2.0115   0.0857   2.0717]
Model Accuracy:     0.8286

Expected params: [-10.003    2.0115   0.0857   2.0717]
=============================================================================================
```

As we can observe, our class replicates the results given by statsmodels Logit.

Plots

The task specifically asks us to make a plot using μ predictions on the *test* data set. The code snippet for the third regression using all covariates looks like this:

```
1.  dp_3 = diagnosticPlot(reg_3)
2.  dp_3.plot(y_te, reg_3.predict(reg_3.params, x_te))
```

All three models yield the same ROC curve, given that the test set contains so few observations. Hence, the graph below represents the graph for all three models:

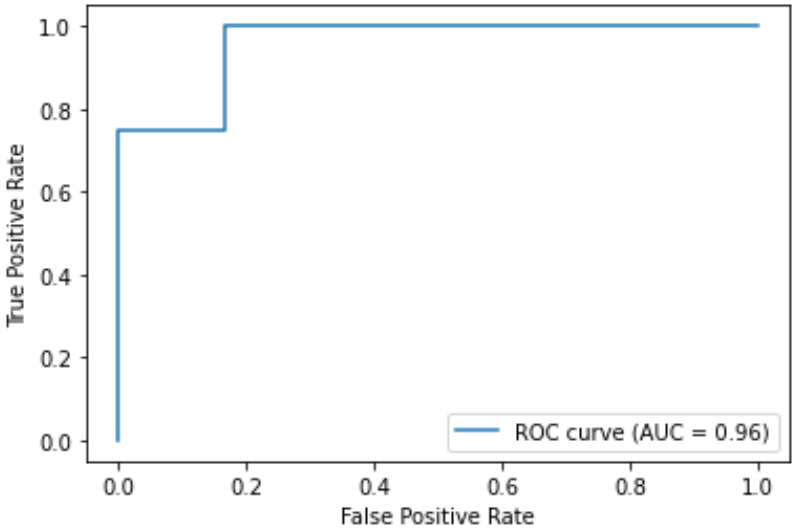

Figure 4.1 – *ROC curve from prediction test*

4.2) testerLinear.py

Approach

In testerLinear.py, we start by importing all our classes (LinearModels, diagnosticPlot, and DataSet). Then we import the data named "real_estate.csv" using our csvDataSet subclass (from DataSet). The dependent variables are scaled, and we add a constant term to the first row.

```
1.  data = csvDataSet("real_estate.csv", scaled=True)
2.  data.add_constant()
3.  x_array = data.x
4.  y_array = data.y
```

For all three model specifications we use the same procedure. First, we run the LinearRegression class with the given x and y arrays, specify the specific model, and optimize the parameters. Example for the first regression is shown below:

```
1.  reg_1 = LinearRegression(x_array, y_array)
2.  reg_1.linearMethod("y ~ b0 + b1*x2 + b2*x3 + b3*x4")
3.  reg_1.optimize()
```

Secondly, for comparability, we run the same model through statsmodel using the OLS (Ordinary Least Squares) function.

Results

```
                    LINEAR MODEL – Linear Regression 1
=============================================================================================
Model Specified:    y ~ b0 + b1*x2 + b2*x3 + b3*x4
Fitted Parameters:  [ 42.8515 -11.0751 -34.7741  12.9744]
Model Accuracy:     0.5411
```

```
Expected params: [ 42.8515 -11.0751 -34.7741  12.9744]
Expected R2: 0.5411
================================================================================================

                      LINEAR MODEL - Linear Regression 2
================================================================================================
Model Specified:    y ~ b0 + b1*x1 + b2*x2 + b3*x3 + b4*x4 + b5*x5
Fitted Parameters:  [ 31.8567   4.7028 -11.7988 -28.143   11.3613  18.7219]
Model Accuracy:     0.5823

Expected params: [ 31.8567   4.7028 -11.7988 -28.143   11.3613  18.7219]
Expected R2: 0.5823
================================================================================================

                      LINEAR MODEL - Linear Regression 3
================================================================================================
Model Specified:    y ~ b1*x1
Fitted Parameters:  [54.7921]
Model Accuracy:     -1.7732

Expected params: [54.7921]
Expected R2: 0.6852
================================================================================================
```

As we can observe, our class replicates the results given by statsmodels OLS. R-squared is computed without centering in statsmodels OLS since the model does not contain a constant. Given that we are not asked to compute an uncentered R-squared, our model accuracy will be wrong and differ from statsmodels OLS.

Plots

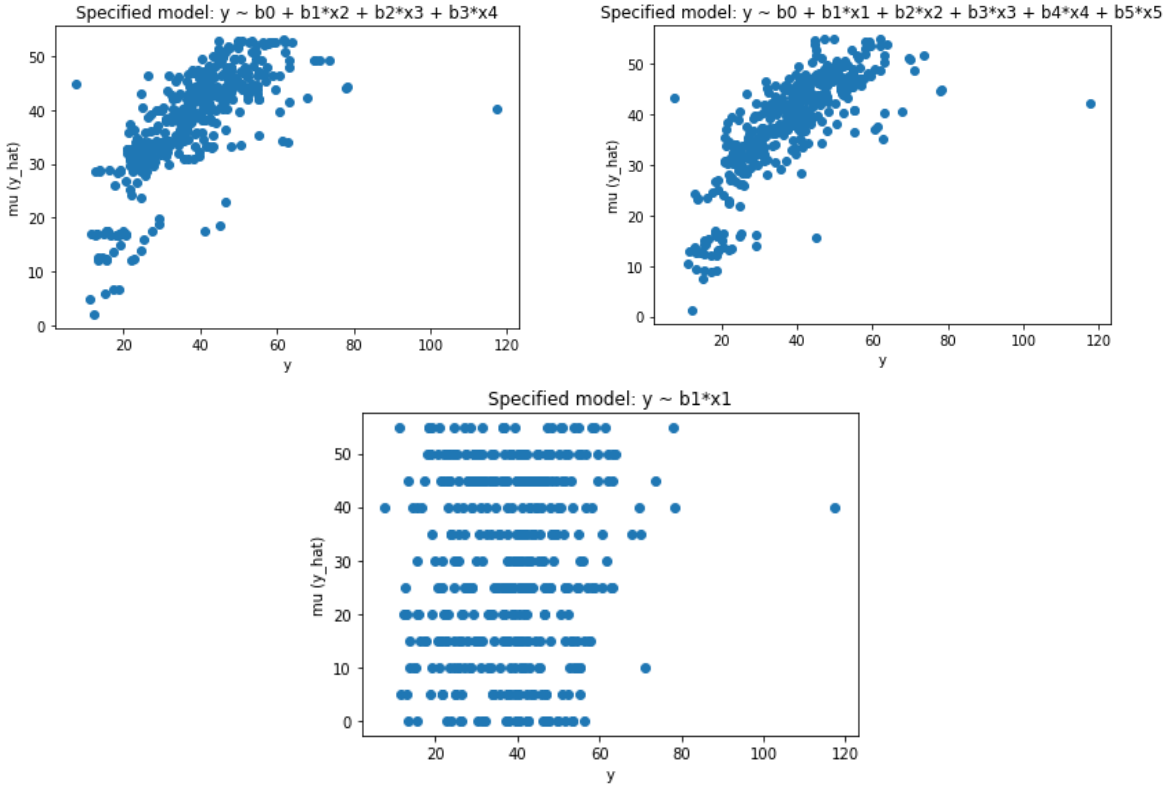

Figure 4.2 – *Scatter plots of y vs μ for different linear models*