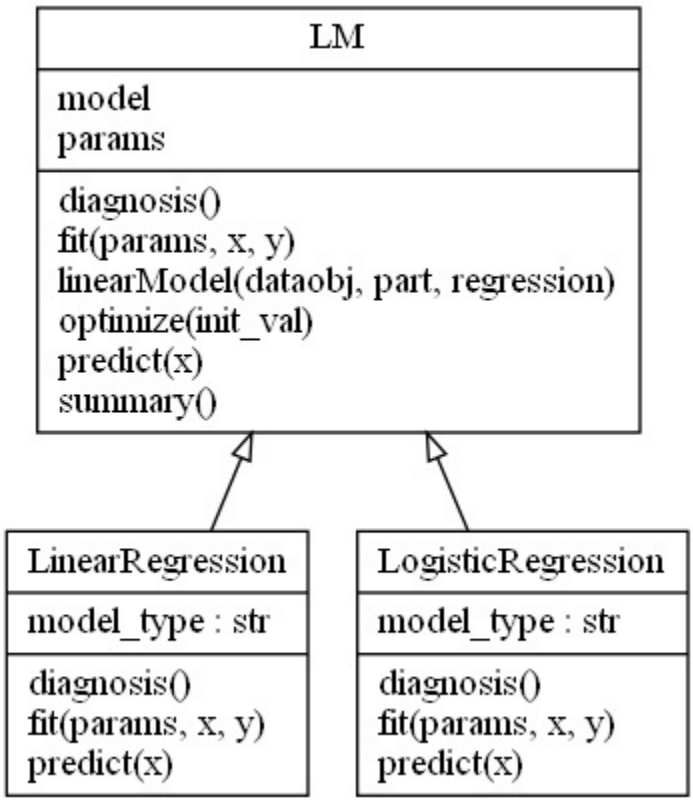
Introduction

This report is part of the group project exam in GRA4152 “Object Oriented Programming” for the 2022 fall semester at BI Norwegian Business School. The second part of the project is a collection of python programs. The aim of the project is to use object-oriented programming to make hierarchies of classes for the intent of linear regressions. The document will answer theory questions, briefly explain the code by parts, and more thoroughly explain programming choices where we have made our assumptions or where we’ve diverged from the exercise prompt.

1 Linear Model Classes

1.2) UML and Public Interface



The picture above is an overview of the inheritance hierarchy in the LM, LinearRegression, and LogisticRegression classes. Each table signifies a class. The first cell in each table shows the name of the class, the second shows class variables and other variables that can be accessed outside the class (either through methods or decorators), and the third cell shows the specific methods for the class. We can see that the LM class is the superclass, and the other two are subclasses that inherit from it.

Required packages to run all classes in the py file:
Numpy, Sklearn, matplotlib, Scipy

The packages need to be imported like this:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import MinMaxScaler
```

Public Interface

class LM:
The LM (Linear Model) class, is designed to be a superclass of the two regression subclass models. The LM classes take a DataSet object as input and use this for linear and logistic regressions.

```
LM.linearModel(dataobj, part, regression)
```

The linearModel method is used to specify which variables from which dataset we want to do the regression on. The method takes three inputs: "*dataobj*" is an object of the DataSet class, containing a dependent variable and an array of independent variables. "*part*" is used to specify whether the user wants to use the full set of observations for the regression (value "full") or only the training part (value "train"). "regression" takes as input a string form of a regression, for instance: "y ~ b0 + b1*x1 + b2*x4" if the user wants to do a regression of y on the first and fourth variable, including an intercept.

The methods *LM.fit(), LM.predict(),* and *LM.diagnosis(),* are all abstract methods and will be overridden in the subclasses because they need to be specified for each model. More on abstract classes in section 1.5.

```
LM.fit(params, x, y)
```

The fit method is only used within the class to calculate the model's deviance, a measure of the model's fit. The deviance is minimized in the *.optimize()* method to find the regression parameters. "*params*" refer to the regression parameters, whereas "*x*" and "*y*" refers to the independent and dependent variables, respectively.

```
LM.predict(x)
```

Estimates the dependent variable based on the model's fitted parameters and the input observations x.

```
LM.diagnosis()
```

The diagnosis method calculates a measure of model performance specific to the type of regression.

```
LM.params()
```

This method is an accessor method that returns the beta parameters if there are fitted parameters in the model. The property decorator is used so that this method can be invoked without the parentheses. The *LM.optimize()* needs to run before this method.

```
LM.optimize()
```

Fits the beta parameters through a minimization algorithm to find the optimal betas for the regression model. The input for instance method is a keyword argument 'init_val' which is the starting value for the minimization method. The default value is one. This instance method is the same as the *.fit()* method from the Scikit-learn library.

```
LM.model()
```

Returns the string representation of the model. The property decorator is used so that this method can be invoked without the parentheses. It would be the same as the 3rd input in the *LM.linearModel()* method, assuming this method was run before running *LM.model()*.

```
LM.__repr__()
```

The repr method is a special method that is invoked when the object (in this case the regression object) is printed through the native Python "print()". It will return a string representation of the specified model, with fitted parameters exchanged for the b's. If the parameters are not yet fitted, the estimates are regarded as "0". If a model has not yet been specified, it will print "I am a Linear Model".

```
LM.summary()
```

This method prints a summary of the object statistics in a clear format. It will print a string representation of the model specification input, the model performance measure, and a table of fitted parameters with the respective values.

```
class LinearRegression(LM)
```

Creates an empty instance of a linear regression class.
This instance object inherits from the LM model and has the method *LM.linearModel()* that needs to run before the instance methods below.

```
LinearRegression.fit(params, x, y)
```

Contains the formula that is used to minimize the deviance of a linear regression model. As mentioned in the public interface of the LM class, this instance method will not be run in public, it's just a helper method for *LM.optimize()*.

```
LinearRegression.predict(x)
```

Predicts the covariates based on the fitted parameters.
The inputs in this method are the covariates that will be fitted with the model's prediction for y. The covariates could be the whole dataset, the test set, or a different training set. The method returns an array for the predicted y values.

```
LinearRegression.diagnosis()
```

Returns the model's performance, specifically the R-squared score.

```
class LogisticRegression(LM)
```
Creates an instance of an empty logistic regression class. The class inherits from the LM class, as indicated by "LM" in the parentheses.

```
LogisticRegression.fit(params, x, y)
```
Contains the formula for minimizing the deviance in the logistic regression model.
In this module, the *LogisticRegression.fit()* method is used as a helper method for *LM.optimize()* function, and should not be used by the user.

```
LogisticRegression.predict(x)
```
Predicts the covariates based on the fitted parameters. The inputs are the covariates that will be fitted with the model's prediction for y. The covariates can be the test set, a different training set, or the whole dataset as in *LinearRegression.predict()* method.

```
LogisticRegression.diagnosis()
```
Returns the model's performance. This method uses the area under the ROC (Receiving Operating Characteristic Curve) score to evaluate how well the model performs on the dataset.

1.3) Regression classes

We decided to keep the classes as was intended, without any additional instance methods.
The instance variables in these classes are private because they should be internally used, and solely accessed through accessor methods for the user.

1.5) OOP Concepts

Inheritance:

LinearRegression and LogisticRegression inherit from LM, because it shares common functionalities, such as *LM.linearModel()* and *LM.optimize()*. Since the LinearRegression and

LogisticRegression models are more specific, they need more specialized instance methods such as .fit(), .predict(), and .diagnose(). This is because they are different models, and have different formulas to compute them.

Abstract methods:

Abstract methods are implemented in the LM superclass to make sure that we use the instance methods in the subclasses. We need to overwrite these methods because they are specific to the regression class. We could of course implement the method in the subclasses first, but since both subclasses need these methods it makes more sense to implement them as abstract methods in the superclass. If the methods are not overridden, we would get a 'NotImplementedError' when we tried to invoke the abstract methods.

Polymorphism:

The regression models have different formulas to minimize the model's deviance. As a consequence, a superclass method inherited from the regression models cannot be used. Since we have the same name for both fit methods in the subclass, polymorphism would be an efficient way to handle this class. When the *LM.optimize()* instance method runs, the method will find the instance method for their respective regression classes, through dynamic lookup, and the LM.fit() method will be overridden.

Class Variables:

Class variables have been implemented in the regression subclasses to label which type of model it is. This class variable will be useful for the *diagnosticPlot* class to plot the functions since there will be different plots needed for logistic regression and linear models.

```
class LinearRegression(LM):
      # Class variable used for model type checks
      model_type = "Linear Regression"
      …
class LogisticRegression(LM):
```

```
# Class variable used for model type checks
model_type = "Logistic Regression"
…
```

2 DataSet Class

The dependencies for both DataSet and csvDataSet subclass are in section 1.2

2.1) DataSet superclass

The exercise prompt says that the add_constant() method should add a row of ones to the top of the x-array. We assume that this means we should alter the dataset by adding a row of ones, rather than returning a copy of the dataset with ones added. Whenever the method is called, a row of constants should thus be added, but because this would make subsequent regressions on the specific dataset impossible, we added a “checker” in the add_constant() method that hinders the method from adding additional constant rows if one is already present. This “checker” is also utilized in the LM classes to ensure that we find the correct “positions” in the covariate array for the variables we are supposed to use.

```
firstRow = dataobj.x[0,:].reshape(1,dataobj.x.shape[1])
hasConstantRow = np.ptp(firstRow, axis=1) == 0
hasConstantRow &= np.all(firstRow != 0.0, axis=1)
```

The first line picks out the first row of the x-array. The second line returns True if the difference between the maximum and minimum value in the first row is 0. The third line checks that the values in the row are nonzero (as a row of zeros would not count as constants).

3 DiagnosticPlot Class

The dependencies in this diagnosticPlot class are the same as in section 1.2.

3.2) Class architecture

The exercise prompt specifies that the .plot() method should have the input parameters “y” and “mu”. This way, there is almost no relation between the LM classes and the diagnosticPlot class because we can specify whichever y or mu we want from outside any class. A clear improvement of the class would be to add a .diagnosticPlot() method to the LM classes directly. As the plot is supposed to show the model fit, there is no specific reason for the plot to be specified outside a regression class. This way, we wouldn’t have to make extra objects, and we could use the LM classes’ instance variables directly.

However, if we were to expand on regression models, a diagnosticPlot class could still be a good idea. Instead of using type tests as we do in the constructor now, we could make subclasses of the diagnosticPlot class (e.g., LogRegDiagnosticPlot) and utilize Python’s dynamic method lookup. When invoking the .plot() method on any diagnosticPlot object (or object of subclasses), polymorphism would ensure that the plot method for the specific object was carried out.

4 Testing

After making any programming module, it’s a good idea to make a program to test the code. The project asked us to write one file to test the LogisticRegression class and one to test the LinearRegression class. As part of these tester files, we also test the diagnosticPlot class and the DataSet classes. The dependencies in these tester files are the same as in section 1.2.

4.1) testerLogistic.py

The tester file starts by importing the “spector” dataset from the statsmodels library, and turns the data into a DataSet object. Some versions of statsmodels store spector as a Pandas dataframe, whereas our code only accepts NumPy arrays, so the user of the test file must manually switch out how to import the dataset, depending on the statsmodels version. We randomly split the dataset object into a training (70 %) and test (30 %) part by invoking the test_train method. To ensure reproducible results, we use seed 12345 for the split.

We make an instance of the LogisticRegression class called "*reg1*". Then the linear model is specified by the *LM.linearModel()* method, then fitting the parameters by the *.optimize()* method, and printing the summary by the *.summary()* method. The last step for the first regression is making an instance of the diagnosticPlot class with the fitted LinearRegression instance, "*reg1*" as input. Then the test program plots the true values of the test set against the model predictions. The process is repeated for "*reg2*" and "*reg3*".

4.2) testerLinear.py

First, the real_estate dataset is loaded from a CSV file and constructed as a csvDataSet object. We scale the independent variable data by setting the "scaled" parameter to True. Second, an instance of the LinearRegression class is made, called "*reg1*". Third, the model of the regression instance is specified (.linearModel()), optimized (.optimize()), and statistics are summarized (.summary()). Lastly, we make an instance of the diagnosticPlot class with "*reg1*" as an instance, and the true values of the dependent variable are plotted against the predictions.