Inheritance Hierarchy of Linear Models
Object Oriented Programming
June 2024

Linear Models

Linear models are defined by the linear predictor

$$f(\boldsymbol{x}) = \sum_i x_i \beta_i = \boldsymbol{x}^T \boldsymbol{\beta}, \tag{1}$$

where $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_\ell)^T$ and $\boldsymbol{x} = (x_1, x_2, \cdots, x_\ell)^T$ are vectors of unknown parameters and covariates respectively, and T denotes the transpose operation. For linear models with an intercept parameter, these vectors are $\boldsymbol{\beta} = (\beta_0, \beta_1, \cdots, \beta_\ell)^T$ and $\boldsymbol{x} = (1, x_1, x_2, \cdots, x_\ell)^T$, where β_0 is the intercept parameter and 1 is a constant variable.

Both linear and logistic regression are examples of linear models, but they have different *behavior* for fitting parameters and making predictions. When it comes to parameter fitting, a common approach is to minimize the model *deviance*. To make predictions, both models use the estimated μ parameter, which is defined differently in each model.

The deviance (dev) in the linear regression model is[1]

$$\text{dev} = \sum_{i=1}^{n} (y_i - \boldsymbol{x}_i^T \boldsymbol{\beta})^2, \tag{2}$$

where y is a continuous dependent variable. Further, the linear model assumes that

$$\mu_i = \boldsymbol{x}_i^T \boldsymbol{\beta} \tag{3}$$

is the μ parameter for the i'th vector $\boldsymbol{x}_i$.

On the other hand, the deviance in the the logistic regression model is[1]

$$\text{dev} = \sum_{i=1}^{n} [\log(1 + \exp(\boldsymbol{x}_i^T \boldsymbol{\beta})) - y_i \cdot \boldsymbol{x}_i^T \boldsymbol{\beta}], \tag{4}$$

where y is a binary variable (only takes values 0 or 1). The μ parameter for the i'th vector $\boldsymbol{x}_i$, in this case, is defined as

$$\mu_i = \frac{\exp(\boldsymbol{x}_i^T \boldsymbol{\beta})}{1 + \exp(\boldsymbol{x}_i^T \boldsymbol{\beta})}. \tag{5}$$

In both cases, all we have to do to estimate the unknown parameters $\boldsymbol{\beta}$ is to minimize Equation 2 and 4, which can be done in two steps in Python! See the appendix for a simple example on optimization with Python.

First, define a function for the model deviance, e.g,

```
def fit(params, x, y):
    import numpy as np
    ...

    return np.sum(...)
```

[1]The equality sign is actually incorrect, but for the sake of simplicity and given that it will not change anything, I allow myself to use it here.

where params is the vector $\boldsymbol{\beta}$, x and y are as in equation 2 and 4. Second, use the `spicy` in-build minimization algorithm `spicy.optimize.minimize` to find the parameters $\boldsymbol{\beta}$ that minimize the deviance, e.g.,

```
def optimize(x, y, init_val=1):
    from scipy.optimize import minimize
    import numpy as np

    init_params = np.repeat(init_val, len_params)
    results    = minimize(fit, init_params, args=(x, y))
    params     = results['x']
```

where `len_params` is the number of unknown parameters $\boldsymbol{\beta}$, `init_params` are initial values for the unknown parameters, and `params` is a list containing the estimated parameters $\hat{\boldsymbol{\beta}}$ that minimize the model deviance.

Note: some times you might get the message *Desired error not necessarily achieved due to precision loss*, which does not mean that the results are wrong.

Model Performance

We select models based on its performance, which we measure using different metrics. A common metric for linear models is the R-squared statistic, which is defined as

$$R^2 = 1 - \frac{D}{D_o}, \tag{6}$$

where $D_o = \sum_{i=1}^{n}(y_i - \bar{y})^2$, $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, and D is the deviance in equation 2.

On the other hand, the area under the ROC curve (AUC) is a common metric to measure model performance in logistic regressions. This metric is much harder to calculate by hand, but it is very easy to calculate with `sklearn`, e.g.

```
from sklearn.metrics import roc_auc_score

auc = roc_auc_score(y, mu)
```

where y is the dependent variable and mu is specified as in equation 5.

1 Linear Model Classes

You are hired by the Norwegian Computing Center and your first task is to develop a code for the linear regression and logistic regression models. Your boss says that you must use objected oriented programming (OOP) in Python for this task, and gives you the following instructions:

1.1) Python Classes: Create the following classes

- LM (Linear Models)
- LinearRegression
- LogisticRegression

1.2) Regression Classes: LinearRegression and LogisticRegression should at least contain the following methods

- fit: model deviance
- predict: model estimate for μ
- diagnosis: model performance

You can add more methods if needed/wished. If you do it, remember to provide a justification.

1.3) Linear Models Class: LM should at least contain the following methods

- linearModel: you should be able to specify a linear model by a string that specifies the dependent variable and covariates, i.e. "y $\sim$ b0 + b1*x1". Based on the string, this method selects the correct covariates from your data set. For example, suppose you have a data set with 10 covariates and that you want to fit the model "y $\sim$ b0 + b1*x1 + b2*x5", that is a model only using covariates 1 and 5 (plus the constant term).
- fit: model deviance
- predict: model estimate for μ.
- params: get fitted parameters $\boldsymbol{\beta}$.
- optimize: numerical minimization with `spicy`.
- model: get the string representation of the specified model.
- diagnosis: model performance.
- __rep__: prints out a string with fitted parameters, e.g. "y $\sim$ 4.32 + 12.05*x1". If the method is called before we specify a model in the method linearModel, it prints "I am a LinearModel. If parameters have not been fitted, it prints the specified model with 0s in all parameters, i.e. "y $\sim$ 0 + 0*x1".
- summary: prints out the model specified in linearModel, the fitted parameters, and the model accuracy.

In this class (LM), you are expected to use decorators in the accessor methods.

1.4) OOP Concepts: When coding the Python classes in 1.1) you must use the following concepts

- Inheritance
- Abstract methods
- Polymorphism
- Class variables

Explain in which class or method you use each concept and argue why.

2 DataSet Class

Your boss at the Norwegian Computing Center tells you that your code will be used by some junior economists that are not very familiar with Python programming. Hence, you are asked to code a class to handle different types of data sets. Specifically,

2.1) DataSet superclass: DataSet should at least contain the following methods

- __init__: verify that the arguments x and y are `numpy` arrays. This method also checks that x is a numpy array where the rows are covariates and columns observations. Otherwise, it transpose x to get the right format for equation 1. If `scaled` is specified, apply `sklearn.preprocession.MinMaxScaler` to the data x, e.g. `x = scaler.fit_transform(x)`.
- add_constant: append a vector of 1s in the first row of x.
- train_test: use the method `sample` from the library `random` to create a train and test data set. Commonly the train data set is 70% of the total number of observations. The remaining observations are used as the test data set. Make sure that you are able to set a seed value as an argument.
- x: returns x
- y: returns y
- x_tr: returns x_tr
- y_tr: returns y_tr
- x_te: returns x_te
- y_te: returns y_te

You should use decorators in the accessor methods.

2.2) csvDataSet subclass: csvDataSet should at least contain the following methods

- __init__: use the library `csv` to read a csv file and transform it into a `numpy` array. Use `numpy.float32` as default `dtype`.

3 DiagnosticPlot Class

Your boss come running into your office and quickly tells you to make a new class as follows:

3.1) diagnosticPlot class: diagnosticPlot only contains the following methods

- __init__(object): check if the object is an instance of LogisticRegression or LinearRegression
- plot(y, mu): for linear regressions make a scatter plot (`matplotlib.pyplot.scatter`) of y against mu. For logistic regression models, plot the ROC curve using this code

```
from sklearn import metrics
fpr, tpr, thresholds = metrics.roc_curve(y, mu)
roc_auc = metrics.auc(fpr, tpr)
display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc
    , estimator_name='ROC curve')
display.plot()
plt.show()
```

4 Testing your Code

You think that it is a good idea to write a program showing how to use all of the classes that you have coded for the junior economists.

4.1) testerLogistic.py

- Load the dataset `spector` from `statsmodels`. Then load the covariates `spector_data.exog.values` and dependent variable `spector_data.endog.values` into an object of the class `DataSet`.
- Create a train and test data set using the seed value 12345. The test data set shoudl be 30% of the data.
- Define and fit model parameters for the following models: "y $\sim$ b0 + b1*x1", "y $\sim$ b0 + b1*x1 + b2*x2", and "y $\sim$ b0 + b1*x1 + b2*x2 + b3*x3". Use the train data set.
- For each of the above models: i) print the `summary` method, and ii) make a plot with the `plot` method in the `dianosticPlot` class, using `mu` predictions for the test data set.

4.1) testerLinear.py

- Load the dataset `real_estate.csv` from itslearning and use the `csvDataSet` class to load it.
- Scale the data set.
- Define and fit model parameters for the following models: "y $\sim$ b0 + b1*x2 + b2*x3 + b3*x4", "y $\sim$ b0 + b1*x1 + b2*x2 + b3*x3 + b4*x4 + b5*x5", and "y $\sim$ b1*x1". Use the whole data set.
- For each of the above models: i) print the `summary` method, and ii) make a plot with the `plot` method in the `dianosticPlot` class.

You can use `sm.OLS` and `sm.Logit` from `statsmodels.api` to verify that your tester files are correct. Just remember to add the constant term in `statsmodels` when needed and to pass the correct covariates. If you see that `statsmodels` reports an uncentered R-squared, just ignore it.

Appendix

`scipy.optimize.minimize` uses the BFGS algorithm for solving unconstrained nonlinear optimization problems. On its most simple form, `scipy.optimize.minimize` must be invoke by passing a function $f(x)$ to be minimized and an initial guess for the optimal value of x.

Suppose you want to optimize the function $f(x) = x^2 + 5x$, i.e. find the x value that minimize the function. You know how to optimize such a simple function. That is, taking the first derivative wrt x, and solving for x at the point where the slope is zero. So, let's do it.

$$\begin{aligned} \frac{\partial}{\partial x} x^2 + 5x &= 2x + 5 \\ &\rightarrow \\ 0 &= 2x + 5 \\ x &= -2.5 \end{aligned}$$

Let's try `scipy.optimize.minimize` now, with $x = 5$ as our initial guess (you can try other values if you want).

```
1 import numpy as np
2 from scipy.optimize import minimize
3
4 def myfun(param):
5     y = (param**2) + 5*param
6     return y
7
8 minimize(myfun, 5)
```

The output of the above code is

```
1 fun: -6.249999999999875
2 hess_inv: array([[0.50000005]])
3 jac: array([-6.55651093e-07])
4 message: 'Optimization terminated successfully.'
5 nfev: 8
6 nit: 3
7 njev: 4
8 status: 0
9 success: True
10 x: array([-2.50000036])
```

Line 10 shows the x value that minimize the function. Pretty accurate, right?