Department of Physics
PH3010 Advanced Skills

Introduction to Multiple Regression

1. Introduction

This extension to the Machine Learning project involves application of the `scikit-learn` package to the problem of *multiple regression*. In the initial part of the project we discussed classification, where two types of objects are assigned a numerical label y, e.g. 0 and 1. In regression, this discrete label is replaced by a continuous variable.

Consider, for example, data on some *target variable* y such as the yield of some crop together with values of a number of *features* (sometimes called *predictors* or *explanatory variables*) $\mathbf{x} = (x_1, \ldots, x_n)$ such as yearly rainfall, soil pH, amount of fertilizer used, average temperature, etc. Suppose we have data consisting of feature vectors $\mathbf{x}_i$ and the corresponding values of the target variables y_i for $i = 1, \ldots, N$ instances or events. The idea of multiple regression is to use data to predict y given new values for the variables $\mathbf{x}$. This is analogous to curve fitting (i.e., simple regression), but with the usual scalar variable x replaced by a multidimensional vector $\mathbf{x}$. The term "multiple" here refers to the dimension of $\mathbf{x}$; if the target function is a vector, then this is called *multivariate* regression. Here we will consider only a scalar target variable y.

Some basic ideas behind multiple regression are presented in Sec. 2, using linear regression and the multilayer perceptron as examples. Section 3 provides some information on how these methods are implemented using the `scikit-learn` package and Sec. 4 gives some exercises for the project.

2. Basic ideas of multiple regression

A good introduction to multiple regression can be found in Ref. [1]. The main ideas can be seen as an extension of a least-squares fit of a curve to data points (x_i, y_i) with $i = 1, \ldots, N$ to the fit of a (hyper-)surface to measurements y_i, each carried out at a point $\mathbf{x} = (x_1, \ldots, x_n)$ in an n-dimensional space. For $n = 2$ this would correspond to fitting a surface, as illustrated in Fig. 1.

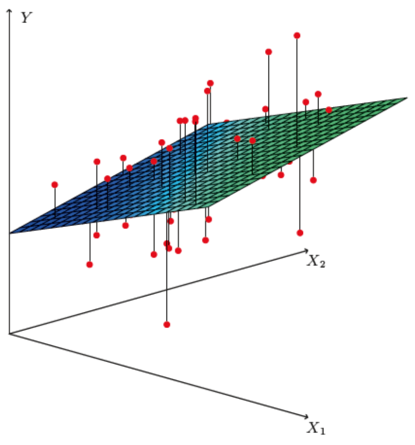

Figure 1: A function $f(x_1, x_2)$ represented as a surface fitted to data points $(x_{1,i}, x_{2,i}, y_i)$ (from Ref. [1]).

As in the case of curve fitting, we assume a model for the target variable $y = f(\mathbf{x}; \mathbf{w})$, where $\mathbf{x}$ is the vector of predictor variables and $\mathbf{w}$ is a set of parameters or "weights". To determine the optimal values of the weights, one minimizes a loss function $L(\mathbf{w})$ such as

$$L(\mathbf{w}) = \sum_{i=1}^{N} |y_i - f(\mathbf{x}_i; \mathbf{w})|^2 \ . \tag{1}$$

This is similar to the sum of squared residuals as one might use in a fit measurements $y_i \pm \sigma_i$ versus x_i, where σ_i is the standard deviation of y_i. In contrast to this case, however, the data here only include y_i and $\mathbf{x}_i$, and so the squared residuals are not multiplied by any other weighting factor involving the standard deviation. Equation 1 gives what is called a *quadratic loss function.* Other loss functions such as the mean absolute error or penalized residual sum of squares can also be used.

2.1. Linear regression

In linear regression, the function $f(\mathbf{x}; \mathbf{w})$ is a linear function of the n components of the feature vector, i.e.,

$$f(\mathbf{x}; \mathbf{w}) = w_0 + \sum_{i=1}^{n} w_i x_i \ . \tag{2}$$

Adjusting the weights to minimize the loss function is equivalent to fitting a hyperplane to the data as in Fig. 1.

2.2. Multilayer perceptron regression

In many problems, the y values may lie on some nonlinear surface and for such a case the function $f(\mathbf{x}; \mathbf{w})$ should therefore be nonlinear in $\mathbf{x}$. Here we consider use of the neural network or multilayer perceptron for this purpose.

A multilayer perceptron (MLP) for regression is similar to one used for classification. The network consists of functions or nodes $\varphi_i^{(k)}$ arranged in layers. The layer $k = 0$ consists of the input variables $\varphi_i^{(0)} = x_i$. The layers $k = 1$ through $K - 1$ are called *hidden* layers and $k = K$ represents the output node. The ith node in layer k depends on the nodes in the previous layer as

$$\varphi_i^{(k)} = h\left(w_{i0}^{(k)} + \sum_{j=1}^{n} w_{ij}^{(k)} \varphi_j^{(k-1)} \right) \ , \tag{3}$$

where $i = 1, \ldots, m^{(k)}$ and h is the activation function, e.g., a logistic sigmoid, tanh, or relu function. For classification, the nodes of the final hidden layer are used again as the argument the activation function. For regression, one simply leaves off the final activation function. That is, the final output is given by

$$f(\mathbf{x}; \mathbf{w}) = w_0^{(K)} + \sum_{j=1}^{n} w_j^{(K)} \varphi_j^{(K-1)} \ , \tag{4}$$

where the $\varphi_j^{(K-1)}$ are the nodes of the last hidden layer.

3. Example of multiple regression using `scikit-learn`

In this section we present an example to illustrate the regression problem outlined above. First, events are generated according to a Monte Carlo model with values E representing the energy of a high-energy particle such as a proton. The energies represent the target values and thus play the role of the variable y. The particle enters a calorimeter (a detector that measures energy) where it creates a shower of other particles. The goal is to estimate the incident energy given the signals measured by the calorimeter.

The calorimeter consists of three layers, which give measured signal values s_1, s_2 and s_3, which are proportional to the energy deposited in each layer. Some fraction of the energy leaks through and is undetected; this fraction increases with incident energy. The detector also measures a quantity η that is related to the angle of the incident particle relative to the detector ($\eta = -\ln \tan(\theta/2)$ is called the *pseudorapidity*), as illustrated in Fig. 2.

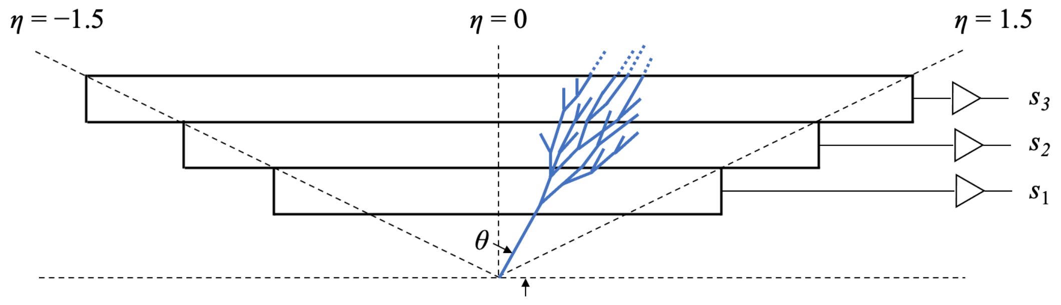

Figure 2: Schematic illustration of a hadron calorimeter and hadron shower initiated by a particle of energy E emitted at an angle θ or equivalently pseudorapidity $\eta = -\ln \tan(\theta/2)$. The hadron shower produces signals s_1, s_2 and s_3. Some of the shower's energy may penetrate through the calorimeter undetected.

The number of shower particles grows roughly with the energy of the incident particle, and so by measuring the signals s_1, s_2 and s_3 one can estimate the incident energy. As the amount of energy leakage depends on the angle of the particle, the variable η also contains some information that helps to relate the measured signals to the incident energy. Thus for the present example the target value is the energy ($y = E$) and the feature vector contains the measured values $\mathbf{x} = (\eta, s_1, s_2, s_3)$. The goal is to find a function $f(\mathbf{x})$ that can be used to estimate a particle's energy.

A naive guess for the energy of the incident particle is simply the sum of the signal values $s_1 + s_2 + s_3$. Figure 3 shows scatter plots of this estimate versus the true energy for (a) all events, (b) events with $|\eta| < 0.5$ corresponding to almost normal incidence to the surface of the calorimeter, (c) $0.5 < |\eta| < 1.0$ and (d) $1.0 < |\eta| < 1.5$, corresponding to a more oblique angle of incidence. For normal incidence (smaller $|\eta|$), more of the energy leaks through the calorimeter and thus the sum of signal values $\sum_i s_i$ tends to underestimate the true energy.

The goal of multiple regression here is to construct an estimator for the energy that is much better than simply the sum $\sum_i s_i$. A program that carries out linear regression for this problem is given in Appendix A. Here the most important parts of the code are discussed.

First the data are read in from a file called `trainingData.txt`. This is a text file with five columns of values: E (in GeV), η, s_1, s_2 and s_3 (the units of the energy and measured signals are GeV). The values are read in and assigned to numpy arrays with

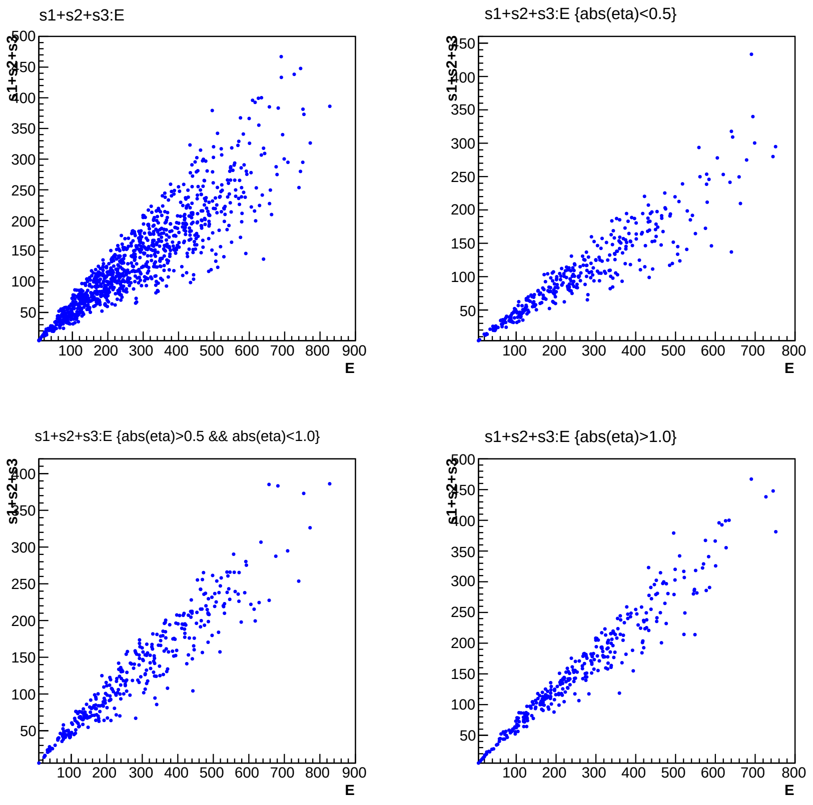

Figure 3: Scatter plots of the estimated versus true energy for different angular regions (see text).

```
events = np.loadtxt('trainingData.txt')
nEvt = events.shape[0]
X = events[:,1:]            # columns 1,2,3,4 are eta, s1, s2, s3
y = events[:,0]             # target value E is column 0
```

Then just as in classification problems, the data is split into two samples, one used to train the fit function, that is, to minimize the loss function $L(\mathbf{w})$ and thus determine the weights $\mathbf{w}$, and another part used to evaluate the performance of the resulting $f(\mathbf{x};\mathbf{w})$.

Finding the weights for regression in `scikit-learn` is very similar to the corresponding step with a classifier. First one initializes the regression function (or "regressor"), e.g., by creating an object of type `LinearRegressor` here called `regr`, and then minimizing the loss function by calling the function `fit` using the feature vectors and true target values from the training portion of the data:

```
regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
```

This finds the weights $\mathbf{w}$ and thus determines the regression function $f(\mathbf{x};\mathbf{w})$.

The function can then be used on the test data and a measure of how well the predicted values agree with the true targets computed. There are many possible ways of comparing the two. Here this is done with the so-called R^2 value (also called the coefficient of determination, see, e.g., Ref. [2]), defined such that higher R^2 corresponds to better predictive power with a maximum value of 1. This is found using the regressor's `score` function with

```
y_pred = regr.predict(X_test)
R2 = regr.score(X_test, y_test)
```

4. Project exercises

Exercise 1: Run the program simpleRegressor.py and describe the output. Make a scatter-plot of the *relative resolution*

$$r = \frac{\hat{E} - E}{E} \tag{5}$$

versus the true energy E, where $\hat{E} = f(\mathbf{x}; \mathbf{w})$ is the estimated ("reconstructed") energy. Find the mean and standard deviation of r. Instead of the R^2 coefficient defined earlier, we can use as the measure of quality

$$\langle r^2 \rangle = \sigma_r^2 + \langle r \rangle^2 = \frac{1}{N} \sum_{i=1}^{N} r_i^2 \;, \tag{6}$$

where σ_r is the standard deviation and $\langle r \rangle$ is the mean of r.

Exercise 2: Modify the program to include an MLP regressor. Adjust the number of hidden layers and nodes to minimize the MSE.

Exercise 3: Investigate preprocessing of the inputs with `scikit-learn`'s `preprocessing` package. Try, for example, transforming the input values so that they all have a mean of zero and standard deviation of one.

Exercise 4: Investigate using cross-validation to determine the optimal architecture.

Exercise 5: Try implementing any of the other regression algorithms available in `scikit-learn`.

Exercise 6: Try using different loss functions available in `scikit-learn`.

References

[1] Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani, *An Introduction to Statistical Learning with Applications in R*, Springer, 2013; `http://www-bcf.usc.edu/~gareth/ISL/`

[2] Wikipedia entry for *Coefficient of determination*, `en.wikipedia.org/wiki/Coefficient_of_determination`.

A. Python code

Program `simpleRegressor.py` for multiple regression.

```python
#  simpleRegressor.py
#  G. Cowan / RHUL Physics / November 2021
#  Simple program to illustrate regression with scikit-learn

import scipy as sp
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

from sklearn import linear_model
from sklearn.model_selection import train_test_split
from sklearn import metrics

#  read the data in from file
events = np.loadtxt('trainingData.txt')
nEvt = events.shape[0]
X = events[:,1:]           # columns 1,2,3,4 are eta, s1, s2, s3
y = events[:,0]            # target value E is column 0
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)

# create regressor object, train and test
regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
R2 = regr.score(X_test, y_test)
print(f"Test R2 score: {R2:.3f}")

# make a plot
matplotlib.rcParams.update({'font.size':12})     # set all font sizes
fig, ax = plt.subplots(1,1)
plt.gcf().subplots_adjust(bottom=0.15)
plt.gcf().subplots_adjust(left=0.15)
ax.set_xlim((0.,1000.))
ax.set_ylim((0.,1000.))
x0,x1 = ax.get_xlim()
y0,y1 = ax.get_ylim()
ax.set_aspect(abs(x1-x0)/abs(y1-y0))    # make square plot
xtick_spacing = 200
ytick_spacing = 200
ax.yaxis.set_major_locator(ticker.MultipleLocator(xtick_spacing))
ax.yaxis.set_major_locator(ticker.MultipleLocator(ytick_spacing))
plt.xlabel('true energy   (GeV)', labelpad=3)
plt.ylabel('reconstructed energy   (GeV)', labelpad=3)
plt.scatter(y_test, y_pred, s=3, color='dodgerblue', marker='o')
plt.figtext(0.3, 0.81, f'Linear Regressor')
plt.figtext(0.3, 0.73, f'$R^2$ = {R2:.3f}')
plt.show()
plt.savefig("LinearRegressor.pdf", format='pdf')
```

G. Cowan
RHUL Physics
Version 1.1, December 2021