Logistic Regression and Gradient Methods

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, USA

December 12, 2022

Logistic Regression

In linear regression our main interest was centered on learning the coefficients of a functional fit (say a polynomial) in order to be able to predict the response of a continuous variable on some unseen data. The fit to the continuous variable y_i is based on some independent variables $\boldsymbol{x}_i$. Linear regression resulted in analytical expressions for standard ordinary Least Squares or Ridge regression (in terms of matrices to invert) for several quantities, ranging from the variance and thereby the confidence intervals of the parameters $\boldsymbol{\beta}$ to the mean squared error. If we can invert the product of the design matrices, linear regression gives then a simple recipe for fitting our data.

Classification problems

Classification problems, however, are concerned with outcomes taking the form of discrete variables (i.e. categories). We may for example, on the basis of DNA sequencing for a number of patients, like to find out which mutations are important for a certain disease; or based on scans of various patients' brains, figure out if there is a tumor or not; or given a specific physical system, we'd like to identify its state, say whether it is an ordered or disordered system (typical situation in solid state physics); or classify the status of a patient, whether she/he has a stroke or not and many other similar situations.

The most common situation we encounter when we apply logistic regression is that of two possible outcomes, normally denoted as a binary outcome, true or false, positive or negative, success or failure etc.

Optimization and Deep learning

Logistic regression will also serve as our stepping stone towards neural network algorithms and supervised deep learning. For logistic learning, the minimization

of the cost function leads to a non-linear equation in the parameters $\boldsymbol{\beta}$. The optimization of the problem calls therefore for minimization algorithms. This forms the bottle neck of all machine learning algorithms, namely how to find reliable minima of a multi-variable function. This leads us to the family of gradient descent methods. The latter are the working horses of basically all modern machine learning algorithms.

We note also that many of the topics discussed here on logistic regression are also commonly used in modern supervised Deep Learning models, as we will see later.

Basics

We consider the case where the dependent variables, also called the responses or the outcomes, y_i are discrete and only take values from $k = 0, \ldots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and $y_i = 1$. Our outcomes could represent the status of a credit card user that could default or not on her/his credit card debt. That is

$$y_i = \begin{bmatrix} 0 & \text{no} \\ 1 & \text{yes} \end{bmatrix}.$$

Linear classifier

Before moving to the logistic model, let us try to use our linear regression model to classify these two outcomes. We could for example fit a linear model to the default case if $y_i > 0.5$ and the no default case $y_i \leq 0.5$.

We would then have our weighted linear combination, namely

$$\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{1}$$

where $\boldsymbol{y}$ is a vector representing the possible outcomes, $\boldsymbol{X}$ is our $n \times p$ design matrix and $\boldsymbol{\beta}$ represents our estimators/predictors.

Some selected properties

The main problem with our function is that it takes values on the entire real axis. In the case of logistic regression, however, the labels y_i are discrete variables. A typical example is the credit card data discussed below here, where we can set the state of defaulting the debt to $y_i = 1$ and not to $y_i = 0$ for one the persons in the data set (see the full example below).

One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to values $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 if otherwise. We will encounter this model in our first demonstration of

neural networks. Historically it is called the “perceptron" model in the machine learning literature. This model is extremely simple. However, in many cases it is more favorable to use a “soft" classifier that outputs the probability of a given category. This leads us to the logistic function.

Simple example

The following example on data for coronary heart disease (CHD) as function of age may serve as an illustration. In the code here we read and plot whether a person has had CHD (output = 1) or not (output = 0). This ouput is plotted the person's against age. Clearly, the figure shows that attempting to make a standard linear regression fit may not be very meaningful.

```
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.model_selection import train_test_split
from sklearn.utils import resample
from sklearn.metrics import mean_squared_error
from IPython.display import display
from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("chddata.csv"),'r')

# Read the chd data as  csv file and organize the data into arrays with age group, age, and chd
chd = pd.read_csv(infile, names=('ID', 'Age', 'Agegroup', 'CHD'))
chd.columns = ['ID', 'Age', 'Agegroup', 'CHD']
output = chd['CHD']
age = chd['Age']
```

```
agegroup = chd['Agegroup']
numberID  = chd['ID']
display(chd)

plt.scatter(age, output, marker='o')
plt.axis([18,70.0,-0.1, 1.2])
plt.xlabel(r'Age')
plt.ylabel(r'CHD')
plt.title(r'Age distribution and Coronary heart disease')
plt.show()
```

Plotting the mean value for each group

What we could attempt however is to plot the mean value for each group.

```
agegroupmean = np.array([0.1, 0.133, 0.250, 0.333, 0.462, 0.625, 0.765, 0.800])
group = np.array([1, 2, 3, 4, 5, 6, 7, 8])
plt.plot(group, agegroupmean, "r-")
plt.axis([0,9,0, 1.0])
plt.xlabel(r'Age group')
plt.ylabel(r'CHD mean values')
plt.title(r'Mean values for each age group')
plt.show()
```

We are now trying to find a function $f(y|x)$, that is a function which gives us an expected value for the output y with a given input x. In standard linear regression with a linear dependence on x, we would write this in terms of our model

$$f(y_i|x_i) = \beta_0 + \beta_1 x_i.$$

This expression implies however that $f(y_i|x_i)$ could take any value from minus infinity to plus infinity. If we however let $f(y|y)$ be represented by the mean value, the above example shows us that we can constrain the function to take values between zero and one, that is we have $0 \leq f(y_i|x_i) \leq 1$. Looking at our last curve we see also that it has an S-shaped form. This leads us to a very popular model for the function f, namely the so-called Sigmoid function or logistic model. We will consider this function as representing the probability for finding a value of y_i with a given x_i.

The logistic function

Another widely studied model, is the so-called perceptron model, which is an example of a "hard classification" model. We will encounter this model when we discuss neural networks as well. Each datapoint is deterministically assigned to a category (i.e $y_i = 0$ or $y_i = 1$). In many cases, and the coronary heart disease data forms one of many such examples, it is favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category k. Logistic regression is the most common example of a so-called soft classifier. In logistic regression, the probability that a data point x_i belongs to a category

$y_i = \{0, 1\}$ is given by the so-called logit function (or Sigmoid) which is meant to represent the likelihood for a given event,

$$p(t) = \frac{1}{1 + \exp -t} = \frac{\exp t}{1 + \exp t}.$$

Note that $1 - p(t) = p(-t)$.

Examples of likelihood functions used in logistic regression and nueral networks

The following code plots the logistic function, the step function and other functions we will encounter from here and on.

```
"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative scale.
The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')

plt.show()

"""Step Function"""
z = numpy.arange(-5, 5, .02)
step_fn = numpy.vectorize(lambda z: 1.0 if z >= 0.0 else 0.0)
step = step_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, step)
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('step function')

plt.show()

"""tanh Function"""
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.tanh(z)
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, t)
ax.set_ylim([-1.0, 1.0])
ax.set_xlim([-2*mt.pi,2*mt.pi])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('tanh function')

plt.show()
```

Two parameters

We assume now that we have two classes with y_i either 0 or 1. Furthermore we assume also that we have only two parameters β in our fitting of the Sigmoid function, that is we define probabilities

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}),$$

where $\boldsymbol{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

Note that we used

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}).$$

Maximum likelihood

In order to define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0, 1\}$ and where the data points are drawn independently, we use the so-called Maximum Likelihood Estimation (MLE) principle. We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \boldsymbol{\beta})\right]^{y_i} \left[1 - p(y_i = 1|x_i, \boldsymbol{\beta}))\right]^{1-y_i}$$

from which we obtain the log-likelihood and our **cost/loss** function

$$\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \log p(y_i = 1|x_i, \boldsymbol{\beta}) + (1 - y_i) \log \left[1 - p(y_i = 1|x_i, \boldsymbol{\beta}))\right]\right).$$

The cost function rewritten

Reordering the logarithms, we can rewrite the **cost/loss** function as

$$\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. Since the cost (error) function is just the negative log-likelihood, for logistic regression we have that

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

This equation is known in statistics as the **cross entropy**. Finally, we note that just as in linear regression, in practice we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization as we did for Ridge and Lasso regression.

Minimizing the cross entropy

The cross entropy is a convex function of the weights $\boldsymbol{\beta}$ and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to the two parameters β_0 and β_1 we obtain

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right),$$

and

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i\frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right).$$

A more compact expression

Let us now define a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\boldsymbol{X}$ which contains the x_i values and a vector $\boldsymbol{p}$ of fitted probabilities $p(y_i|x_i, \boldsymbol{\beta})$. We can rewrite in a more compact form the first derivative of cost function as

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T\left(\boldsymbol{y} - \boldsymbol{p}\right).$$

If we in addition define a diagonal matrix $\boldsymbol{W}$ with elements $p(y_i|x_i, \boldsymbol{\beta})(1 - p(y_i|x_i, \boldsymbol{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}.$$

Extending to more predictors

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log\frac{p(\boldsymbol{\beta x})}{1 - p(\boldsymbol{\beta x})} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Here we defined $\boldsymbol{x} = [1, x_1, x_2, \dots, x_p]$ and $\boldsymbol{\beta} = [\beta_0, \beta_1, \dots, \beta_p]$ leading to

$$p(\boldsymbol{\beta x}) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}{1 + \exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}.$$

Including more classes

Till now we have mainly focused on two classes, the so-called binary system. Suppose we wish to extend to K classes. Let us for the sake of simplicity assume we have only two predictors. We have then following model

$$\log \frac{p(C = 1|x)}{p(K|x)} = \beta_{10} + \beta_{11} x_1,$$

and

$$\log \frac{p(C = 2|x)}{p(K|x)} = \beta_{20} + \beta_{21} x_1,$$

and so on till the class $C = K - 1$ class

$$\log \frac{p(C = K - 1|x)}{p(K|x)} = \beta_{(K-1)0} + \beta_{(K-1)1} x_1,$$

and the model is specified in term of $K - 1$ so-called log-odds or **logit** transformations.

More classes

In our discussion of neural networks we will encounter the above again in terms of a slightly modified function, the so-called **Softmax** function.

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression (also known as softmax regression), multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks. Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the predicted probability for the k-th class given a sample vector $\boldsymbol{x}$ and a weighting vector $\boldsymbol{\beta}$ is (with two predictors):

$$p(C = k|\mathbf{x}) = \frac{\exp\left(\beta_{k0} + \beta_{k1} x_1\right)}{1 + \sum_{l=1}^{K-1} \exp\left(\beta_{l0} + \beta_{l1} x_1\right)}.$$

It is easy to extend to more predictors. The final class is

$$p(C = K|\mathbf{x}) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp\left(\beta_{l0} + \beta_{l1} x_1\right)},$$

and they sum to one. Our earlier discussions were all specialized to the case with two classes only. It is easy to see from the above that what we derived earlier is compatible with these equations.

To find the optimal parameters we would typically use a gradient descent method. Newton's method and gradient descent methods are discussed in the material on optimization methods.

Wisconsin Cancer Data

We show here how we can use a simple regression case on the breast cancer data using Logistic regression as our algorithm for classification.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
```

Question: How would you scale these data?

Using the correlation matrix

In addition to the above scores, we could also study the covariance (and the correlation matrix). We use **Pandas** to compute the correlation matrix.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()
import pandas as pd
# Making a data frame
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)

fig, axes = plt.subplots(15,2,figsize=(10,20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]
ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:,i], bins =50)
    ax[i].hist(malignant[:,i], bins = bins, alpha = 0.5)
    ax[i].hist(benign[:,i], bins = bins, alpha = 0.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["Malignant", "Benign"], loc ="best")
fig.tight_layout()
plt.show()

import seaborn as sns
```

```
correlation_matrix = cancerpd.corr().round(1)
# use the heatmap function from seaborn to plot the correlation matrix
# annot = True to print the values inside the square
plt.figure(figsize=(15,8))
sns.heatmap(data=correlation_matrix, annot=True)
plt.show()
```

Discussing the correlation data

In the above example we note two things. In the first plot we display the overlap of benign and malignant tumors as functions of the various features in the Wisconsing breast cancer data set. We see that for some of the features we can distinguish clearly the benign and malignant cases while for other features we cannot. This can point to us which features may be of greater interest when we wish to classify a benign or not benign tumour.

In the second figure we have computed the so-called correlation matrix, which in our case with thirty features becomes a 30×30 matrix.

We constructed this matrix using **pandas** via the statements

```
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)
```

and then

```
correlation_matrix = cancerpd.corr().round(1)
```

Diagonalizing this matrix we can in turn say something about which features are of relevance and which are not. This leads us to the classical Principal Component Analysis (PCA) theorem with applications. This will be discussed later this semester (week 43).

Other measures in classification studies: Cancer Data again

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
#Cross validation
accuracy = cross_validate(logreg,X_test,y_test,cv=10)['test_score']
print(accuracy)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test_scaled,y_tes


import scikitplot as skplt
y_pred = logreg.predict(X_test)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
plt.show()
y_probas = logreg.predict_proba(X_test)
skplt.metrics.plot_roc(y_test, y_probas)
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
plt.show()
```

Optimization, the central part of any Machine Learning algortithm

Almost every problem in machine learning and data science starts with a dataset X, a model $g(\beta)$, which is a function of the parameters β and a cost function $C(X, g(\beta))$ that allows us to judge how well the model $g(\beta)$ explains the observations X. The model is fit by finding the values of β that minimize the cost function. Ideally we would be able to solve for β analytically, however this is not possible in general and we must use some approximative/numerical method to compute the minimum.

Revisiting our Logistic Regression case

In our discussion on Logistic Regression we studied the case of two classes, with y_i either 0 or 1. Furthermore we assumed also that we have only two parameters β in our fitting, that is we defined probabilities

$$
\begin{aligned}
p(y_i = 1|x_i, \boldsymbol{\beta}) &= \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}, \\
p(y_i = 0|x_i, \boldsymbol{\beta}) &= 1 - p(y_i = 1|x_i, \boldsymbol{\beta}),
\end{aligned}
$$

where $\boldsymbol{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

The equations to solve

Our compact equations used a definition of a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\boldsymbol{X}$ which contains the x_i values and a vector $\boldsymbol{p}$ of fitted probabilities $p(y_i|x_i, \boldsymbol{\beta})$. We rewrote in a more compact form the first derivative of the cost function as

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{p}\right).$$

If we in addition define a diagonal matrix $\boldsymbol{W}$ with elements $p(y_i|x_i,\boldsymbol{\beta})(1-p(y_i|x_i,\boldsymbol{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}.$$

This defines what is called the Hessian matrix.

Solving using Newton-Raphson's method

If we can set up these equations, Newton-Raphson's iterative method is normally the method of choice. It requires however that we can compute in an efficient way the matrices that define the first and second derivatives.

Our iterative scheme is then given by

$$\boldsymbol{\beta}^{\mathrm{new}} = \boldsymbol{\beta}^{\mathrm{old}} - \left(\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta}^T}\right)^{-1}_{\boldsymbol{\beta}^{\mathrm{old}}} \times \left(\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\right)_{\boldsymbol{\beta}^{\mathrm{old}}},$$

or in matrix form as

$$\boldsymbol{\beta}^{\mathrm{new}} = \boldsymbol{\beta}^{\mathrm{old}} - \left(\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}\right)^{-1} \times \left(-\boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{p})\right)_{\boldsymbol{\beta}^{\mathrm{old}}}.$$

The right-hand side is computed with the old values of β.

If we can compute these matrices, in particular the Hessian, the above is often the easiest method to implement.

Brief reminder on Newton-Raphson's method

Let us quickly remind ourselves how we derive the above method.

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method requires the evaluation of both the function f and its derivative f' at arbitrary points. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we normally discourage the use of this method.

The equations

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots.$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s - x)f'(x) \approx 0,$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}.$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Simple geometric interpretation

The above is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the x-axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally

Extending to more than one variable

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0, \end{aligned}$$

which we Taylor expand to obtain

$$\begin{aligned} 0 = f_1(x_1 + h_1, x_2 + h_2) = &\quad f_1(x_1, x_2) + h_1 \partial f_1/\partial x_1 + h_2 \partial f_1/\partial x_2 + \dots \\ 0 = f_2(x_1 + h_1, x_2 + h_2) = &\quad f_2(x_1, x_2) + h_1 \partial f_2/\partial x_1 + h_2 \partial f_2/\partial x_2 + \dots \end{aligned}.$$

Defining the Jacobian matrix $\mathbf{J}$ we have

$$\mathbf{J} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix},$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^n \\ x_2^n \end{pmatrix} + \begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix},$$

where we have defined

$$\begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix} = -\mathbf{J}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}.$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case **J** is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations. In our case, the Jacobian matrix is given by the Hessian that represents the second derivative of cost function.