Data Analysis and Machine Learning: Logistic Regression

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Sep 26, 2018

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 $\hat{x}_i$. Linear regression resulted in analytical expressions (in terms of matrices to invert) for several quantities, ranging from the variance and thereby the confidence intervals of the parameters $\hat{\beta}$ 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, 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

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 $\hat{\beta}$. The optmization 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 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 $k = 0, \dots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $\hat{X} \in \mathbb{R}^{n\times p}$ 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 who could default or not on her/his credit card debt. That is

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

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

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

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

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.

The code for plotting the perceptron can be seen here. This si nothing but the standard Heaviside step function.

The logistic function

The perceptron 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, 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 $1 - p(t) = p(-t)$. The following code plots the logistic function.

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

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

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

Note that we used

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

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

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

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

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

The cost function rewritten

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

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

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

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

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 $\hat{\beta}$ 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

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

and

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

A more compact expression

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

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

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

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

Optimizing the cost function

Newton's method and gradient descent methods

A **scikit-learn** example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else

from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)

X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
plt.show()
```

A simple classification problem

```
import numpy as np
from sklearn import datasets, linear_model
import matplotlib.pyplot as plt


def generate_data():
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    return X, y


def visualize(X, y, clf):
    # plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
    # plt.show()
    plot_decision_boundary(lambda x: clf.predict(x), X, y)
    plt.title("Logistic Regression")


def plot_decision_boundary(pred_func, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
```