Lecture Notes on Regression

Rein van den Boomgaard

1 Statistical View on Regression

We assume there is a functional relation between the independent variable x (the input) and dependent variable y (the output). We also assume that this functional relation (map from x to y) can be written as:

$$y = m(x).$$

The exact analytical form of the function m is most often not known. In fact all we have are examples of pairs (x_i, y_i) and the observations of y for any given x are not equal to $m(x) + R$ where R is a random variable. So we have:

$$Y = m(x) + R$$

.

The goal of regression is to find the map m from the set of examples. To make regression mathematically tractable we assume that m is from a known parameterized family of functions. In general there may be more then one parameter defining m. In machine learning the parameters of the map m are denoted as the vector $\boldsymbol{\theta}$ and we write

$$Y = m(x; \boldsymbol{\theta}) + R.$$

The semicolon is used to distinguish between 'proper' arguments and parameters. In case we have more then one independent variable we collect them in vector $\boldsymbol{x}$ and write:

$$Y = m(\boldsymbol{x}; \boldsymbol{\theta}) + R.$$

To simplify things we often assume that the random variable R is independent on the variable $\boldsymbol{x}$ and normally distributed with zero mean and variance σ^2. Then we have the $Y|\boldsymbol{x}$ is also normally distributed with mean $m(\boldsymbol{x}; \boldsymbol{\theta})$:

$$Y|\boldsymbol{x} \approx \mathrm{N}(m(\boldsymbol{x}; \boldsymbol{\theta}), \sigma^2).$$

2 Maximum Likelihood Estimator

The goal of regression can be formulated as: given the data $(\boldsymbol{x}_i, y_i)$ for $i = 1, \ldots, N$, what is the parameter vector $\boldsymbol{\theta}$ such that the model $m(\cdot; \boldsymbol{\theta})$ fits the data best.

In other words, what parameter vector $\boldsymbol{\theta}$ makes the observed data most plausible? A way to calculate such an optimal parameter vector is the **maximum likelihood estimator**:

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} f(y_1, \ldots, y_N | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N; \boldsymbol{\theta}).$$

Here f is the joint conditional probability density function that gives the probability of making measurements $y_1, \ldots, y_N$ given the independent variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ assuming a model with parameter vector $\boldsymbol{\theta}$.

Assuming that the residuals for each observation are independent and identically distributed we have:

$$f(y_1, \ldots, y_N | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N; \boldsymbol{\theta}) = \prod_{i=1}^{N} f(y_i | \boldsymbol{x}_i; \boldsymbol{\theta}).$$

For normally distributed residuals we get:

$$f(y_1, \ldots, y_N | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N; \boldsymbol{\theta}) = \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2}\right).$$

Using the **log likelihood**, defined as:

$$\ell(\boldsymbol{\theta}) = \log \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2}\right)$$

we can write:

$$\begin{aligned}
\ell(\boldsymbol{\theta}) &= \sum_{i=1}^{N} \log\left[\frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2}\right)\right] \\
&= \sum_{i=1}^{N} \left[-\log(\sigma\sqrt{2\pi}) - \frac{1}{2\sigma^2}\left(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta})\right)^2\right]
\end{aligned}$$

Note that maximizing the log likelihood leads to the same estimate as *minimizing* the sum of squared errors

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta})\right)^2$$

3 Linear Regression

When we assume a linear model (i.e. linear in its parameters) the maximum likelihood estimator for normally distributed residuals boils down the *ordinary least squares estimation*. The solution is not hard to find. The linear model is:

$$m(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}^\intercal \boldsymbol{x}$$

Collecting all vectors $\boldsymbol{x}_i$ in a matrix X:

$$X = (\boldsymbol{x}_1 \cdots \boldsymbol{x}_N)$$

and all values y_i in a vector $\boldsymbol{y}$ the sum of square errors can be written as:

$$\sum_{i=1}^{N} \left(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta})\right)^2 = \|\boldsymbol{y} - X\boldsymbol{\theta}\|^2$$

and thus:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \|\boldsymbol{y} - X\boldsymbol{\theta}\|^2$$

Remember from linear algebra class that the solution is:

$$\hat{\boldsymbol{\theta}} = (X^\intercal X)^{-1} X^\intercal \boldsymbol{y}$$

4 Beyond Linear Regression

You may wonder why we need more then such simple expression for regression? There are two main reasons why linear regression is not always useful:

- Linear Regression assumes as outlined above assumes a linear model whereas in reality not every model is linear in its parameters (we will see an example shortly).

- Regression as introduced here assumes normally distributed residuals. In some situations this is a wrong assumption, then the noise shows *heavier tails* meaning that residuals with large (absolute) values are more common then implied by the normal distribution.

  One such distribution is the Laplace distribution where the squared error $(y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$ is replaced with the absolute error $|y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta})|$. When you note that the pdf of the Laplace distribution is not differentiable at the origin it should become obvious that finding the optimal $\boldsymbol{\theta}$ is more difficult.

For a generic non linear regression model, assuming iid residuals, the MLE is:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \log f(y_i | \boldsymbol{x}_i; \boldsymbol{\theta})$$

Assuming normally distributed residuals as well, the MLE reduces to:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$$

Unfortunately in both these cases closed form solutions (solutions that can be written as formulas using the basic math functions) are often not known (or even known not to exist). We are then left with the problem of numerical optimization.

5 Numerical Optimization: Gradient Descent

Consider the function F in one variable x of which we seek the point x^* such that F has a local minimum in $x^\star$. Assuming that F is nicely differentiable we must have that $F'(x^\star) = 0$ and $F''(x^\star) > 0$ for $x^\star$ to be a local minimum.

Let $x^{(1)}$ be a first guess of where a local minimum is. Assume that we can calculate the derivative $F'(x^{(1)})$ in case this derivative value is negative we know that the local minimum is to the right of $x^{(1)}$. If the derivative is positive the local minimum must be on the left.

Consider the following incremental update rule:

$$x^{(k+1)} = x^{(k)} - \eta F'(x^{(k)}$$

the factor η is called the *stepsize* (in principle the stepsize can be a function of x and the iteration number k, here we consider only constant η).

Selecting a step size is not trivial. Too large a step size and the true local minimum will be stepped over, there the sign of the derivative changes and the next step can again jump over the local minimum leading to oscillatory behaviour. Too small a step size on the other hand can lead to very very slow convergence.

There are many clever algorithms that change the step size (also called the learning rate) as the search progresses. We won't deal with these mathematical subtilities.

Deciding at what k to stop the iterations is non trivial too. You could look for the x value not changing (much) anymore, or you could look at the value F itself.

Note that finding a local maximum can be found with almost the same iteration rule, but the $-$ sign is replaced with a $+$ sign.

Now consider the situation where F is a function from $\mathbb{R}^n$ to $\mathbb{R}$: $\boldsymbol{x} \in \mathbb{R}^n \mapsto F(\boldsymbol{x}) \in \mathbb{R}$. The derivative is then replaced with the gradient vector ∇F and we obtain the following rule:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \eta \nabla F(\boldsymbol{x}^{(k)}$$

Remember that the elements of the gradient vector are the partial derivatives of F with respect to the elements of $\boldsymbol{x}$ and that it points into the direction of largest increase in value. So in case we are looking for a local minimum we need to make a step in the *negative* gradient direction.

A word of advice to computer scientists: never write (mathematical) software if someone else has done it already. Unless of course you need someting new, or run an algorithm on architectures for which standard mathematical software is not available. But believe me this doesn't mean that you can lean back and avoid math. You have to read the documentation and decide what algorithm will work for you (e.g. read the documentation of the scipy.optimize.minimize, it doesn't even contain the simple gradient descent method...).

6 Example (Linear Regression)

As a simple example consider linear regression. Although it can be solved using linear algebra techniques we can solve it with numerical methods too (just for fun of course).

We use a simple linear model: $y_i = \theta_1 + \theta_2 x_i + r_i$ (here r_i is the residual). Assuming normally distributed residuals the MLE reduces to:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i - m(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$$

as we have shown in a previous section. Collecting all our data in arrays (matrix and vector) we define

$$X = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \end{pmatrix}^\intercal$$

and

$$\boldsymbol{y} = \begin{pmatrix} y_1 & y_2 & \cdots & y_N \end{pmatrix}^\intercal$$

With this notation the MLE can be rewritten as:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \|\boldsymbol{y} - X\boldsymbol{\theta}\|^2$$

The gradient vector of the objective function $\|\boldsymbol{y} - X\boldsymbol{\theta}\|^2$ to be minimized equals $2(X^\intercal X\boldsymbol{\theta} - X^\intercal \boldsymbol{y})$ (remember the vector derivatives from the computer vision course).

The gradient descent rule then becomes:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - 2\eta(X^\intercal X\boldsymbol{\theta}^{(k)} - X^\intercal \boldsymbol{y})$$

The code below first generates a noisy version of the line $y = 0.5x + 2$ and then estimates the parameters using the scipy.optimization.minimize function. This is numerical minimization method. It needs the function to be minimized (we call it J), a start value for the parameter vector (we use (0,0)), the gradient function (which is called jac, for Jacobian, and so we named it Jac) and a string that defines the method to be used.

Results of the minimization are returned in a dictionary that is printed at the end. Also the 'true' function is shown (in green) and the estimated function (in blue).

```
from pylab import linspace, randn, ones, vstack, plot, savefig, dot, norm
from scipy.optimize import minimize

th1 = 2
th2 = .5

x = linspace(0,10,100)
y = th1 + th2*x + 0.3*randn(*x.shape)
X = vstack((ones(x.shape),x)).T

def J(th):
    return norm(y - dot(X,th))**2
```

```
def Jac(th):
    return 2*(dot(dot(X.T, X),th) - dot(X.T,y))

res = minimize(J, (0,0), jac=Jac, method='CG')
print res

eth1, eth2 = res.x
plot(x, y, 'xb')
plot(x, eth1+eth2*x, 'b')
plot(x, th1+th2*x, 'g')
savefig('figures/linregression.png')

 None
```

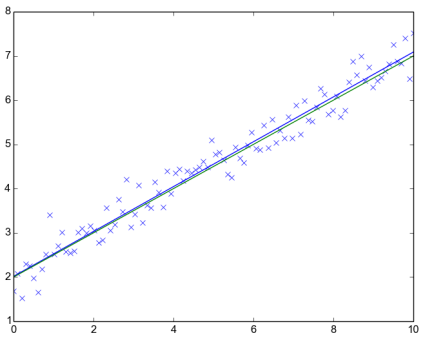

Figure 1: Numerical Linear Regression

7 Exercise (Non-linear Regression)

Consider the model $Y = m(x;\boldsymbol{\theta}) + R$ where we assume the residual R is normally distributed (and iid) with:

$$m(x;\boldsymbol{\theta}) = \theta_1 \sin(\theta_2 x)$$

This a non-linear model (because of the θ_2 *in* the sin function) and thus the MLE is:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

where

$$\begin{aligned} J(\boldsymbol{\theta}) = J(\theta_1,\theta_2) &= \sum_{i=1}^{N} (y_i - m(\boldsymbol{x}_i;\boldsymbol{\theta}))^2 \\ &= \sum_{i=1}^{N} (y_i - \theta_1 \sin(\theta_2 x))^2 \end{aligned}$$

Use the previous section as a template to make a program to:

1. generate 'noisy' data for this model
2. define the function J that returns $J(\boldsymbol{\theta})$
3. calculate the gradient of J (and define it in a function Jac)

4. use the conjugate gradient method to calculate the MLE.

For this non-linear model i would not try to write it down in linear algebra terms (it is bound to fail). Collect the x_i's and y_i's in two vectors and make a straightforward calculation of $J(\boldsymbol{\theta})$ (the summation loop can be avoided in numpy of course).