Optimization of mathematical functions

Dario Beraldi

January 2, 2016

Contents

Abstract

Optimizing a mathematical function means to find the set of parameters that minimize or maximize the given function. These are examples of what this procedure means. We don't code the actual solvers, rather we show how to use the R function `optim`. Source code and README for this document is on https://github.com/dariober/sympy-books. A useful link is also http://www.magesblog.com/2013/03/how-to-use-optim-in-r.html

Optimize residual sum of squares function

Say you have a vector of responses (response variable, y) and a vector of predictors (x), what is the best line fit? We can assess the quality of the fit using the sum if squares

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

where $f(x)$ is a function applied to the predictor to approach y. Since the lowest the RSS the best is the fit, we need to optimize the RSS function to minimize it. If we have only one predictor of real numbers and we want a linear fit, $f(x)$ becomes the equation of a line $y = \alpha + \beta x$, so we have:

$$RSS = \sum_{i=1}^{n} (y_i - (\alpha + \beta x_i))^2$$

So the task is: Find the values of the parameters α and β that make RSS as small as possible.

First define in `R` the objective function RSS and prepare some toy data. The `optim` function is quite prescriptive about how the objective function should be coded:

> **fn**: A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.

So the first arg should be a vector of parameters to optimize, α and β in this case, then the actual x and y values

```
rssFn<- function(par= c(a, b), x, y){
    rss<- sum((y - (par[1] + par[2] * x))^2)
    return(rss)
}

# xy data
set.seed(1234)
x<- 1:10
y<- 3 * x + rnorm(n= length(x))
```

Now optimize

```
optim(par= c(0, 1), rssFn, x= x, y= y)

$par
[1] -0.191130  2.965032

$value
[1] 8.822208

$counts
function gradient
      63       NA

$convergence
[1] 0

$message
NULL
```

The parameters are $\alpha \sim 0$ and $\beta \sim 3$ as expected from the toy dataset. Compare to R linear model function which implemenents ordinary least square

```
lm(y ~ x)
...
Coefficients:
(Intercept)            x
    -0.1896       2.9648
```

There is a slight discrepancy between the two methods, probably due to the choice optimization method and starting values. Note also that with only two parameters to optimize a grid search of the parameter space might be feasible solution.

What if instead of linear fit we want a polynomial of degree 2? In this case the objective function has three parameters to opimize

$$RSS = \sum_{i=1}^{n}(y_i - (\alpha + \beta_1 x_i + \beta_2 x_i^2))^2$$

Let's recode the previous R function, optimize and check against ordinary least square:

```
rssFn<- function(par= c(a, b1, b2), x, y){
    rss<- sum((y - (par[1] + par[2] * x + par[3] * x^3))^2)
    return(rss)
}

smry<- summary(lm(y ~ x + I(x^2)))
...
Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.91247    1.27703  -0.715  0.49803
x            3.32623    0.53334   6.237  0.00043 ***
I(x^2)      -0.03286    0.04725  -0.695  0.50927
...
sum(smry$residuals^2) # <--- RSS
[1] 8.252231
```

It appears that the default optimization method doesn't, *Nelder-Mead*, perform very well:

```
optim(par= c(0, 1, 2), rssFn, x= x, y= y, method= "Nelder-Mead")
$par
[1] 0.58416441 1.38456141 0.01874051
$value
[1] 133.961 # <--- RSS

## This looks better:
optim(par= c(0, 1, 2), rssFn, x= x, y= y, method= "BFGS")
$par
[1] -0.700144497  3.158923891 -0.001841632
$value
[1] 8.324194 # <-- RSS
```

See Wikipedia for residual sum of squares

Optimize normal distribution to fit data

We want to fit normal distibution function to a given vector of data points. Which parameters of the function best approximate the data? This a *maximum likelihood* problem: What parameters value maximize the likelihood o the data?

The normal distribution has two parameters, mean μ and standard deviation σ. The probability density function is

$$pdf(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The likelihood of the data is

$$Lik = \prod_{i=1}^{n} pdf(x_i|\mu,\sigma)$$

Which values of μ and σ make this product as small as possible for the given data?

In practice we can't work with this likelihood function since it will easily go in underflow. Instead we work with the log likelihood where we sum the logs of the probabilities:

$$logLik = \sum_{i=1}^{n} log(pdf(x_i|\mu,\sigma))$$

In R we define the likelihood function as

```
logLikFn<- function(par= c(mu, sigma), x){
    mu<- par[1]
    sigma<- par[2]
    lik<- 0
    for(xi in x){
        lik<- lik + dnorm(xi, mean= mu, sd= sigma, log= TRUE)
    }
    return(lik)
}

## Some data:
set.seed(1234)
x<- rnorm(n= 10, mean= 2, sd= 3)

## And optimize by maximizing the logLik
optim(par= c(0, 1), logLikFn, x= x, method= 'BFGS', control= list(fnscale= -1))

$par
[1] 0.8504075 2.8340301
$value
[1] -24.60649

## Compare to ML in R
library(MASS)
fit<- fitdistr(x, 'normal')
     mean          sd
  0.8505278   2.8340610
 (0.8962088) (0.6337153)

[1] -24.60649 # fit$loglik
```

N.B.: Function `logLikFn` uses a for loop to emphasis the summation of the log likelihoods.

Optimize k-means clustering

In k-means clustering we group observations in a pre-defined number of groups k [1]. The task is to assign observations to groups so that the total within-group variation is minimazied. In symbols

$$\min_{C_1 \ldots C_K} \{\sum_{k=1}^{K} W(C_k)\}$$

We want to minimize the sum of $C_1 \ldots C_K$, *i.e.* the total group variation, where each group variation is given by the function W. We need to decide on the function W, how do we measure variation within groups? Most common choice is the mean squared Euclidean distance. For group C_k defined as

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i\prime \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i\prime j})^2$$

Which means: For each pair of group elements sum the squared distance over the p dimensions. Divide by the number of elements in group C_k.

This python implementation follows closely the code in the PuLP package for set partitioning.

```
import pulp
import math
import itertools

def sumEuclDist(cluster):
    """
    Return the mean sum of the Euclidean distances in cluster.
```

[1]See Introduction to Statistical Learning

```
    cluster: List of lists as a numeric matrix with rows element of the cluster and
    columns dimensions.
    """
    pairs= itertools.combinations(range(0, len(cluster)), 2)
    sed= 0
    for x in pairs:
        x1= cluster[x[0]]
        x2= cluster[x[1]]
        sed += eucliDist(x1, x2)
    return sed / len(cluster)

def eucliDist(x1, x2):
    """ Euclidean distance between vectors x1 and x2
    """
    assert len(x1) == len(x2)
    ec= 0
    for i in range(0, len(x1)):
        ec += (x1[i] - x2[i])**2
    return math.sqrt(ec)
```

Prepare a toy dataset to cluster in two groups:

```
datamat= [
    [-0.36211972,  0.1518168],
    [0.08322877, -0.1724220],
    [0.32533235, -0.1639896],
    [-0.70370931, -0.1693356],
    [0.12873741, -0.2670113],
    [0.63788028,  1.1518168],
    [1.08322877,  0.8275780],
    [1.32533235,  0.8360104],
    [0.29629069,  0.8306644],
    [1.12873741,  0.7329887]]

max_clusters = 2
max_cluster_size = len(datamat) - 1
```

Begin resolving, as explained in the PuLP docs, this strategy is computationally demanding and it will not perform well on large dataset. This is just for illustration about how to solve a partioning problem.

```
#create list of all possible clusters
possible_clusters = [tuple(c) for c in pulp.allcombinations(range(0, len(datamat)),
                                        max_cluster_size) if len(c) > 1]

#create a binary variable to state that a cluster setting is used
xvar = pulp.LpVariable.dicts(
    name= 'cluster',           ## prefix to the name of each LP variable created
    indexs= possible_clusters, ## keys to the dictionary of LP variables
    lowBound = 0,              ## Lower/Upper bound of the variable values
    upBound = 1,               ##
    cat = pulp.LpInteger)      ## Type of variable (default is contnuous)

## To each cluster assign the sum of euclidean distances, i.e. how diverse is this cluster
## We want to find the pairs of cluster which have the smallest sum distances.
clst= []
for cluster in possible_clusters:
    dat= [datamat[i] for i in cluster]
    clst.append(sumEuclDist(dat) * xvar[cluster])
```

```
## Define problem
cluster_model = pulp.LpProblem(name= "Clustering Model", sense= pulp.LpMinimize)
## Objective function
cluster_model += sum(clst)

## Add contraints:
## =============
#specify the maximum number of clusters
cluster_model += \
    sum([xvar[cluster] for cluster in possible_clusters]) <= max_clusters, \
    "Maximum_number_of_clusters"

#An element must seated at one and only one cluster
for el in range(0, len(datamat)):
    cluster_model += sum([xvar[cluster] for cluster in possible_clusters
                            if el in cluster]) == 1, "Must_seat_%s"%el

## Solve and print results
cluster_model.solve()

print("The choosen clusters are out of a total of %s:"%len(possible_clusters))
for cluster in possible_clusters:
    if xvar[cluster].value() == 1.0:
        print(cluster, sumEuclDist([datamat[i] for i in cluster]))

((0, 1, 2, 3, 4), 1.128967284173976)
((5, 6, 7, 8, 9), 1.1289672841739762)
```

Check result with R ’s kmeans function

```
datamat<- data.frame(
    x= c(-0.36211972, 0.08322877, 0.32533235, -0.70370931, 0.12873741,
        0.63788028, 1.08322877, 1.32533235, 0.29629069, 1.12873741),
    y= c(0.1518168, -0.1724220, -0.1639896, -0.1693356, -0.2670113, 1.1518168,
        0.8275780, 0.8360104, 0.8306644, 0.7329887))
plot(datamat)

kmeans(datamat, 2)
K-means clustering with 2 clusters of sizes 5, 5
...
Clustering vector:
 [1] 2 2 2 2 2 1 1 1 1 1
```

Optimizing lasso regression

Lasso regression optimizes regression coefficients to make the RSS as small as possible, just like linear regression. Lasso, however, penalizes large effect sizes (large coefficients) and possibly sets the to zero if advantageous to minimize RSS. The penalty is stated in such way that there is fixed ”amount” of effect size to distribute in the best way across coeffcients. In symbols

$$\min_{\beta}\{\sum_{i=1}^{n}(y_i - (\beta_0 + \sum_{j=1}^{p}\beta_j x_{ij}))^2\}$$

Constrained by

$$\sum_{j=1}^{p} |b_j| \leq s$$