DATA 1010
In-class exercises
Samuel S. Watson
12 November 2018

Problem 1

Find the distance from the plane $3x + 2y + z = 6$ to the point $P = (4, 7, 1)$.

Solution

The vector $[3, 2, 1]$ is perpendicular to the given plane, so moving t units directly away from some point in the plane means adding the vector

$$t\frac{[3,2,1]}{|[3,2,1]|} = t\frac{[3,2,1]}{\sqrt{13}}$$

to that point. The value of the function $3x + 2y + z$ is 6 for any point in the plane, and adding $t\frac{[3,2,1]}{\sqrt{13}}$ increases that value by $t\sqrt{13}$:

$$3\left(x + \frac{3t}{\sqrt{13}}\right) + 2\left(y + \frac{2t}{\sqrt{13}}\right) + 1\left(z + \frac{t}{\sqrt{13}}\right) = 3x + 2y + z + \frac{9t + 4t + t}{\sqrt{13}} = 6 + t\sqrt{13}.$$

Since the value of $3x + 2y + z$ is equal to $3(4) + 2(7) + 1(1) = 27$ at the point P, we can solve $t\sqrt{13} = 27 - 6$ to find that the distance is $\boxed{21/\sqrt{13}}$.

Problem 2

Find the distance from the hyperplane $\{\mathbf{x} \in \mathbb{R}^n : \boldsymbol{\beta} \cdot \mathbf{x} - \alpha = 0\}$ to the point $\mathbf{x}$.

Solution

Generalizing the idea we developed in the previous problem, we see that the distance function is

$$\frac{|\boldsymbol{\beta} \cdot \mathbf{x} - \alpha|}{|\boldsymbol{\beta}|}.$$

Problem 3

Simulate data for a binary classification problem in the plane for which the two classes can be separated by a line. Write a Julia function for finding the thickest slab which separates the two classes.

Solution

Let us describe the separating slab as $\{\mathbf{x} \in \mathbb{R}^2 : -1 \leq \boldsymbol{\beta} \cdot \mathbf{x} - \alpha \leq 1\}$. The width of this slab is $2/|\boldsymbol{\beta}|$, by the result in Problem 2.

Suppose that the separating line is $\boldsymbol{\beta} \cdot \mathbf{x} - \alpha = 0$. We can check whether a point is on the correct side of the slab by checking whether $\boldsymbol{\beta} \cdot \mathbf{x} - \alpha \geq 1$ for points of class 1 and less than or equal to -1 for points of class -1. More succinctly, we can check whether $y_i(\boldsymbol{\beta} \cdot \mathbf{x}_i - \alpha) \geq 1$ for all $i = 1, \ldots, n$ (assuming that the sample points are $(\mathbf{x}_i, y_i)$ where i ranges from 1 to n).

So, we are looking for the values of $\boldsymbol{\beta}$ and α which minimize $|\boldsymbol{\beta}|$ subject to the conditions $y_i(\boldsymbol{\beta} \cdot \mathbf{x}_i - \alpha) \geq 1$ for all $1 \leq i \leq n$. This is a **constrained** optimization problem, since we are looking to maximize the value of a function over a domain defined by some constraining inequalities.

Constrained optimization is a ubiquitous problem in applied mathematics, and numerous solvers exist for them. Most of these solvers are written in low-level languages like C and have bindings for the most popular high-level

languages (Python, R, Julia, MATLAB, etc.). No solver is uniformly better than the others, so solving a constrained optimization problem often entails trying different solvers to see which works best for your problem.

Julia has a package which makes this process easier by providing a single interface for problem encoding (`JuMP`). Let's begin by sampling our points and looking at a scatter plot. We go ahead and load the `Ipopt` package, which provides the solver we'll use.

```
using Plots, JuMP, Ipopt, Random; Random.seed!(1234);
gr(aspect_ratio=1)
function samplepoint(μ=[3,3])
    class = rand([-1,1])
    if class == -1
        X = randn(2)
    else
        X = μ + [1 -1/2; -1/2 1] * randn(2)
    end
    (X,class)
end
n = 100
samples = [samplepoint() for i=1:n]
x₁s = [x₁ for ((x₁,x₂),y) in samples]
x₂s = [x₂ for ((x₁,x₂),y) in samples]
ys = [y for (x,y) in samples]
scatter(x₁s,x₂s,group=ys)
```

Next we describe our constrained optimization problem in `JuMP`.

```
# problem data is stored in a `Model` object:
m = Model(solver=IpoptSolver(print_level=0))
# we add variables to the model with the `@variable` macro
@variable(m,β[1:2])
@variable(m,α)
# we add constraints with the `@constraint` macro
for (x,y) in samples
    @constraint(m,y*(β⋅x - α) ≥ 1)
end
# we add the objective function with the @objective macro
# we also have to specify whether its a minimization or
# maximization problem
@objective(m,Min,β[1]^2+β[2]^2)
# Once the problem is encoded, it can be solved,
solve(m)
# which makes the optimizing values of the variables
# retrievable via `getvalue`
β,α = getvalue(β), getvalue(α)
```

Finally, we plot our separating line:

```
l(x₁) = (α - β[1]*x₁)/β[2] xs = [-2,5] plot!(xs,[l(x₁) for x₁ in xs],label="")l
```

Problem 4

Now suppose that the data are not separable by a plane. Explain why

$$L(\boldsymbol{\beta}, \alpha) = \lambda|\boldsymbol{\beta}|^2 + \frac{1}{n}\sum_{i=1}^{n}\left[1 - y_i(\boldsymbol{\beta}\cdot\mathbf{x}_i - \alpha)\right]_+$$

is a reasonable quantity to minimize. (Note: u_+ means $\max(0, u)$, and λ is a parameter of the loss function).

Solution

Minimizing the first term is the same as minimizing the value of $|\boldsymbol{\beta}|$, which was the *hard-margin* objective function we saw in Problem 3. The second term penalizes points which are not on the right side of the slab (in units of margin widths; points on the slab midline get a penalty of 1, on the wrong edge of the slab they get a penalty of 2, and so on).

Problem 5

Simulate some overlapping data and minimize the loss function given in Problem 4. Choose the value of λ using cross-validation.

Solution

We begin by generating our samples. We make the means closer to create overlap:

```
samples = [samplepoint([1,1]) for i=1:n]
x₁s = [x₁ for ((x₁,x₂),y) in samples]
x₂s = [x₂ for ((x₁,x₂),y) in samples]
ys = [y for (x,y) in samples]
scatter(x₁s,x₂s,group=ys)
```

Next we define our loss function, including a version that takes $\boldsymbol{\beta}$ and α together in a single vector called `params`.

```
L(λ,β,α,samples) = λ*norm(β)^2 + 1/n*sum(max(0,1-y*(β⋅x - α)) for (x,y) in samples)
L(λ,params,samples) = L(λ,params[1:end-1],params[end],samples)
```

Since this optimization problem is unconstrained, we can use the `Optim` package to do the optimization. We define a function `SVM` which returns the $\boldsymbol{\beta}$ and α which minimize the empirical loss:

```
import Optim
function SVM(λ,samples)
    params = Optim.optimize(params->L(λ,params,samples),ones(3),Optim.BFGS()).minimizer
    params[1:end-1], params[end]
end
```

To choose λ, we write some functions to do cross-validation:

```
function errorrate(β,α,samples)
    count(y*(β⋅x-α) < 0 for (x,y) in samples)
end
function CV(λ,samples,i)
    β,α = SVM(λ,[samples[1:i-1];samples[i+1:end]])
    x,y = samples[i]
    y*(β⋅x - α) < 0
end
function CV(λ,samples)
    mean(CV(λ,samples,i) for i=1:length(samples))
end
```

Finally, we optimize over λ:

```
λ = Optim.optimize(λ->CV(first(λ),samples),1/1000,1/4).minimizer
β,α = SVM(λ,samples)
(x₁) = (α - β[1]x₁)/β[2]
xs = [-2,3]
scatter(x₁s,x₂s,group=ys)
plot!(xs,[(x₁) for x₁ in xs],label="")
```