homework6

December 13, 2022

1. Below is code to run a GPR. We continue to assume that $y_i = h(x_i) + \epsilon_i$ and assume that the target h is "smooth".

Effectively, we write down the (prior) distribution of our target at the data inputs $h(x_i)$ and at a collection of grid points $h(g_i)$ as a multivariate normal/Gaussian, and compute the the posterior

$$h(g_i)|h(x_i) = y_i - \epsilon \sim MVN(\mu, \Sigma),$$

which also happens to be a mutlivariate Gaussian (with fairly easy to compute mean vector μ and covariance structure Σ). The normality and the covariance in the prior is the result of doing basis expansion, where each basis vector is $\phi_i(x) = f_Z(\frac{x-c_i}{\lambda})$, where $f_Z(\cdot)$ is the standard normal pdf and c_i is the mean and λ is the standard deviation of a normal. Infinite basis expansion with finite computation. The **Kernel Trick**.

There are at least two tuning parameters to deal with here, the spread of the normal basis functions, λ, and the variance of the ϵ errors, σ. We might learn these from a cross-validation routine, which gets called *Empirical Bayes*.

```
rbf.k <- function(x,y,lambda)
  return(exp(-1/(2*lambda^2)*(sum((x-y)^2))))

target <- function(x)
  return(log(x+.1)+sin(5*pi*x))

gpreg <- function(x, y, lam, sig, design) {
  \# Evaluates mean and covariance of GP at grid of points on [0,1]
  \# Inputs:
  \#   x, y: input and output values of data set
  \#   lam: smoothing parameter in RBF kernel
  \#   sig: error standard deviation of y
  \#   design: grid of points to evaluate the GP
  \# Returns:
  \#   mean=posterior mean, vars=posterior variance, and design=evaluation points
  n <- length(y)
  x <- as.matrix(x)
  design <- as.matrix(design)
  m <- nrow(design)
  Sigma <- matrix(0,nrow=n+m, ncol=n+m)
  all <- rbind(x, design)
  for (i in 1:nrow(Sigma)) {
    for (j in i:nrow(Sigma))
```

```
      Sigma[i,j] <- rbf.k(all[i,], all[j,], lam) -> Sigma[j,i]
  }
  S11 <- Sigma[1:n, 1:n]
  S12 <- Sigma[1:n, (n+1):ncol(Sigma)]
  S21 <- Sigma[(n+1):ncol(Sigma), 1:n]
  S22 <- Sigma[(n+1):ncol(Sigma),(n+1):ncol(Sigma)]
  inv <- S21%*%solve(S11+sig^2*diag(n))
  mean <- inv%*%y
  cov <- S22-inv%*%S12
  vars <- diag(cov)
  return(list(mean=mean, vars=vars))
}


\#Sample Usage
n <- 10
x <- runif(n)
y <- c()
sig <- .1
for (i in 1:n) y[i] <- target(x[i])
y <- y + rnorm(n, 0, sig)

design <- seq(0,1,.01)
truth <- c()
for (i in 1:length(design)) truth[i] <- target(design[i])

gpfit <- gpreg(x, y, .1, sig, design)
plot(c(0,1),c(min(gpfit$mean-2*sqrt(gpfit$vars)), max(gpfit$mean+2*sqrt(gpfit$vars))),
  t="n", xlab="x", ylab="y")
points(x,y)
lines(design, gpfit$mean)
lines(design, gpfit$mean + 2*sqrt(gpfit$vars), col="blue")
lines(design, gpfit$mean - 2*sqrt(gpfit$vars), col="blue")
lines(design, truth, lty=3, col="red")
legend(0, 2, c("Truth", "Estimate", "Credible Bounds"), c("black", "red", "blue"))
```

We can also do this in higher dimesions. What shows up in the covariance is the squared Euclidian distance of the design points, so this works in any dimension (and the kernel trick means that we only need to compute the distance in the original d-dimensional data space, even though we are doing basis expansion.

Here, we do it with a linear response surface, with the size of the points reflecting the value of the response variable. We need to be careful with the size of the grid, especially with my unoptimized code. My grid is 25 by 25 and it still takes a few seconds to run.

This one is overfit!

```
a <- seq(0,1, length.out = 25)
truth <- matrix(0, nrow=25, ncol=25)
k <- 1
```

```
grid <- matrix(0, nrow=25^2, ncol=2)
for (i in 1:25) {
  for (j in 1:25) {
    truth[i,j] <- 3 + 5 * a[i] - 2 * a[j]
    grid[k,] <- c(a[i],a[j])
    k <- k + 1
  }
}
```

```
x <- cbind(runif(50), runif(50))
y <- 3 + 5 * x[,1] - 2 * x[,2] + rnorm(50,0,1)
image(truth)
points(x, cex=y)
gpfit <- gpreg(x,y, .1, 1, grid)
image(matrix(gpfit$mean, ncol=25, byrow=TRUE))
points(x, cex=y)
```

Find a good smoothness level for both this problem here and revisit the response surface that we used the additive model on in a previous homework, this time fitting that data with a GPR.

[41]:
```
import pymc3 as pm
from pymc3.distributions.timeseries import GaussianRandomWalk
from scipy import optimize
from theano import shared

import numpy as np

X = np.random.uniform(0,1,(50, 2))
y = 3 + 5 * X[:,0]**2 + np.sin(X[:,1])*10 + np.random.normal(0, np.sqrt(0.3))
```

Let's create a model with a shared parameter for specifying different levels of smoothing. We use very wide priors for the "mu" and "tau" parameters.

[42]:
```
LARGE_NUMBER = 1e5

model = pm.Model()
with model:
    smoothing_param = shared(0.9)
    mu = pm.Normal("mu", sigma=LARGE_NUMBER)
    tau = pm.Exponential("tau", 1.0 / LARGE_NUMBER)
    z = GaussianRandomWalk("z", mu=mu, tau=tau / (1.0 - smoothing_param),␣
 ↪shape=y.shape)
    obs = pm.Normal("obs", mu=z, tau=tau / smoothing_param, observed=y)
```

Let's also make a helper function for inferring the most likely values of z.

[43]:
```
def infer_z(smoothing):
    with model:
```

```
        smoothing_param.set_value(smoothing)
        res = pm.find_MAP(vars=[z])
        return res["z"]
```

[44]:
```
smoothing = 0.5
z_val = infer_z(smoothing)
print(z_val)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
[ 6.37342913  7.28873523  6.34630367  6.80060536  6.91575622  9.57418674
  9.54674344  9.17459877 11.02474033 11.63640592  9.84853638 11.39743636
  8.74249963  8.64534766  8.27629919  8.47924614  9.27264347  8.51622468
  9.08332711 10.94984438 11.65906464 11.22302169 11.52752424 10.16231861
 10.24959172 10.08296068 12.41904215 10.93286525 11.01706365 11.21768051
  8.75088947  8.80541668  7.84820485  9.2676931  10.0800529   8.98637043
 11.05686474 10.51137917  8.90559508  9.05338069  8.32992169  9.56533373
 11.10514493  9.14252526  9.36579002  9.82545917  8.82458511 10.84647595
 11.10009294 10.47216014]
```

I will assume that "the response surface that we used the additive model on in a previous homework" refers to Homework 3, Question 3.

[45]:
```
X = np.random.uniform(0,1,(200,2))
y = 5 + X[:,0]**2 + np.sin(X[:,1])*10 + np.random.normal(0, np.sqrt(0.3))
```

[46]:
```
smoothing = 0.5
z_val = infer_z(smoothing)
print(z_val)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
[ 6.37342913  7.28873523  6.34630367  6.80060536  6.91575622  9.57418674
  9.54674344  9.17459877 11.02474033 11.63640592  9.84853638 11.39743636
  8.74249963  8.64534766  8.27629919  8.47924614  9.27264347  8.51622468
  9.08332711 10.94984438 11.65906464 11.22302169 11.52752424 10.16231861
 10.24959172 10.08296068 12.41904215 10.93286525 11.01706365 11.21768051
  8.75088947  8.80541668  7.84820485  9.2676931  10.0800529   8.98637043
 11.05686474 10.51137917  8.90559508  9.05338069  8.32992169  9.56533373
 11.10514493  9.14252526  9.36579002  9.82545917  8.82458511 10.84647595
 11.10009294 10.47216014]
```

2. k-NN and the kernel trick.

Recall (if you have a linear algebra background) the *dot product* between two vectors $\mathbf{x}$ and $\mathbf{y}$ is given as $\mathbf{x} \cdot \mathbf{y} = \sum_{\mathbf{i=1}}^{\mathbf{p}} \mathbf{x_i y_i}$ and the squared norm is $\|x\|^2 = x \cdot x$

a. Show that the kernel function

$$K(x, y) = x \cdot y + ||x||^2||y||^2$$

for $x, y \in \mathbb{R}^2$ corresponds to augmenting the data space with a single extra feature, $x_3 = x_1^2 + x_2^2$, so that $\phi : \mathbb{R}^2 \to \mathbb{R}^3$.

$K(x, y) = \sum_{i=1}^{p} x_i y_i + (\sqrt{x_1^2 + \cdots + x_n^2}^2)(\sqrt{y_1^2 + \ldots y_n^2}^2) = \sum_{i=1}^{p} x_i y_i + (x_1^2 + \ldots x_n^2)(y_1^2 + \cdots + y_n^2) = \sum_{i=1}^{p} x_i y_i + x_1^2 y_1^2 + \ldots x_1^2 y_n^2 + \ldots x_n^2 y_n^2$.

b. The k-NN classifier only relies on the Euclidian distance between points:

$$\|\mathbf{x} - \mathbf{y}\|^{1/2} = \sqrt{\sum_i (x_i - y_i)^2}$$

Show that this algorithm is **kernelizable**, i.e. the only way the algorithm uses the data is contained in the **Gram matrix**, the matrix of dot (inner) products $\mathbf{G} := [\mathbf{x}_i \cdot \mathbf{x}_j]_{i,j=1}^n$

By page 35 in ESL, the metric for the k-NN classifier is $K_k(x, x_0) = I(||x - x_0|| \leq ||x_{(k)} - x_0||)$.

c. Here's my function for k-NN, with a slightly different generation of the distance matrix (less elegant but easier to see what it's doing).

```
dist.comp <- function(pt, data) {
  dists <- c()
  for (i in 1:nrow(data)) {
    dists[i] <- sqrt(sum((pt-data[i,])^2))
  }
  return(dists)
}

knn <- function(pt, data, labels, k) {
  \# length of pt should match ncol of data
  dists <- dist.comp(pt, data)
  inds <- which(dists <= sort(dists)[k])
  names(which.max(table(labels[inds])))
}
```

and it implemented on the iris data (note: this code is pretty slow!)

```
pred.iris <- c()
for (i in 1:150) {
  pred.iris[i] <- knn(iris[i,1:4], iris[-i,1:4], iris[-i,5], 4)
}
pred.iris
```

Consider the following data set

```
set.seed(47)
r <- c(runif(75,0,.5), runif(75,.5,1))
theta <- runif(150,0,2*pi)
x <- r*cos(theta); y <- r*sin(theta)
classes <- c('in', 'out')
```

```
label <- classes[(r>.5)+1]
plot(x,y, col="blue")
points(x[label=="in"], y[label=="in"], col='red')
```

Let's try 3-NN on this data

```
pred.bull <- c()
for (i in 1:150) {
  pred.bull[i] <- knn(c(x[i], y[i]), cbind(x[-i], y[-i]), label[-i], 3)
}
rbind(pred.bull, label)
mean(pred.bull==label)  \#true classification rate
plot(x,y, col="blue")
points(x[label=="in"], y[label=="in"], col='red')
points(x,y, cex=.5, col='blue')
points(x[pred.bull=="in"], y[pred.bull=="in"], col='red', cex=.5)
```

Not bad, but perfect is clearly attainable (given the data generation, the true decision boundary is the circle with radius .5)

Edit the distance function, replacing the Euclidean distance with its representation in terms of dot products, and replace those dot products with the kernel function given above (it's the same as augmenting the data, but I want you to use the kernel). Rerun this algorithm and discuss your findings.

Will this strategy always work? Or is it particular to something about this data set?

I assume that this strategy will always work if the substitution is truly equivalent.