

Homework_5

Nancy Zhang and Kyle Forrester

2025-10-13

Setup

First, we load in our data. This is the NOAA.new dataset from Canvas.

```
NOAA.new <- read.csv("/Users/nancyz/STAT486/Data/NewNOAA.csv") #replace with your path
x <- NOAA.new[,3] #temperature rise
y <- NOAA.new[,2] #rate of billion-dollar disasters
```

Edited smoother.pck This section contains the edited smoother.pck and its functions, with PRESS statistic in each smoother. The function chunks are fully commented, as this data was taken from Canvas/slides.

```
smoother.pck <- #received error that version of this package not available for this version of R
c("smoother.pck", "bin.mean", "gauss.mean", "gauss.reg", "gauss.mean.trunc",
  "gauss.reg.trunc", "my.hat.w")
bin.mean <-
function(x,y,nbin,xcol=2,do.plot=F)
{
  #order x and y
  o1<-order(x)
  x1<-x[o1]
  y1<-y[o1]
  #find min and max x
  r1<-range(x)
  #width of each bin = (max - min) / number of bins
  inc<-(r1[2]-r1[1])/nbin
  yvec<-NULL
  smat<-matrix(0,length(x1),length(x1))
  #for each bin:
  for(i in 1:nbin){
    #find min and max values for the bin
    bin.low<-r1[1]+(i-1)*inc
    bin.high<-r1[1]+i*inc
    #I1: true if x1 is within the current bin or a later bin

    I1<-x1>=bin.low
    #I2: true if x1 is within the current bin or an earlier bin

    if(i<nbin){
      I2<-x1<bin.high
    }else{
      I2<-x1<=(bin.high+200)
    }
    #I3: true if both I1 and I2 are true, meaning x1
    #is within the current bin
```

```

    I3<-as.logical(I1*I2)
    #find mean of the bucket
    yval<-mean(y1[I3])
    n1<-sum(I3)
    if(n1>0){
      #convert values in each bucket to the mean and add to matrix
      smat[I3, I3]<-1/n1
      yvec<-c(yvec,rep(yval,n1))
    }
  }
n99<-length(x1)
#calculate degrees of freedom, delta1, delta2, and R based on number of data points
dferror<-length(x1)-sum(diag(2*smat-smat%*(t(smat))))
delta1<-sum(diag(t(diag(n99)-smat)%*(diag(n99)-smat)))
R<-t(diag(n99)-smat)%*(diag(n99)-smat)
delta2<-2*sum(diag(R%*R))
#plot if enabled
if(isTRUE(do.plot)) lines(x1,as.numeric(smat%*y1),col=xcol)
#calculate residuals and use PRESS function on the sum of squares
ypred<-y
ypred[o1]<-smat%*y1
resid<-y1-(smat%*y1)
hii<-diag(smat); den<-1-hii; den[den<=1e-8]<-1e-8; loo<-as.numeric(resid)/den; loo[!is.finite(loo)]<-0
PRESS<-sum(loo^2)
#return
list(smat=smat,df=sum(diag(smat)),dferror=dferror,delta1=delta1,delta2=delta2,resid=resid,pred=ypred,x=x1)
}

gauss.mean <-
function(x,y,lambdaxcol=3,do.plot=T)
{
  #order data
  o1<-order(x)
  x1<-x[o1]
  y1<-y[o1]
  r1<-range(x)
  smat<-NULL
  #for each data point
  n1<-length(x1)
  for(i in 1:n1){
    #take a sample with the data point as the mean and lambda as the standard deviation
    v1<-dnorm(x1,x1[i],lambda)
    #normalize vector by dividing by mean and add to matrix
    v1<-v1/sum(v1)
    smat<-rbind(smat,v1)
  }
  yhat<-smat%*y1
  #plot if enabled
  if(do.plot){
    lines(x1,yhat,col=xcol)
  }
  n99<-length(x1)

```

```

#calculate degrees of freedom, delta1, delta2, and R based on number of data points
dferror<-length(x1)-sum(diag(2*smat-smat%*(t(smat))))
delta1<-sum(diag(t(diag(n99)-smat)%*(diag(n99)-smat)))
R<-t(diag(n99)-smat)%*(diag(n99)-smat)
delta2<-2*sum(diag(R%*R))
#calculate residuals and use PRESS function on the sum of squares
resid<-y1-smat%*y1
ypred<-y
ypred[o1]<-smat%*y1
hii<-diag(smat); den<-1-hii; den[den<=1e-8]<-1e-8; loo<-as.numeric(resid)/den; loo[!is.finite(loo)]<-0
PRESS<-sum(loo^2)
#return
list(smat=smat,df=sum(diag(smat)),dferror=dferror,delta1=delta1,delta2=delta2,resid=resid,pred=ypred,pr

}

gauss.reg <-
function(x,y,lambda,xcol=4,do.plot=T)
{
#order data
o1<-order(x)
x1<-x[o1]
y1<-y[o1]
r1<-range(x)
smat<-NULL
n1<-length(x1)
#for each data point
for(i in 1:n1){
#take a sample with the data point as the mean and lambda as the standard deviation
v1<-dnorm(x1,x1[i],lambda)
#normalize vector by dividing by mean
v1<-v1/sum(v1)
#generate hat matrix
H1<-my.hat.w(x1,v1)
smat<-rbind(smat,H1[i,])
}
yhat<-smat%*y1
#plot of enabled
if(do.plot){
lines(x1,yhat,col=xcol)
}
n99<-length(x1)
#calculate degrees of freedom, delta1, delta2, and R based on number of data points
dferror<-length(x1)-sum(diag(2*smat-smat%*(t(smat))))
delta1<-sum(diag(t(diag(n99)-smat)%*(diag(n99)-smat)))
R<-t(diag(n99)-smat)%*(diag(n99)-smat)
delta2<-2*sum(diag(R%*R))
#calculate residuals
resid<-y1-smat%*y1
ypred<-y
ypred[o1]<-smat%*y1
hii<-diag(smat); den<-1-hii; den[den<=1e-8]<-1e-8; loo<-as.numeric(resid)/den; loo[!is.finite(loo)]<-0
PRESS<-sum(loo^2) #added PRESS statistic calculation

```

```

#return
list(smat=smat,df=sum(diag(smat)),dferror=dferror,delta1=delta1,delta2=delta2,resid=resid,pred=ypred,pr

}

gauss.mean.trunc <-
function(x,y,lambda,nnn,xcol=5,do.plot=T)
{
  #order data
  o1<-order(x)
  x1<-x[o1]
  y1<-y[o1]
  r1<-range(x)
  smat<-NULL
  n1<-length(x1)
  trunc.val<-n1-nnn
  #for each data point
  for(i in 1:n1){
    #generate sample with sd lambda and mean x1[i]
    v1<-dnorm(x1,x1[i],lambda)
    #order vector
    o2<-order(v1)
    #remove values less than the threshold for truncating
    thresh<-v1[o2[trunc.val]]
    v1<-v1*(v1>thresh)
    #normalize and add to matrix
    v1<-v1/sum(v1)
    smat<-rbind(smat,v1)
  }
  yhat<-smat%*%y1
  #graph if enabled
  if(do.plot){
    lines(x1,yhat,col=xcol)
  }
  n99<-length(x1)
  #calculate degrees of freedom, delta1, delta2, and R based on number of data points
  dferror<-length(x1)-sum(diag(2*smat-smat%*%(t(smat))))
  delta1<-sum(diag(t(diag(n99)-smat)%*%(diag(n99)-smat)))
  R<-t(diag(n99)-smat)%*%(diag(n99)-smat)
  delta2<-2*sum(diag(R%*%R))
  #predict y values and calculate residuals
  resid<-y1-smat%*%y1
  ypred<-y
  ypred[o1]<-smat%*%y1
  hii<-diag(smat); den<-1-hii; den[den<=1e-8]<-1e-8; loo<-as.numeric(resid)/den; loo[!is.finite(loo)]<-0
  PRESS<-sum(loo^2) #added PRESS statistic calculation
  #return
  list(smat=smat,df=sum(diag(smat)),dferror=dferror,delta1=delta1,delta2=delta2,resid=resid,pred=ypred,pr

}

gauss.reg.trunc <-
function(x,y,lambda,nnn,xcol=6,do.plot=T)
{

```

```

#order data
o1<-order(x)
x1<-x[o1]
y1<-y[o1]
r1<-range(x)
smat<-NULL
n1<-length(x1)
trunc.val<-n1-nnn
#for all data
for(i in 1:n1){
  #generate sample with sd lambda and mean x1[i]
  v1<-dnorm(x1,x1[i],lambda)
  #order vector
  o2<-order(v1)
  #remove values less than threshold for truncating
  thresh<-v1[o2[trunc.val]]
  v1<-v1*(v1>thresh)
  #normalize vector
  v1<-v1/sum(v1)
  #generate hat matrix
  H1<-my.hat.w(x1,v1)
  smat<-rbind(smat,H1[i,])
}
yhat<-smat%*%y1
if(do.plot){ #plot if enabled
  lines(x1,yhat,col=xcol)
}
n99<-length(x1)
#calculate degrees of freedom, delta1, delta2, and R based on number of data points
dferror<-length(x1)-sum(diag(2*smat-smat%*%(t(smat))))
delta1<-sum(diag(t(diag(n99)-smat)%*%(diag(n99)-smat)))
R<-t(diag(n99)-smat)%*%(diag(n99)-smat)
delta2<-2*sum(diag(R%*%R))
#predict y values and calculate residuals
resid<-y1-smat%*%y1
ypred<-y
ypred[o1]<-smat%*%y1
hii<-diag(smat); den<-1-hii; den[den<=1e-8]<-1e-8; loo<-as.numeric(resid)/den; loo[!is.finite(loo)]<-0
PRESS<-sum(loo^2) #added PRESS statistic calculation
#return
list(smat=smat,df=sum(diag(smat)),dferror=dferror,delta1=delta1,delta2=delta2,resid=resid,pred=ypred,pr
}

my.hat.w <-
function(x,wt){
  x1<-cbind(1,x)
  x1%*%solve(t(x1)%*%diag(wt)%*%x1)%*%t(x1)%*%(diag(wt))
}

```

Comparing Smoothers with PRESS and Plots This section is to apply each smoother to the NOAA.new data using the parameters given in slide 17. The reference from the slides is provided first, then the updated code with more readability is below.

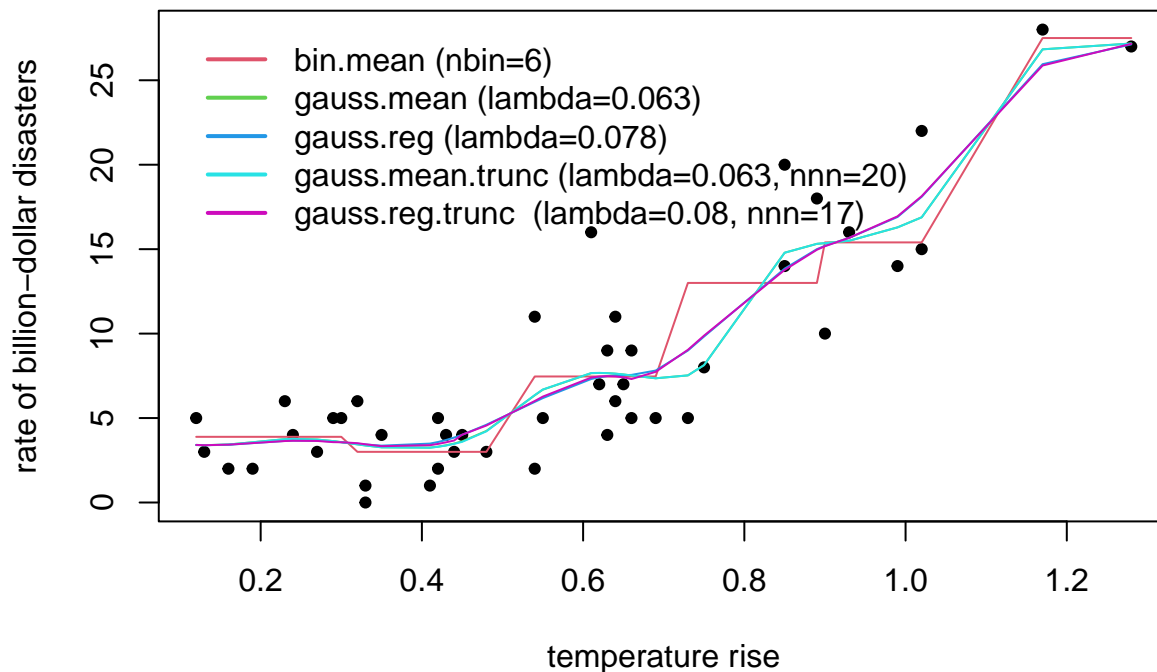
```
#plot(NOAA1[,3],NOAA1[,2],xlab="temperature rise",ylab="rate of billion dollar
#weather disasters")
#dum<-bin.mean(NOAA1[,3],NOAA1[,2],6)
#dum<-gauss.mean(NOAA1[,3],NOAA1[,2],.063)
#gauss.reg(NOAA1[,3],NOAA1[,2],.078,do.plot=T)
#gauss.mean.trunc(NOAA1[,3],NOAA1[,2],.063,20,do.plot=T)
#gauss.reg.trunc(NOAA1[,3],NOAA1[,2],.08,17,do.plot=T)
#lines(lowess(NOAA1[,3],NOAA1[,2]),col=7)
#lines(smooth.spline(NOAA1[,3],NOAA1[,2]),col=8)
```

Plot We have a nice plot to show the differences between all our smoothers.

```
#base plot where we'll have all our smoothers on
plot(x,y,xlab = "temperature rise",ylab = "rate of billion-dollar disasters",pch = 19,cex = 0.7)

#refit with plotting turned on for ALL
fit_bin<-bin.mean(x,y,nbin=6,xcol=2,do.plot=TRUE) #bin.mean function
fit_gm<-gauss.mean(x,y,lambda=0.063,xcol=3,do.plot=TRUE) #gauss.mean function
fit_gr<-gauss.reg(x,y,lambda=0.078,xcol=4,do.plot=TRUE) #gauss.reg function
fit_gm_trunc<-gauss.mean.trunc(x,y,lambda=0.063,nnn=20,xcol=5,do.plot = TRUE) #gauss mean truncated fun
fit_gr_trunc<-gauss.reg.trunc(x,y,lambda=0.08,nnn=17,xcol=6,do.plot =TRUE) #gauss reg truncated function

#legend for the lines
legend("topleft",inset = 0.02,bty = "n",lwd=2,lty=rep(1, 5),col=c(2,3,4,5,6),
      legend = c("bin.mean (nbin=6)",
                  "gauss.mean (lambda=0.063)",
                  "gauss.reg (lambda=0.078)",
                  "gauss.mean.trunc (lambda=0.063, nnn=20)",
                  "gauss.reg.trunc (lambda=0.08, nnn=17)"))
```



```
#comparing PRESS in a table:
press_table<-data.frame(
```

```

smoother=c("bin.mean (nbin=6)",
           "gauss.mean (lambda=0.063)",
           "gauss.reg (lambda=0.078)",
           "gauss.mean.trunc (lambda=0.063, nnn=20)",
           "gauss.reg.trunc (lambda=0.08, nnn=17)"),
PRESS=c(fit_bin$press,
        fit_gm$press,
        fit_gr$press,
        fit_gm_trunc$press,
        fit_gr_trunc$press))
print(press_table)

```

```

##              smoother    PRESS
## 1          bin.mean (nbin=6) 632.1817
## 2      gauss.mean (lambda=0.063) 461.9978
## 3      gauss.reg (lambda=0.078) 560.7915
## 4 gauss.mean.trunc (lambda=0.063, nnn=20) 461.8456
## 5 gauss.reg.trunc (lambda=0.08, nnn=17) 561.7968

```

Greedy Random Search Function This function references the pseudo-code from slide 19 of Class 9. Its purpose is to find better values of the adjustable parameters, based on minimizing PRESS.

```

greedy.search<-function(fit.eval,init.gen,perturb.gen,in.bounds,max.no.improve=300,target=Inf){
#parameters: fit.eval - function to use for fit, init.gen - function to generate parameters for fit, perturb.gen - function to perturb parameters
#in.bounds - function to check whether value is within the given bounds, max.no.improve - max number of tries
#do a random fit, choose parameters = pi0
pi0<-init.gen()
crit0<-fit.eval(pi0) #check and save fit
best.par<-pi0
best.crit<-crit0
noimp<-0
#while fit not as good as criterion-> we use patience OR target
while(noimp<max.no.improve && best.crit>target)
{#randomly perturb parameters, pi0 + small increments
  cand<-perturb.gen(best.par)
  #check if parameters meet any constraints
  if(in.bounds(cand)){
    #check fit
    ccrit<-fit.eval(cand)
    #if better save new parameters as pi0
    if(ccrit+1e-12<best.crit)
    {best.par<-cand
     best.crit<-ccrit
     noimp<-0
    }
  }
  else
  {noimp<-noimp+1
  }
}
else{noimp<-noimp+1
}
}
#output pi0 and final fit criterion
list(par=best.par,press=best.crit,tries=noimp)}

```

Performance of the Greedy Random Search Function We'll now see how well the greedy random search function minimizes PRESS for each smoother. Using the implementations highlighted above, we use a few minimal wrappers for each of the different smoothers.

```
#final greedy search chunk to to minimize PRESS for each smoother
#we use wrappers with similar deisng and structure
#bin.mean has discrete int nbin, gauss.* continuous lambda
#ranges and step sizes
dx<-diff(range(x))
n<-length(x)
lam.range<- pmax(1e-6,c(0.02, 0.30)*dx) #width of search window
nbin.range<-c(2L,max(6L,min(50L,floor(n/3)))) #keep reasonable bins
nnn.range <-c(5L,max(10L,min(floor(0.6*n), n-1L))) #kept neighbors

#helper methods
clamp<-function(v,lo,hi) pmin(hi,pmax(lo,v))
rand_int<-function(lo,hi) sample(seq.int(lo,hi), 1)

#optimize nbin for bin.mean with greedy search
fit.eval.bin<-function(p) {
  #clamp and round to keep nbin valid
  nbin<-as.integer(clamp(round(p$nbin),nbin.range[1],nbin.range[2]))
  #return PRESS for these params
  bin.mean(x,y,nbin = nbin,do.plot = FALSE)$press
}

#random valid start for nbin
init.gen.bin<-function()list(nbin=rand_int(nbin.range[1],nbin.range[2]))
#try +/- 1 bin each step
perturb.gen.bin<-function(p)list(nbin=as.integer(p$nbin+sample(c(-1L,1L),1)))
#keep only finite, in-range nbin
in.bounds.bin<- function(p)is.finite(p$nbin)&&p$nbin>=nbin.range[1] && p$nbin<=nbin.range[2]

set.seed(486)
#run greedy for bin.mean
best_bin<-greedy.search(
fit.eval=fit.eval.bin,
init.gen=init.gen.bin,
perturb.gen=perturb.gen.bin,
in.bounds=in.bounds.bin,
max.no.improve=400)

#optimize lambda for gauss.mean
fit.eval.gm<-function(p) {
  #keep lambda in allowed window
  lam<-clamp(as.numeric(p$lambda),lam.range[1],lam.range[2])
  #PRESS for gauss.mean
  gauss.mean(x,y,lambda=lam,do.plot=FALSE)$press
}

#random lambda start
init.gen.gm<-function() list(lambda=runif(1,lam.range[1],lam.range[2]))
#lognormal step to nudge lambda up/down
perturb.gen.gm<-function(p) list(lambda=p$lambda*exp(rnorm(1,0,0.15)))
#lambda must be finite and positive
in.bounds.gm<-function(p) is.finite(p$lambda)&&p$lambda>0
```



```

#run greedy for gauss.mean
best_gm<-greedy.search(
  fit.eval=fit.eval.gm,
  init.gen=init.gen.gm,
  perturb.gen=perturb.gen.gm,
  in.bounds=in.bounds.gm,
  max.no.improve=400
)

#optimize lambda for gauss.reg
fit.eval.gr<-function(p){
  #clamp lambda to search range
  lam<-clamp(as.numeric(p$lambda),lam.range[1],lam.range[2])
  #PRESS for gauss.reg
  gauss.reg(x,y,lambda=lam,do.plot=FALSE)$press
}

#random start for lambda
init.gen.gr<-function() list(lambda=runif(1,lam.range[1],lam.range[2]))
#gentle multiplicative jitter on lambda
perturb.gen.gr<-function(p) list(lambda=p$lambda*exp(rnorm(1,0,0.15)))
#must be finite and > 0
in.bounds.gr<-function(p) is.finite(p$lambda)&& p$lambda>0

#run greedy for gauss.reg
best_gr<-greedy.search(
  fit.eval=fit.eval.gr,
  init.gen=init.gen.gr,
  perturb.gen=perturb.gen.gr,
  in.bounds=in.bounds.gr,
  max.no.improve=400
)

#optimize lambda and nnn for gauss.mean.trunc
fit.eval.gmt<-function(p){
  #clamp both params
  lam<-clamp(as.numeric(p$lambda),lam.range[1],lam.range[2])
  nnn<-as.integer(clamp(round(p$nnn),nnn.range[1],nnn.range[2]))
  #PRESS for truncated mean smoother
  gauss.mean.trunc(x,y,lambda=lam,nnn=nnn,do.plot=FALSE)$press
}

#random start for both params
init.gen.gmt<-function() list(
  lambda=runif(1,lam.range[1],lam.range[2]),
  nnn=rand_int(nnn.range[1],nnn.range[2])
)

#alternate stepping lambda or nnn
perturb.gen.gmt<-function(p){
  if(runif(1)<0.5){
    #jitter lambda, keep nnn
    list(lambda=p$lambda*exp(rnorm(1,0,0.15)),nnn=p$nnn)
  }else{
    #move nnn by a small integer step
    list(lambda=p$lambda,nnn=as.integer(p$nnn+sample(c(-2L,-1L,1L,2L),1)))
  }
}

```

```

    }
  }
  #basic validity checks for both params
  in.bounds.gmt<-function(p){
    ok1<-is.finite(p$lambda)&& p$lambda>0
    ok2<-is.finite(p$nnn)&&round(p$nnn)>=nnn.range[1]&&round(p$nnn)<=nnn.range[2]
    ok1&&ok2
  }
  #run greedy for gauss.mean.trunc
  best_gmt<-greedy.search(
    fit.eval=fit.eval.gmt,
    init.gen=init.gen.gmt,
    perturb.gen=perturb.gen.gmt,
    in.bounds=in.bounds.gmt,
    max.no.improve=500
  )
  #optimize lambda and nnn for gauss.reg.trunc
  fit.eval.grt<-function(p){
    #clamp both params
    lam<-clamp(as.numeric(p$lambda),lam.range[1],lam.range[2])
    nnn<-as.integer(clamp(round(p$nnn),nnn.range[1],nnn.range[2]))
    #PRESS for truncated local regression
    gauss.reg.trunc(x,y,lambda=lam,nnn=nnn,do.plot=FALSE)$press
  }
  #random start for both params
  init.gen.grt<-function() list(
    lambda=runif(1,lam.range[1],lam.range[2]),
    nnn=rand_int(nnn.range[1],nnn.range[2])
  )
  #alternate between nudging lambda or nnn
  perturb.gen.grt<-function(p){
    if(runif(1)<0.5){
      #multiplicative jitter for lambda
      list(lambda=p$lambda*exp(rnorm(1,0,0.15)),nnn=p$nnn)
    }else{
      #small integer step for nnn
      list(lambda=p$lambda,nnn=as.integer(p$nnn+sample(c(-2L,-1L,1L,2L),1)))
    }
  }
  #validity checks for both params
  in.bounds.grt<-function(p){
    ok1<-is.finite(p$lambda)&& p$lambda>0
    ok2<-is.finite(p$nnn)&&round(p$nnn)>=nnn.range[1]&&round(p$nnn)<=nnn.range[2]
    ok1&&ok2
  }
  #run greedy for gauss.reg.trunc
  best_grt<-greedy.search(
    fit.eval=fit.eval.grt,
    init.gen=init.gen.grt,
    perturb.gen=perturb.gen.grt,
    in.bounds=in.bounds.grt,
    max.no.improve=500
  )

```

```

#collect and print results
best_tbl <- data.frame(
  smoother = c("bin.mean", "gauss.mean", "gauss.reg", "gauss.mean.trunc", "gauss.reg.trunc"),
  PRESS     = c(best_bin$press, best_gm$press, best_gr$press, best_gmt$press, best_grt$press),
  nbin      = c(as.integer(best_bin$par$nbin), NA, NA, NA, NA),
  lambda    = c(NA,
    as.numeric(best_gm$par$lambda),
    as.numeric(best_gr$par$lambda),
    as.numeric(best_gmt$par$lambda),
    as.numeric(best_grt$par$lambda)),
  nnn       = c(NA, NA, NA,
    as.integer(best_gmt$par$nnn),
    as.integer(best_grt$par$nnn))
)

best_tbl <- best_tbl[order(best_tbl$PRESS), ]
rownames(best_tbl) <- NULL
print(best_tbl)

```

```

##      smoother  PRESS nbin   lambda nnn
## 1    gauss.mean 460.6817   NA 0.05713687  NA
## 2      bin.mean 488.8604    8         NA  NA
## 3    gauss.reg 527.3064   NA 0.15981955  NA
## 4 gauss.mean.trunc 544.3961   NA 0.14635581 15
## 5 gauss.reg.trunc 557.9847   NA 0.10810784 25

```

Conclusion The slide (“hard-coded”) parameters are illustrative defaults, not tuned to this dataset. Our greedy search explicitly minimizes PRESS on these data, so it naturally picks different bandwidths/neighbor counts when the x-range, point density, or noise level differs from the slide example. Because the Gaussian bandwidth lambda is in the units of delta.temp, extending the time span to 2024 and changing the scale/density of x shifts the optimal settings—so a difference between defaults and greedy picks is expected and desirable.

gauss.reg (local-linear) likely performed best because it cancels first-order bias and therefore reduces boundary bias—a big deal when there are fewer points at the coolest/warmest delta temp values. The greedy run also selected a relatively wide bandwidth, which smooths over year-to-year spikes while tracking the smooth, mostly monotone trend in disasters, lowering the PRESS (leave-one-out) error. Truncated variants add an extra knob (nnn) that helps when x is extremely uneven, but the design here isn’t that irregular, so the simpler non-truncated local-linear achieved the best bias–variance trade-off.