Part 9:Kernel Smoothing and Density Estimation

Mark Bebbington

AgHortC 2.09A

m.bebbington@massey.ac.nz

Reading for this Part

- Again, there is very little in the text on this part (Sections 4.2 and 8.2 have some related material.
- You can get the following PDF files from the library:
  - Altman NS (1992) An introduction to kernel and nearest –neighbor nonparametric regression. American Statistician 46, 175-185
  - Schucany WR (2004) Kernel smoothers: An overview of curve estimators for the first graduate course in nonparametric statistics. Statistical Science 19, 66-675
  - Sheather SJ (2004) Density Estimation. Statistical Science 19, 588-597
- Other electronic resources:
  - http://www.quantlet.com/mdstat/scripts/anr/html/anrhtmlnode11.html
  - http://en.wikipedia.org/wiki/Kernel_smoother
  - http://en.wikipedia.org/wiki/Kernel_density_estimation
- Books:
  - Silverman BW (1986) Density Estimation for Statistics and Data Analysis. Chapman and Hall, London.
  - Wand MP, Jones MC (1995) Kernel Smoothing. Chapman and Hall, London

Smoothing

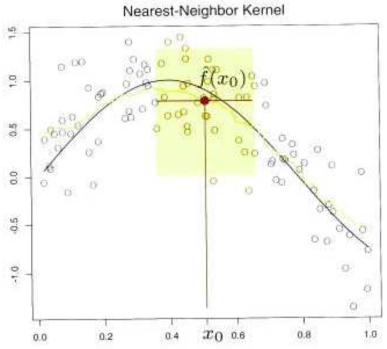
- Can be seen as a class of regression techniques
  - Estimate the regression function f(X) by fitting a different (parameter values) but simple (same formula) model separately at each query point x_0
  - Use only points close to x_0 to fit function there, so that the estimated function is *smooth*
  - Localization achieved by a weighting function, or *kernel*, $K_\lambda(x_0,x_i)$
    - This assigns a weight to x_i based on its distance from x_o
    - λ controls the width of the neighborhood
  - The only parameter to be determined is λ, although the model is the entire (training) dataset

One-Dimensional Kernels

- k-nearest neighbour average

$$\hat{f}(x) = \frac{1}{k}\sum_{i\in N_k(x)} y_i$$

- where $N_k(x)$ is the set of k points nearest to x
- The result is discontinuous in x (bumpy)



Nearest neighbor smoothing in NumPy

```
def nnsmooth(x0,x,y,N):
    ## nearest neighbour smoothing of y = f(x)
    ## with N nearest neighbours, at points x0
    fhat = zeros(shape(x0),dtype=float)
    for i in range(len(x0)):
        d = abs(x-x0[i]) # distance from each point
                         # to evaluation point
        temp = concatenate((d,y),axis=1)
        temp = temp[argsort(temp[:,0],0)] # sort
                                          # output data in
                                          # order of distance
        fhat[i] = (1/float(N))*sum(temp[0:N,1])
                                  # average N nearest outputs
    return fhat
```

Example

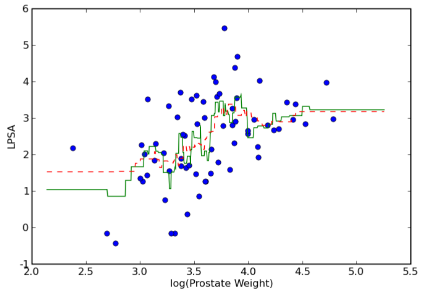
We saw in the prostate cancer data that predictor #2, prostate weight, appeared to be the most significant predictor of cancer. How does it do on its own?

```
data = loadtxt('prostate_train.txt')
## reformat data as X and Y
p = 8
data = data.reshape(-1,p+1)
y = data[:,p:p+1]
x = data[:,1:2] # only predictor 2, prostate weight
# NB – we are using arrays here.

fineness = 1000 ## number of evaluation points
x0 = 0.9*min(x) + (1.1*max(x) -
                           0.9*min(x))*(arange(fineness))/fineness
               # evaluation points, evenly spaced from below the
               # smallest x to above the largest x
```

Results

```
fhat5 = nnsmooth(x0,x,y,5)
fhat10 = nnsmooth(x0,x,y,10)
figure(1)
plot(x,y,'o',x0,fhat5,'-',x0,fhat10,'--')
ylabel('LPSA')
xlabel(‘log(Prostate Weight)')
show()
```

- Program in ‘prostate_nnk.py’



Kernel-weighted average

Instead of giving all points in the neighborhood equal weight, use

$$\hat{f}(x) = \frac{\sum_{i=1}^{N} K_\lambda(x_0, x_i) y_i}{\sum_{i=1}^{N} K_\lambda(x_0, x_i)}$$

where, for example,

$$K_\lambda(x_0, x) = \begin{cases} 0.75(1-(x_0-x)^2/\lambda^2), & |x-x_0| < \lambda \\ 0, & \text{otherwise} \end{cases}$$

is the Epanechnikov quadratic kernel.

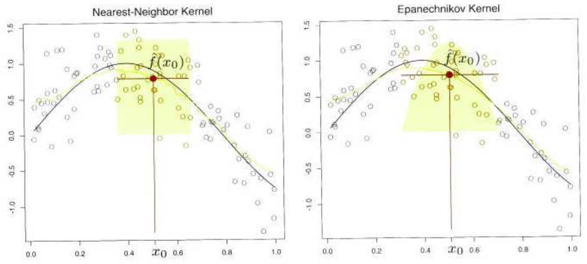


FIGURE 6.1. *In each panel* 100 *pairs* x_i, y_i *are generated at random from the blue curve with Gaussian errors:* $Y = \sin(4X) + \varepsilon$, $X \sim U[0,1]$, $\varepsilon \sim N(0, 1/3)$. *In the left panel the green curve is the result of a 30-nearest-neighbor running-mean smoother. The red point is the fitted constant* $\hat{f}(x_0)$, *and the orange shaded circles indicate those observations contributing to the fit at* x_0. *The solid orange region indicates the weights assigned to observations. In the right panel, the green curve is the kernel-weighted average, using an Epanechnikov kernel with (half) window width* $\lambda = 0.2$.

f(x) is now continuous. As the target moves, points enter the nbhd with initial weight zero, which slowly increases.

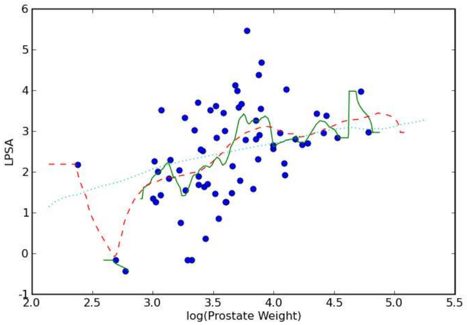
Kernel-weighted average smoothing in NumPy

```
def kwa_smooth(x0,x,y,lam):
    ## kernel weighted average smoothing of y = f(x)
    ## with bandwidth lam, at points x0
    lsq = lam**2
    fhat = zeros(shape(x0),dtype=float)
    for i in range(len(x0)):
        temp1 = 0.75*(1-(x-x0[i])**2/lsq)
        temp2 = (abs(x-x0[i])<lam) # 1 if < lam,
                                   # otherwise 0
        K = sum(temp1*temp2) # note this is array
                             # multiplication
        Ky = sum(temp1*y*temp2) # note this is array
                                #multiplication
        fhat[i] = Ky/K
    return fhat
```

Example

Prostate cancer, weight predictor only:

solid line, λ=0.1, dashed line, λ=0.33, dotted line, λ=1



- Program in `prostate_kwa.py'

Choice of kernel

Besides the Epanechnikov and Gaussian (normal pdf) kernels, there is also the tri-cube kernel

$$K_\lambda(x_0, x) = \begin{cases} \left(1 - |(x_0 - x)/\lambda|^3\right)^3, & |x - x_0| < \lambda \\ 0, & \text{otherwise} \end{cases}$$

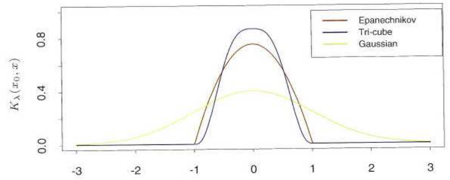


FIGURE 6.2. *A comparison of three popular kernels for local smoothing. Each has been calibrated to integrate to 1. The tri-cube kernel is compact and has two continuous derivatives at the boundary of its support, while the Epanechnikov kernel has none. The Gaussian kernel is continuously differentiable, but has infinite support.*

Exercise 9.1

Repeat the kernel-weighted average smoothing on the prostate data (lecture slides 9.10 – 9.11) with the Gaussian and tri-cube kernels. Compare the results at various values of λ.

Adaptive Neighborhoods

- We have used a fixed λ for the neighborhood size, whereas the nearest neighbor smoothing window adapts to the local density of points x_i.
- To do this with kernels, we replace λ by a function $h_\lambda(x_0)$.
  - $h_\lambda(x_0) = \lambda$ is the previous case
  - $h_\lambda(x_0) = |x_0 - x_{[k]}|$, where $x_{[k]}$ is the kth closest neighbor to x_0 gives a k-nearest neighbor effect.

Details (picky, picky,…)

- λ has to be determined.
  - A large λ implies lower variance (average over more observations) but greater bias
- Metric windows (constant $h_\lambda(x)$) tend to give constant bias, but variance is inversely proportional to local density.
  - Nearest neighbor windows behave in the opposite fashion (constant variance, bias inversely proportional to local density).
- Boundary issues
  - Metric windows contain less points, while nearest neighbor windows get wider
- There are other popular kernels, including the Gaussian density, using the s.d. as a substitute for the window width

Local linear regression

Solves a separate weighted least squares regression problem at each target point x_0:

$$\min_{\alpha(x_0),\beta(x_0)} \sum_{i=1}^{N} K_\lambda(x_0, x_i)\left(y_i - \alpha(x_0) - \beta(x_0)x_i\right)^2$$

with estimate $\hat{f}(x_0) = \hat{\alpha}(x_0) + \hat{\beta}(x_0)x_0$

Denote $b(x)^T=(1,x)$. Let B be the Nx2 regression matrix with ith row $b(x_i)^T$, and $W(x_0)$ the NxN diagonal matrix with ith diagonal element $K_\lambda(x_0,x_i)$. Then

$$\begin{aligned}\hat{f}(x_0) &= b(x_0)^T\left(B^T W(x_0) B\right)^{-1} B^T W(x_0) y \\ &= \sum_{i=1}^{N} l_i(x_0) y_i\end{aligned}$$

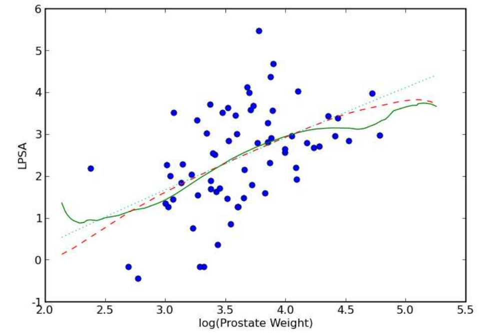
which shows that the estimate is linear in the y_i. The $l_i(x_0)$, which do not involve y, are called the *equivalent kernel.*

Local linear regression in NumPy

```
def llr_smooth(x0,x,y,lam):
    ## local linear regression of y = f(x)
    ## with bandwidth lam, at points x0
    lsq = lam**2
    y = matrix(y)
    B = matrix(concatenate((ones(shape(x)),x),axis=1))
    BT = transpose(B)
    fhat = zeros(shape(x0),dtype=float)
    for i in range(len(x0)):
        temp1 = 0.75*(1-(x-x0[i])**2/lsq)*(abs(x-x0[i])<lam)
        W = matrix(zeros((len(x),len(x))))
        for j in range(len(x)):
            W[j,j] = temp1[j,0]  # create a diagonal matrix
        bx0T = matrix([1.,x0[i]])
        fhat[i] = bx0T*(linalg.inv(BT*W*B))*BT*W*y
    return fhat
```

Example

Prostate cancer, weight predictor only:
solid line, λ=1, dashed line, λ=2, dotted line, λ=4



- Program in \`prostate_llr.py'

Local polynomial regression

Fit local polynomial of degree d:

$$\min_{\alpha(x_0),\beta_j(x_0),j=1,\ldots,d} \sum_{i=1}^{N} K_\lambda(x_0,x_i)\left(y_i - \alpha(x_0) - \sum_{j=1}^{d}\beta_j(x_0)(x_i)^j\right)^2$$

with solution $\hat{f}(x_0) = \hat{\alpha}(x_0) + \sum_{j=1}^{d}\hat{\beta}_j(x_0)(x_0)^j$

Local linear regression tends to be biased in regions of curvature of the true function (\`trim the hills and fill the valleys'). Local quadratic regression corrects this, but the fits at the boundary become much less reliable.

Illustration

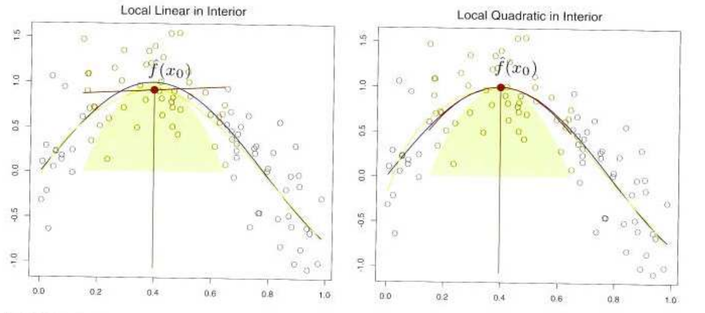


FIGURE 6.5. *Local linear fits exhibit bias in regions of curvature of the true function. Local quadratic fits tend to eliminate this bias.*

Selecting the Kernel width

- There is the usual bias-variance tradeoff:
  - If the window is narrow, the estimate is the average of a small number of y_i close to x_0, and has large variance, and small bias.
  - If the window is wide, then the variance is small due to the effects of averaging, but the bias will be worse, as distant x_i may have values very different from those at x_0.
- The effective degrees of freedom of the local linear regression smoother is trace(S_λ), where $\{S_\lambda\}_{i,j} = l_i(x_j)$.

Effective d.o.f. in NumPy

```
def llr_dof(x,lam):
    lsq = lam**2
    B = matrix(concatenate((ones(shape(x)),x),axis=1))
    BT = transpose(B)
    S = zeros((len(x),len(x)),dtype=float)
    for i in range(len(x)):
        temp1 = 0.75*(1-(x-x[i])**2/lsq)*(abs(x-x[i])<lam)
        W = zeros((len(x),len(x)))
        for j in range(len(x)):
            W[j,j] = temp1[j,0]
        bxT = matrix([1.,x[i]])
        S[i,:] = bxT*(linalg.inv(BT*W*B))*BT*W
    return sum(diag(S))
```

Prostate Data d.o.f

```
print 'lambda, d.o.f.'
for i in range(10):
    lam = (i+3.0)/4
    print lam,llr_dof(x,lam)
lambda, d.o.f.

0.75 4.35423010504
1.0 3.64949706815
1.25 3.1178048386
1.5 2.76182485867
1.75 2.52440544521
2.0 2.37118759479
2.25 2.26765149193
2.5 2.19959296045
2.75 2.15347585745
3.0 2.12306172736

Program in ‘prostate_llr_dof.py’
```

Kernel Density Estimation

- As the title implies, here we want to estimate the density function, direct from the data.
- Suppose we have a random sample $x_1,\ldots x_N$ drawn from a pdf $f_X(x)$, and want to estimate $f_X(x)$ at a point x_0.
  - A natural local estimate is

$$\hat{f}_X(x_0) = \frac{\# x_i \in \mathbb{N}(x_0)}{N\lambda}$$

where $\mathbb{N}(x_0)$ is a small neighborhood around x_o of width λ.

Exercise 9.2

Write a program to fit the `natural’ density estimate (lecture slide 9.24). Test it on the Systolic Blood Pressure variable (1st column) from the South African Heart Disease data.

Parzen estimate

The `natural’ estimate is bumpy, so we prefer the smooth estimate

$$\hat{f}_X(x_0) = \frac{1}{N\lambda}\sum_{i=1}^{N} K_\lambda(x_0, x_i)$$

which counts observations close to x_0 with greater weight.

A popular choice for K_λ is the Gaussian kernel

$$\begin{aligned} K_\lambda(x_0, x) &= \phi(|x - x_0| / \lambda) \\ &= \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-1}{2\lambda^2}(x - x_0)^2\right) \end{aligned}$$

and thus $\hat{f}_X(x) = N^{-1}\sum_{i=1}^{N} \phi_\lambda(x - x_i) = (\hat{F} * \phi_\lambda)(x)$ where ϕ_λ is the Gaussian density with mean zero and standard deviation λ.

The sample empirical distribution $\hat{F}$ puts mass 1/N at each observed x_i, so is `jumpy’. The convolution with ϕ_λ adds independent Gaussian noise to each observation.

Parzen estimate in p dimensions

Use the Gaussian product kernel,

$$\hat{f}_X(x_0) = \frac{1}{N(2\lambda^2\pi)^{p/2}}\sum_{i=1}^{N} \exp\left(-\frac{1}{2}\left(\|x_i - x_0\| / \lambda\right)^2\right)$$

Kernel density estimation in NumPy

```
def kde(x0,x,lam):
# kernel density estimation of sample x
# at points x0, bandwidth lam
    N = len(x)
    lsq = lam**2
    fhat = zeros(shape(x0),dtype=float)
    for i in range(len(x0)):
        K = (1/sqrt(2*pi))*exp(-((x-
  x0[i])**2)/(2*lsq))
        fhat[i] = sum(K)/(lam*N)
    return fhat
```

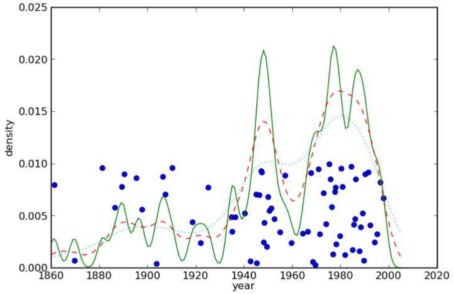
Example

`ruapehu.txt’ contains the onset (start) and stop times of eruptions (VEI > 0) at Ruapehu 1860 – 2006 (inclusive).

What is the density of onsets over the period 1860-2006?

```
data = loadtxt('ruapehu.txt')
data = data.reshape(-1,2)
# get onsets
x = data[:,0:1]
```

Solid line, λ = 2yrs
Dashed line, λ = 5yrs
Dotted line, λ = 10yrs



Program in ‘ruapehu_kde.py’

Choice of Kernel

- The exact choice of kernel is of minor importance compared to the choice of width (or “bandwidth”), λ. However:
  - Kernels with finite support have minor computational advantages, but can lead to areas of zero intensity, which is a major problem in some applications
  - Kernels with long tails may produce inadequate smoothing at short-medium distances, giving a uniform background with spikes around clusters.

Boundary effects

- If the kernel extends beyond the boundary, the density estimate is biased downwards, as there are no points to be picked up.
  - The best way to deal with this is to avoid it, by having data from at least λ beyond the boundary of the estimation region.
  - When this is not possible there are two solutions:
    - Artificially extend the dataset by reflecting the original data in the boundary
    - Weight the estimate inversely by the integral of the kernel over the observation region:

$$\hat{f}_X(x_0) = \frac{1}{N\lambda}\sum_{i=1}^{N} K_\lambda(x_0, x_i) \Big/ \int_{x \in A} K_\lambda(x_0, x)dx$$

Boundary effects (inverse weighting) in NumPy

```
from scipy.stats import *

def kde_bdy(x0,x,lam):
    # kernel density estimation of sample x at points x0, bandwidth lam
    # with weighted boundary condition
    # uses normal CDF from scipy.stats
    N = len(x)
    lsq = lam**2
    minx0 = min(x0)
    maxx0 = max(x0)
    fhat = zeros(shape(x0),dtype=float)
    for i in range(len(x0)):
        K = (1/sqrt(2*pi))*exp(-((x-x0[i])**2)/(2*lsq))
        Kint = norm.cdf((maxx0-x0[i])/lam)- norm.cdf((minx0-x0[i])/lam)
        fhat[i] = (sum(K)/(lam*N))/Kint
    return fhat
```

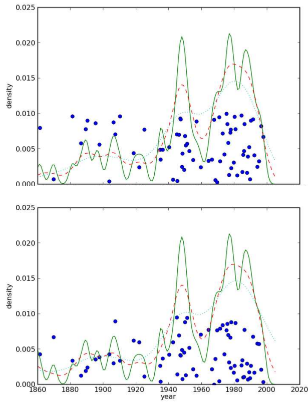
Example, Ruapehu onsets

minx = 1860, maxx = 2006

Without boundary correction

With boundary correction

•program in ‘ruapehu_kdebdy.py’



Selection of Bandwidth

Usual procedure is to use cross-validation, or `leave-one-out’:

- A data point x_0 is left out, and an estimate of $f(x_0)$ constructed from the remaining data points.
- The result, for a given λ, can be calculated using the Kullback-Leibler (KL) score

$$S = \sum_i \log \hat{f}_{-i}(x_i) - \int_{x\in A} \hat{f}(x)dx$$

where the “-i” subscript indicates an estimate not using the ith observation, and the integral should be a constant, given correct normalization.

- The value of λ selected is that maximizing S.

Selecting bandwidth in NumPy

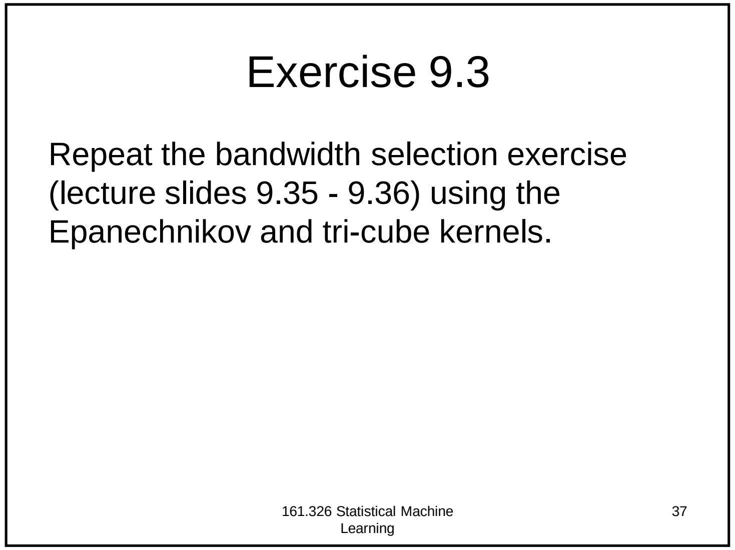
```
def KLscore(x,minx,maxx,lam):
    # calculates the Kullback-Liebler score for kernel
    # density estimation
    # of a sample x at bandwidth lam
    # using weighted boundary conditions at minx, maxx
    lsq = lam**2
    N = len(x)
    fhat = zeros(shape(x),dtype=float)
    for j in range(len(x)):
        x0 = concatenate((x[0:j],x[j+1:N]),axis=0)
        x1 = x[j:j+1]
        K = (1/sqrt(2*pi))*exp(-((x0-x1)**2)/(2*lsq))
        Kint = norm.cdf((maxx-x1)/lam)- norm.cdf((minx-
  x1)/lam)
        fhat[j] = (sum(K)/(lam*(N-1)))/Kint
    return sum(log(fhat))
```

Example: Ruapehu onset dates

```
minx = 1860
maxx = 2006
fineness = 100
minlam = 1.
maxlam = 20.
lam0 = minlam + (maxlam -
minlam)*(arange(fineness))/fineness

S = zeros(shape(lam0),dtype=float)
print 'lambda, S'
for i in range(len(lam0)):
    lam = lam0[i]
    S[i] = KLscore(x,minx,maxx,lam)
    print lam,S[i]
maxidx = argmax(S)
print 'optimum bandwidth =',lam0[maxidx]
```

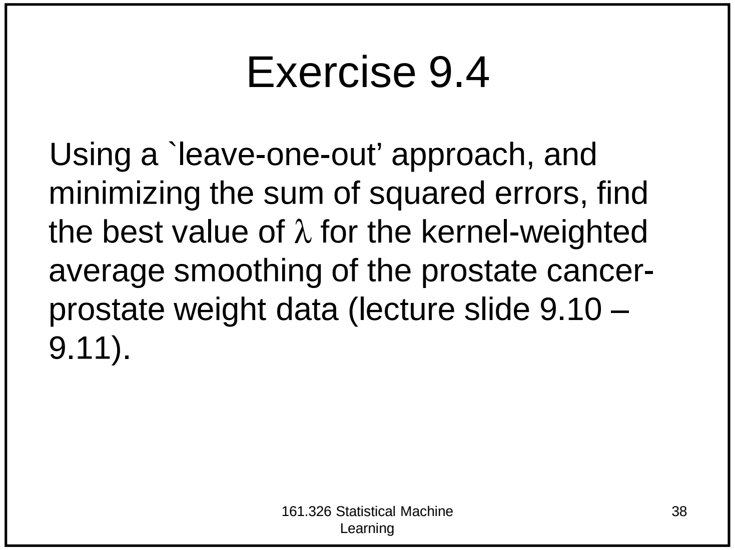
➢optimum bandwidth = 9.17

➢program in ‘ruapehu_kdebw.py’



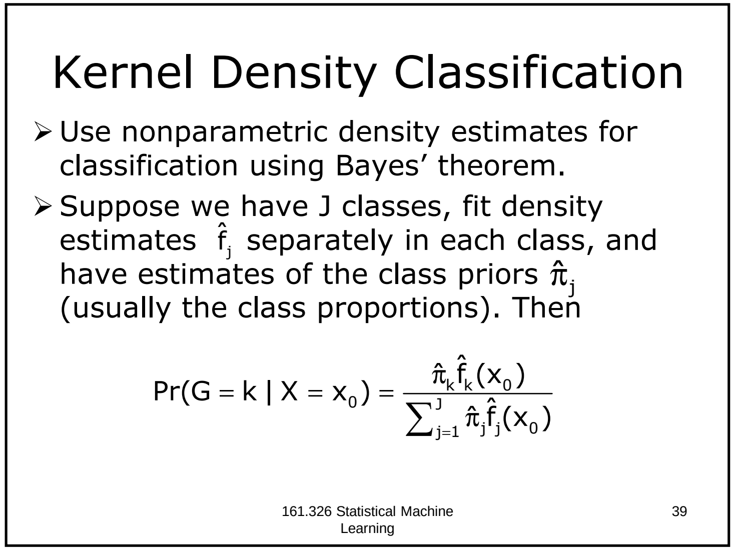
Exercise 9.3

Repeat the bandwidth selection exercise (lecture slides 9.35 - 9.36) using the Epanechnikov and tri-cube kernels.



Exercise 9.4

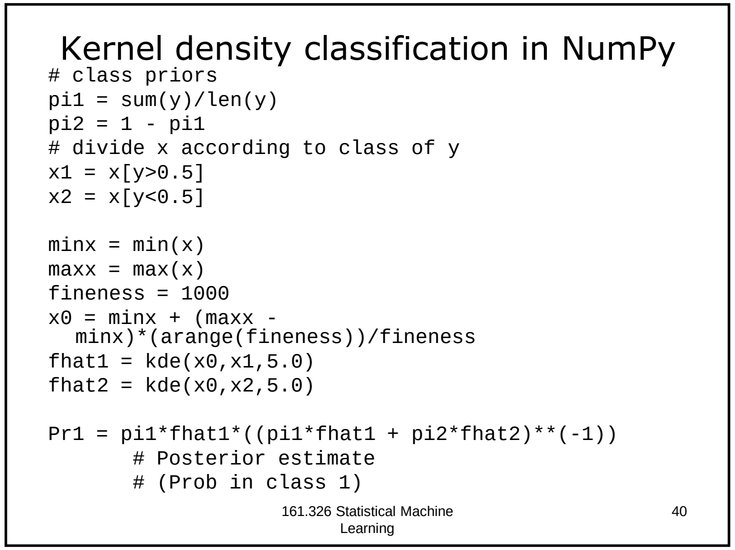
Using a `leave-one-out’ approach, and minimizing the sum of squared errors, find the best value of λ for the kernel-weighted average smoothing of the prostate cancer-prostate weight data (lecture slide 9.10 – 9.11).



Kernel Density Classification

- Use nonparametric density estimates for classification using Bayes’ theorem.
- Suppose we have J classes, fit density estimates $\hat{f}_j$ separately in each class, and have estimates of the class priors $\hat{\pi}_j$ (usually the class proportions). Then

$$\Pr(G = k \mid X = x_0) = \frac{\hat{\pi}_k \hat{f}_k(x_0)}{\sum_{j=1}^{J} \hat{\pi}_j \hat{f}_j(x_0)}$$



Kernel density classification in NumPy

```
# class priors
pi1 = sum(y)/len(y)
pi2 = 1 - pi1
# divide x according to class of y
x1 = x[y>0.5]
x2 = x[y<0.5]

minx = min(x)
maxx = max(x)
fineness = 1000
x0 = minx + (maxx -
  minx)*(arange(fineness))/fineness
fhat1 = kde(x0,x1,5.0)
fhat2 = kde(x0,x2,5.0)

Pr1 = pi1*fhat1*((pi1*fhat1 + pi2*fhat2)**(-1))
       # Posterior estimate
       # (Prob in class 1)
```

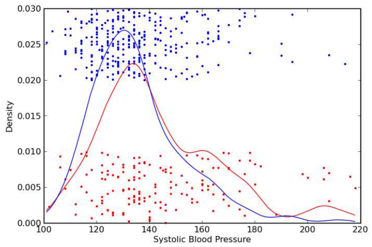
Example

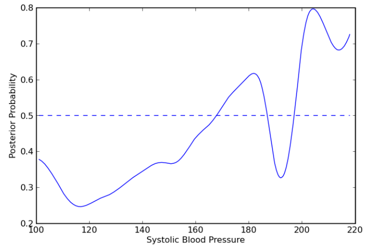
South African Heart Disease data.

- Classify Systolic Blood Pressure for presence/absence of coronary heart disease

```
# get data
p = 9
data = loadtxt('saheartdis.txt')
data = data.reshape(-1,p+1)
y = data[:,9:10] ## chd
x = data[:,0:1] ## systolic blood pressure

……

# jitter for plotting
y1 = random.rand(len(x1),1)/100
y2 = 0.02+random.rand(len(x2),1)/100

figure(1)
plot(x1,y1,'r.',x2,y2,'b.',x0,fhat1,'r-',x0,fhat2,'b-')
ylabel('Density')
xlabel('Systolic Blood Pressure')

figure(2)
plot(x0,Pr1,'b-',array([minx,maxx]),array([0.5,0.5]),'b--')
ylabel('Posterior Probability')
xlabel('Systolic Blood Pressure')
```

SA Heart Disease





Top: kernel density estimates for CHD = 1 (red), and 0 (blue)

Bottom: posterior probability for CHD = 1

$\lambda = 5$

- Program in 'heart_kdc.py'

Robustness

Note that if classification is the goal, good estimates of the densities are unnecessary. All that is needed is a good estimate of the posterior near the decision boundary (for 2 classes, $\{x:\Pr(G=1|X=x) = ½\}$).

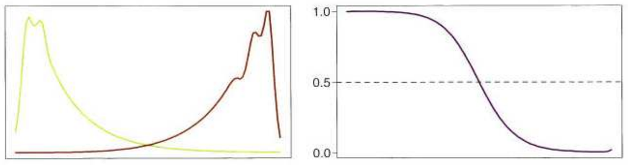


FIGURE 6.15. *The population class densities may have interesting structure (left) that disappears when the posterior probabilities are formed (right).*

Naïve Bayes Classifier

- Sometimes called "Idiot's Bayes".
- Useful when the dimension of the predictor space p is high, which makes density estimation unattractive.
- Assumes that, given a class G=j, the predictors X_k are independent: $$f_j(X) = \prod_{k=1}^{p} f_{j,k}(X_k)$$
- While this is generally not true, it does simply the estimation
  - The individual class-conditional marginal densities $f_{j,k}$ can each be estimated separately using 1-D kernel density estimates
  - If a component X_j of X is discrete, then a histogram estimate can be used instead, allowing us to mix variable types in a predictor vector.

Naïve Bayes in NumPy

```
def nbayesclass(X,y):
    ## naive Bayes classifier for y = {0,1}, X a matrix of p predictors
    p = X.shape[1]
    f0k = zeros(shape(X),dtype=float) # density estimate for y = 0, kth predictor
    f1k = zeros(shape(X),dtype=float) # density estimate for y = 1, kth predictor
    for k in range(p):
        x = X[:,k:k+1] # get kth predictor
        x1 = x[y>0.5]  # and divide according
        x0 = x[y<0.5]  # value of y
        lam = 0.05*(max(x) - min(x)) # bandwidth
        lsq = lam**2
        ## calculate density estimates
        for i in range(len(y)):
            K1 = (1/sqrt(2*pi))*exp(-((x1-x[i])**2)/(2*lsq))
            K0 = (1/sqrt(2*pi))*exp(-((x0-x[i])**2)/(2*lsq))
            f1k[i,k] = sum(K1)/(lam*len(x1))
            f0k[i,k] = sum(K0)/(lam*len(x0))
    ## multiply over predictors
    f1 = transpose(matrix(f1k[:,0]))
    f0 = transpose(matrix(f0k[:,0]))
    for k in range(p-1):
        f1 = multiply(f1,f1k[:,k+1:k+2])
        f0 = multiply(f0,f0k[:,k+1:k+2])
    return (f1 > f0) # classification, 0 or 1
```

Example

South African Heart Disease data, classifying for presence of CHD on basis of 9 predictors

```
data = loadtxt('saheartdis.txt')
data = data.reshape(-1,10)
## reformat data as x and y
y = data[:,9:10] ## chd
X = data[:,0:9]

nbc = nbayesclass(X,y)
errorrate = (0. + len(y) - sum(y == nbc))/len(y)
print 'errorrate =',errorrate
```

```
➢errorrate = 0.294372294372
```

program in ‘heart_nbc.py’