Accuracy of Prediction

Contents

Prediction

- Oftentimes interest lies in **predicting** the value of a variate (the **response** variate) given the values of one or more **explanatory** variates.

- We build a **response model** that encodes how that prediction is to be carried out

$$y = \mu(\mathbf{x}) + \text{error}$$

  - The explanatory variates $\mathbf{x} = (x_1, \ldots, x_p)$ are used to explain or predict the values of the response.
  - We use our observed data to estimate the function $\mu(\mathbf{x})$, yielding the **predictor function** $\widehat{\mu}(\mathbf{x})$.
  - This predictor function $\widehat{\mu}(\mathbf{x})$ is then use to predict y at any given value $\mathbf{x}$.

- **Example:** for simple linear regression we have $\mu(x) = \alpha + \beta x$ and we might estimate the parameters of the function using least squares:

$$\widehat{\mu}(x) = \widehat{\alpha} + \widehat{\beta} x$$

- But how do we know if our predictions are any good? And what does "good" even mean?

5.1 Accuracy of Prediction

- Quantifying the predictive accuracy of one or more models is an extremely important task in modern-day data endeavors

  - The suitability of a model may be determined entirely by its predictive accuracy.

- So how do we measure the accuracy of a model?

  - As a matter of convention, one usually measures the **inaccuracy** of a model's predictions.

- One measure of inaccuracy over the population $\mathcal{P}$ (of size N) is the **average prediction squared error** (APSE)

$$\frac{1}{N}\sum_{u\in\mathcal{P}}\left(y_u-\widehat{\mu}(\mathbf{x}_u)\right)^2$$

  - **Note** that the quantity above is proportional to the more familiar *residual sum of squares*

$$\sum_{i=1}^{N}\widehat{r}_i^2=\sum_{i=1}^{N}\left(y_i-\widehat{\mu}(\mathbf{x}_i)\right)^2$$

    which sometimes also referred to as the estimated *residual mean squared error*.

- This measure quantifies the distance between true and predicted values.
  - In particular, it is the average squared distance between a response observation and its corresponding prediction.

We might use this accuracy (inaccuracy!) measure to choose between competing models, as we will see in the sections to follow.

5.1.1 Example: Loblolly Pine Trees

- This dataset concerns the growth of Loblolly Pine Trees and contains three variates recorded for $N = 84$ trees
  - `height`: the tree height in feet
  - `age`: the age of the tree in years
  - `Seed`: the seed source for the tree

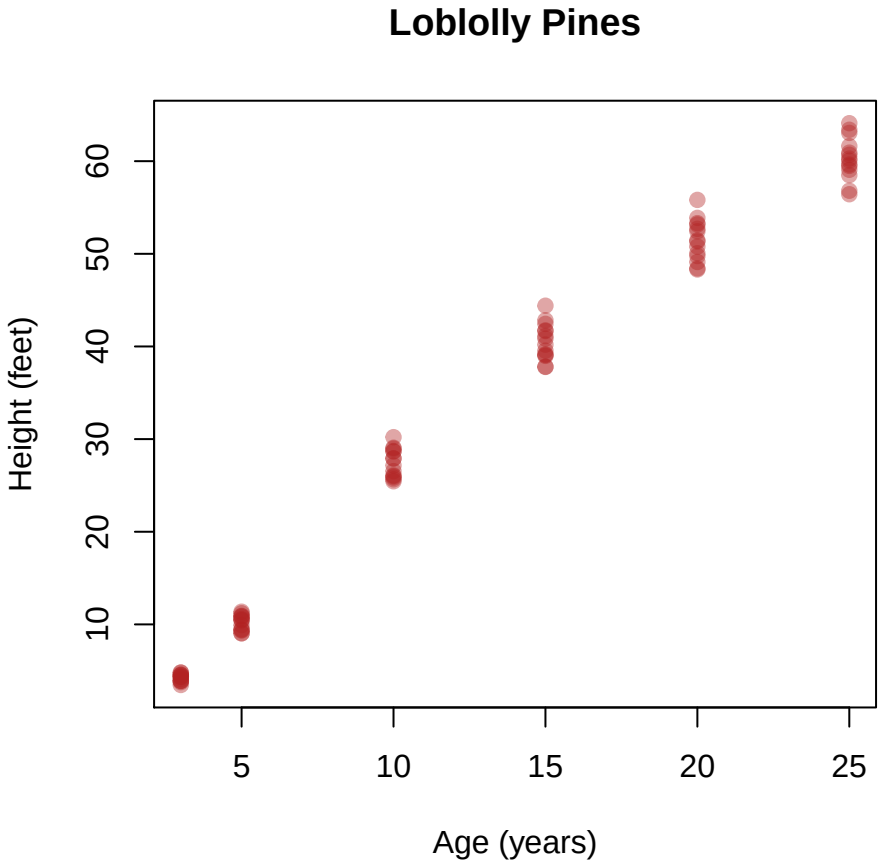
- Here is an excerpt of the data:

```
data(Loblolly)
head(Loblolly)
```

```
##    height age Seed
## 1    4.51   3  301
## 15  10.89   5  301
## 29  28.72  10  301
## 43  41.74  15  301
## 57  52.70  20  301
## 71  60.92  25  301
```

- And here is a visualization of the relationship between `height` and `age`:

```r
plot(Loblolly$age, Loblolly$height, xlab="Age (years)", ylab="Height (feet)",
     main = "Loblolly Pines",
     col=adjustcolor("firebrick", 0.4), pch=19)
```

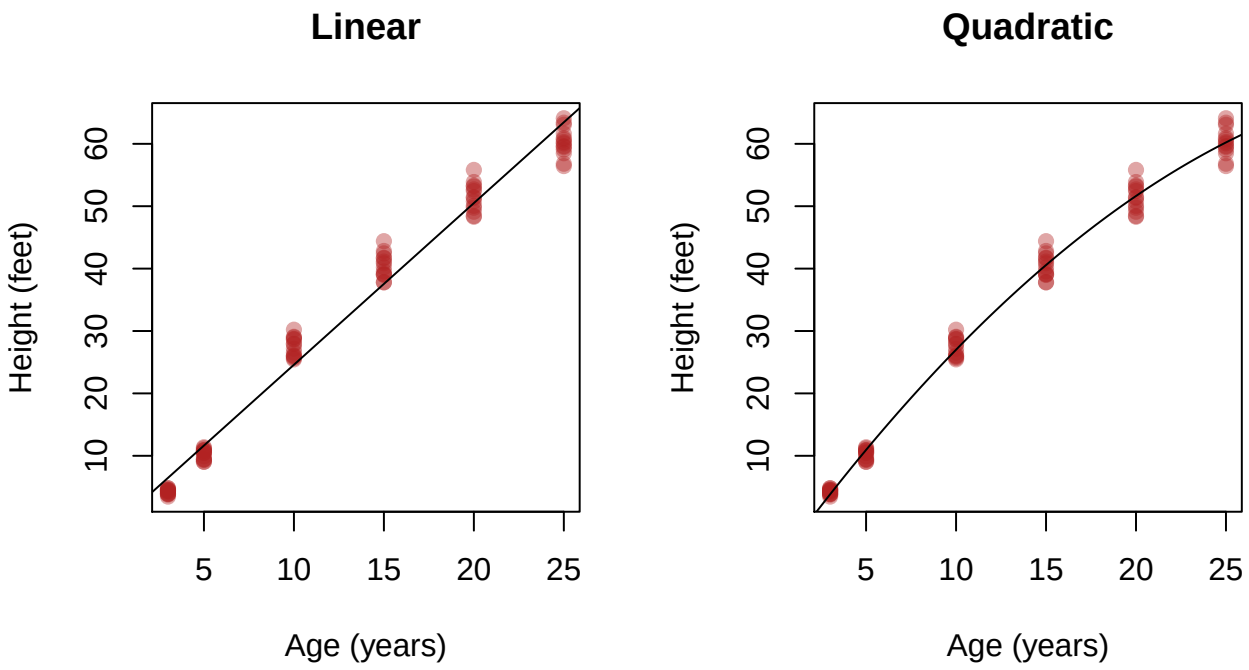


Fitting Polynomials

- We might consider fitting a linear or quadratic (in `age`) model to this data using a power transformation to make the data more linear.

```r
par(mfrow=c(1,2) )

plot(Loblolly$age[order(Loblolly$age)],
     Loblolly$height[order(Loblolly$age)],
     xlab="Age (years)", ylab="Height (feet)",
     main = "Linear",
     col=adjustcolor("firebrick", 0.4), pch=19)

lm.age =  lm(height ~ age, data=Loblolly)
age.seq = seq(0, 30, by=0.1)
lines(age.seq,  predict(lm.age,newdata=data.frame(age=age.seq)))

plot(Loblolly$age[order(Loblolly$age)],
     Loblolly$height[order(Loblolly$age)],
     xlab="Age (years)", ylab="Height (feet)",
     main = "Quadratic",
```

```
     col=adjustcolor("firebrick", 0.4), pch=19)

lm.age2 =  lm(height ~ poly(age, 2), data=Loblolly)
lines(age.seq,  predict(lm.age2,newdata=data.frame(age=age.seq)))
```



- What do you observe in these plots? → the quadratic fit appears to be slightly better than the linear one, because its curve intersects the points better.
- How well do the models fit the data?

Both appear to do a good job but ....

- We might also consider fitting higher order polynomials as well
  - we might want some additional functions to facilitate this

;

- `getXYpop` is a function that takes in a data frame with any number of variates and returns a data frame containing only the explanatory variate (x) and the response variate (y):

```
getXYpop <- function(xvarname, yvarname, pop) {
  popData <- pop[, c(xvarname, yvarname)]
  names(popData) <- c("x", "y")
  popData
}
```

;

- `getmuhat` is a function that uses least squares to fit polynomial response models (for arbitrary degrees in `x`):

```
getmuhat <- function(sampleXY, complexity) {
  formula <- paste0("y ~ ",
                    if (complexity==0) "1"
                    else {
                      if (complexity < 20 ) {
                        paste0("poly(x, ", complexity, ", raw = FALSE)")
                        ## due to Numerical overflow
                      } else {
                        ## if complexity >= 20 use a spline.
                        paste0("bs(x, ", complexity, ")")
                      }
                    }
  )

  fit <- lm(as.formula(formula), data = sampleXY)

  ## From this we construct the predictor function
  muhat <- function(x){
    if ("x" %in% names(x)) {
      ## x is a dataframe containing the variate named
      ## by xvarname
      newdata <- x
    } else
      ## x is a vector of values that needs to be a data.frame
    {newdata <- data.frame(x = x) }
    ## The suppress warnings prediction
    suppressWarnings({
      ypred = predict(fit, newdata = newdata, silent = TRUE)    })

    ypred
  }
  ## muhat is the function that we need to calculate values
  ## at any x, so we return this function from getmuhat
  muhat
}
```

;

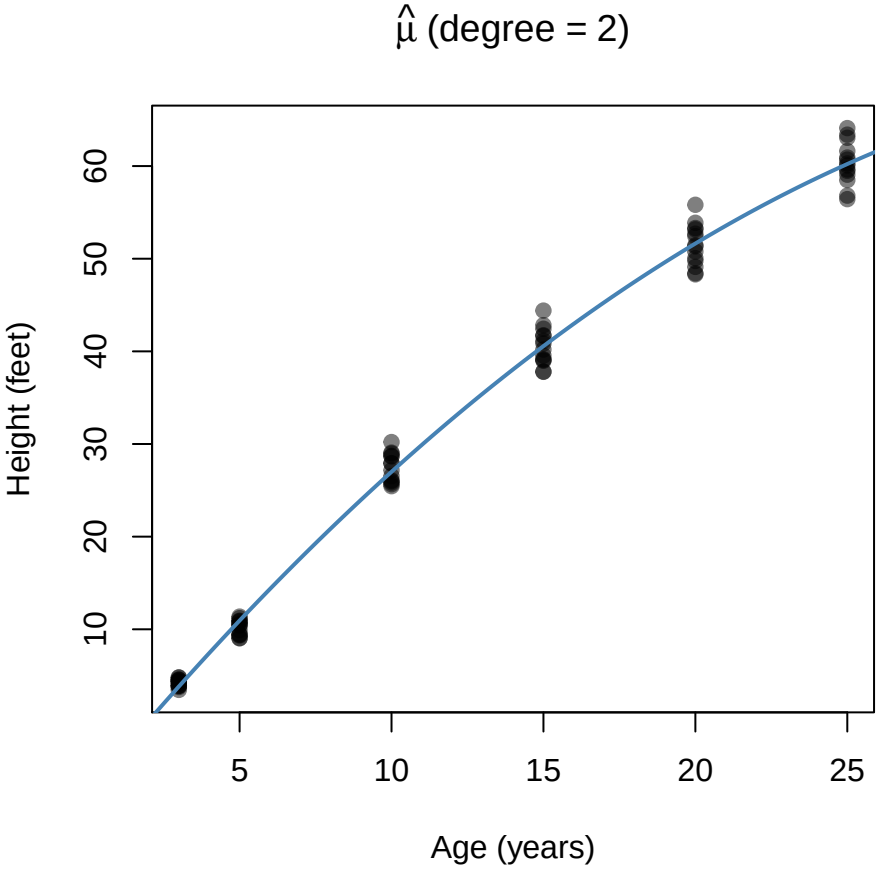
- plotLoblollyfit is a function that plots the Loblolly Pine data and overlays the fitted model muhat

```
plotLoblollyfit <- function(muhat, complexity=NULL) {

  if ( is.null(complexity) ) title = bquote(hat(mu) ~ "(Piecewise)")
  else title = bquote(hat(mu) ~ "(degree =" ~ .(complexity) * ")")

plot(Loblolly[,c("age", "height")],
     main= title,
     xlab = "Age (years)", ylab = "Height (feet)",
     pch=19, col= adjustcolor("black", 0.5))

xlim =  extendrange(Loblolly[, "age"])
curve(muhat, from = xlim[1], to = xlim[2],
      add = TRUE, col="steelblue", lwd=2, n=1000)

}
```

;

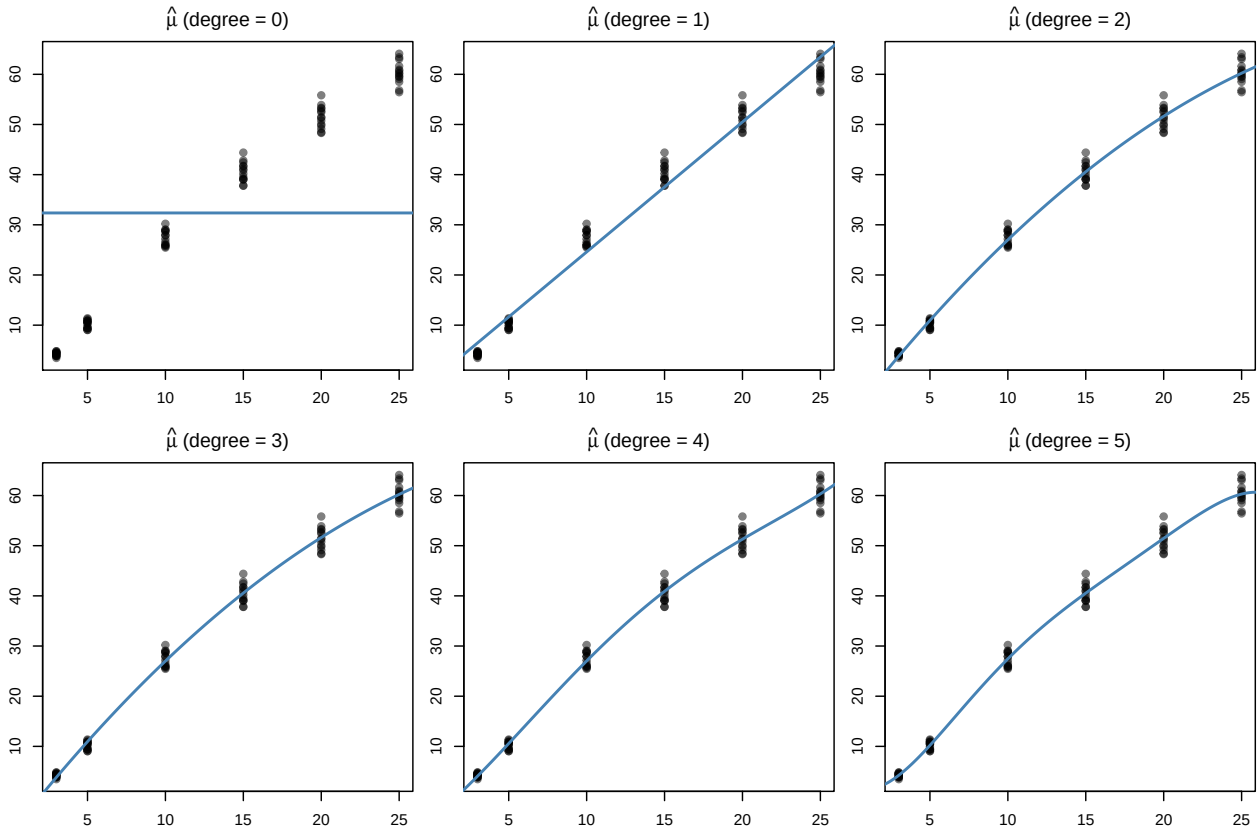
- Let's use these functions to fit and plot the quadratic model:

```
Lobpop <- getXYpop(xvarname="age",
                   yvarname="height",
                   pop=Loblolly)

muhat <- getmuhat(Lobpop, complexity=2)

plotLoblollyfit(muhat, complexity=2)
```



- Let's now explore fitting higher order polynomials

```
par(mfrow=c(2,3), mar=2.5*c(1,1,1,0.1))

dset = 0:5
muhats = lapply(dset, getmuhat, sampleXY=Lobpop  )

for (deg in dset) plotLoblollyfit(muhats[[deg+1]], complexity = deg)
```



- What do you observe in these plots?
- How well do the models fit the data?

All models with degree ≥ 2 fit the data well, but the higher degrees don't seem to do a much better job.

- We can use the APSE

$$\frac{1}{N}\sum_{u\in\mathcal{P}}(y_u-\widehat{\mu}(x_u))^2$$

to quantify the accuracy of these models

  - The function `apse` will perform this calculation for us.

```
apse <- function(y, x, predfun) {
  {mean((y - predfun(x))^2, na.rm = TRUE)}
}
```

- The APSE calculated over the entire population (for each model) is shown below:

```
dset = 0:5
Lob.apse = unlist( lapply(muhats, apse,
                          y= Lobpop$y, x =Lobpop$x ))


temp= rbind( 0:5, round(unlist(Lob.apse), 2) )
```

```
row.names(temp) = c("Polynomial Degree", "APSE")
dimnames(temp)[[2]] = rep("", 6)
temp
```

```
##
## Polynomial Degree   0.00 1.00 2.00 3.00 4.00 5.00
## APSE              422.31 8.48 2.79 2.79 2.72 2.64
```

- Which model is best?
- How much do we gain by making the model more complex (i.e., increasing degree of the polynomial)?

Strictly speaking Degree 5 has the smallest APSE, but it's not much different from the degree 2 APSE. So I would the quadratic model.

Fitting Piecewise Functions

- To model the relationship between `height` and `age` we could also fit a piecewise constant function
  - where different x intervals define different "pieces"
  - and the predictor function is based on the average y value in each "piece"

Such an approach is especially relevent here because x =`age` only has 6 unique values:

```
unique(Loblolly$age)
```

```
## [1]  3  5 10 15 20 25
```

- We will consider three appraoches to this:
  1. – piecewise constant with linear interpolation (`getmuFun`)
  2. – piecewise constant with each unique `age` defining the left endpoint of a "piece" (`getmuFun2`)
  3. – piecewise constant with each unique `age` in the middle of a "piece" (`getmuFun3`)

- Here we define `getmuFun`

```
getmuFun <- function(pop, xvarname, yvarname){
  pop   = na.omit(pop[, c(xvarname, yvarname)])

  # rule = 2 means return the nearest y-value when extrapolating, same as above.
  # ties = mean means that repeated x-values have their y-values averaged, as above.
  muFun = approxfun(pop[,xvarname], pop[,yvarname], rule = 2, ties = mean)
  return(muFun)
}
```

- Here we define `getmuFun2`

```
getmuFun2 <- function(pop, xvarname, yvarname){
  pop   = na.omit(pop[, c(xvarname, yvarname)])

  # rule = 2 means return the nearest y-value when extrapolating, same as above.
```

```
  # ties = mean means that repeated x-values have their y-values averaged, as above.
  muFun = approxfun(pop[,xvarname], pop[,yvarname], method = "constant", rule = 2, ties = mean, f=1/2)
  return(muFun)
}
```

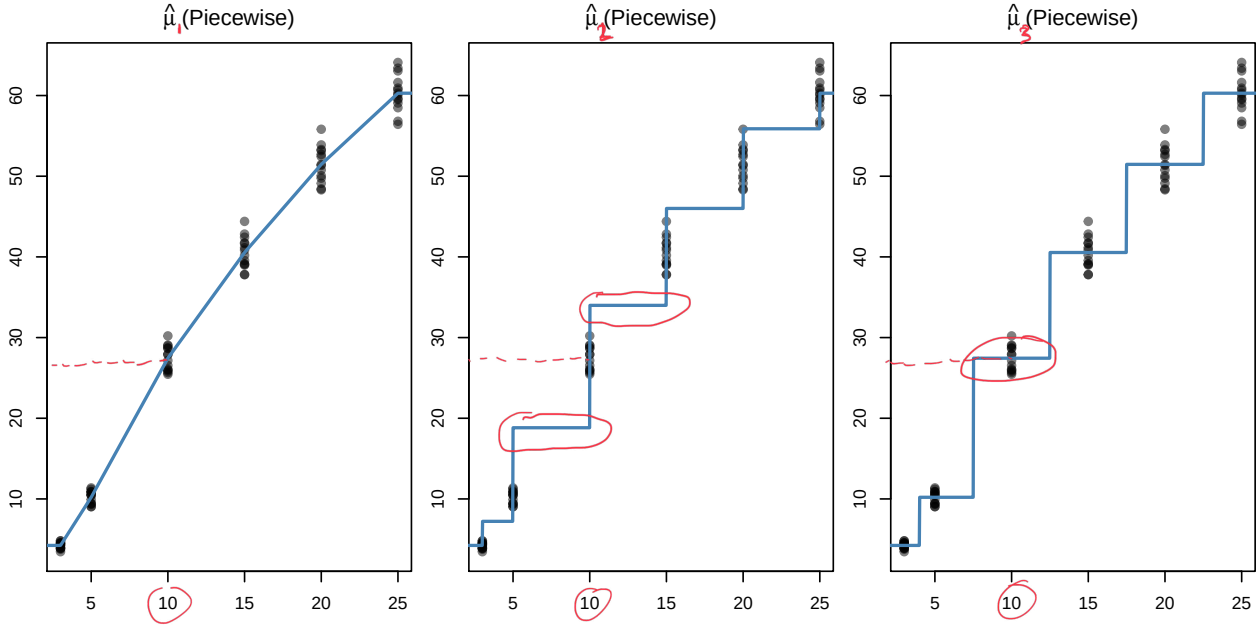
- Here we define getmuFun3

```
getmuFun3 <- function(pop, xvarname, yvarname){
  ## First remove NAs
  pop <- na.omit(pop[, c(xvarname, yvarname)])
  x <- pop[, xvarname]
  y <- pop[, yvarname]
  xks <- unique(x)
  muVals <- sapply(xks,
                   FUN = function(xk) {
                     mean(y[x==xk])
                   })
  ## Put the values in the order of xks
  ord <- order(xks)
  xks <- xks[ord]
  xkRange <-xks[c(1,length(xks))]
  minxk <- min(xkRange)
  maxxk <- max(xkRange)
  ## mu values
  muVals <- muVals[ord]
  muRange <- muVals[c(1, length(muVals))]
  muFun <- function(xVals){
    ## vector of predictions
    ## same size as xVals and NA in same locations
    predictions <- xVals
    ## Take care of NAs
    xValsLocs <- !is.na(xVals)
    ## Just predict non-NA xVals
    predictions[xValsLocs] <- sapply(xVals[xValsLocs],
      FUN = function(xVal) {
        if (xVal < minxk) {
          result <- muRange[1]
        } else if (xVal > maxxk) {
          result <- muRange[2]
        } else if ( any(xVal == xks) ) {
          result <- muVals[xks == xVal]
        } else {
          xlower <- max(c(minxk, xks[xks < xVal]))
          xhigher <- min(c(maxxk, xks[xks >= xVal]))
          mulower <- muVals[xks == xlower]
          muhigher <- muVals[xks == xhigher]

          midx = (xlower + xhigher)/2
          if (xVal <= midx) result <- mulower
          else result <- muhigher
        }
        result
      }
```

```
    )
    ## Now return the predictions (including NAs)
    predictions
  }
  muFun
}
```

- The three functions fitted to the population are visualized below

```
par(mfrow=c(1,3), mar=2.5*c(1,1,1,0.1))

mu.age = getmuFun(pop=Loblolly,
                  xvarname="age", yvarname="height")
plotLoblollyfit(mu.age)

mu.age2 = getmuFun2(pop=Loblolly,
                  xvarname="age", yvarname="height")
plotLoblollyfit(mu.age2)

mu.age3 = getmuFun3(pop=Loblolly,
                  xvarname="age", yvarname="height")
plotLoblollyfit(mu.age3)
```



- To better understand how these three functions differ, let's evaluate them just before, at and after $x = 10$:

```
x = c(9.9, 10, 10.1)
round(cbind(x, mu=mu.age(x), mu2=mu.age2(x), mu3=mu.age3(x)), 2)
```

```
##          x    mu   mu2   mu3
## [1,]  9.9 27.10 18.82 27.44
## [2,] 10.0 27.44 27.44 27.44
## [3,] 10.1 27.70 33.99 27.44
```

- Which function fits the population better?

For the observed data, these functions are equivalent. They differ with respect to x values not observed.

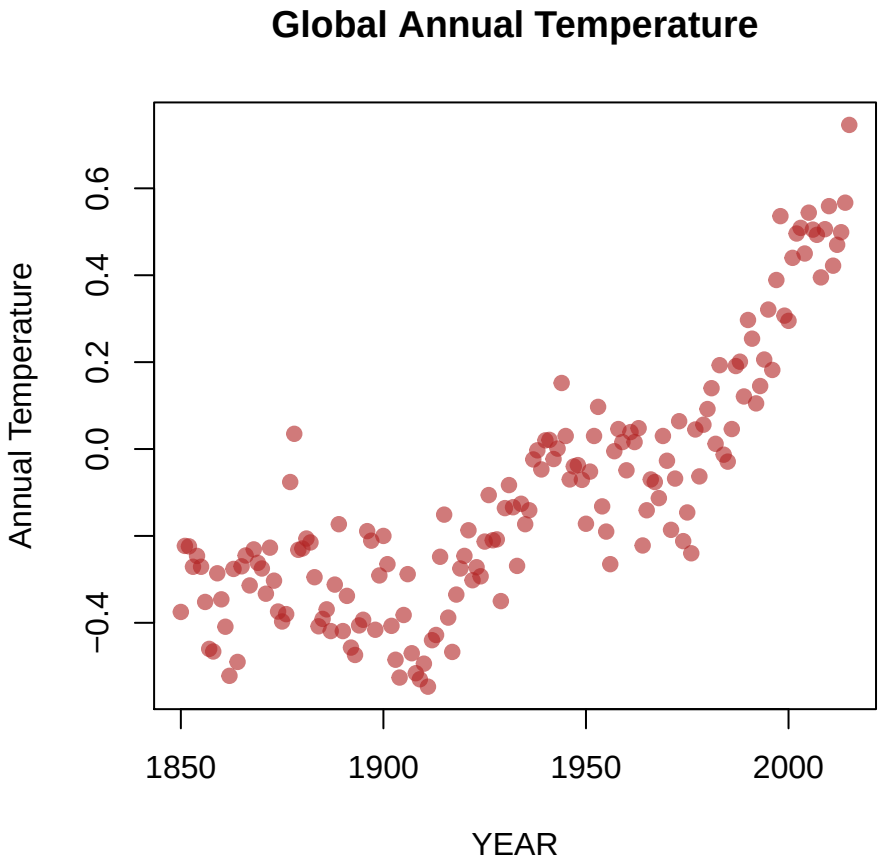
5.1.2 Example: Global Temperature Data

- This dataset concerns annual temperatures of the planet over time. The variables of interest are $y =$ `ANNUAL` temperature and $x =$ `YEAR`
  - Let's load and visualize it:

```
temperature <- read.csv("/Users/nstevens/Dropbox/Teaching/STAT_341/Lectures/Data/temperature.csv")
head(temperature)
```

```
##   YEAR    JAN    FEB    MAR    APR    MAY    JUN    JUL    AUG    SEP
## 1 1850 -0.702 -0.284 -0.732 -0.570 -0.325 -0.213 -0.128 -0.233 -0.444
## 2 1851 -0.303 -0.362 -0.485 -0.445 -0.302 -0.189 -0.215 -0.153 -0.108
## 3 1852 -0.308 -0.477 -0.505 -0.559 -0.209 -0.038 -0.016 -0.195 -0.125
## 4 1853 -0.177 -0.330 -0.318 -0.352 -0.268 -0.179 -0.059 -0.148 -0.409
## 5 1854 -0.360 -0.280 -0.284 -0.349 -0.230 -0.215 -0.228 -0.163 -0.115
## 6 1855 -0.176 -0.400 -0.303 -0.217 -0.336 -0.160 -0.268 -0.159 -0.339
##      OCT    NOV    DEC ANNUAL
## 1 -0.452 -0.190 -0.268 -0.375
## 2 -0.063 -0.030 -0.067 -0.223
## 3 -0.216 -0.187  0.083 -0.224
## 4 -0.359 -0.256 -0.444 -0.271
## 5 -0.188 -0.369 -0.232 -0.246
## 6 -0.211 -0.212 -0.510 -0.271
```

```
plot(temperature$YEAR, temperature$ANNUAL, xlab="YEAR", ylab="Annual Temperature",
     main="Global Annual Temperature", col=adjustcolor("firebrick", 0.6), pch=19 )
```



- In what follows we shall fit a variety of polynomials and piecewise functions to this data and use APSE to determine which model(s) seem appropriate

Fitting Polynomials

- To begin with, let's define `plotTemperaturefit`, a function that plots the Temperature data and overlays the fitted model `muhat`

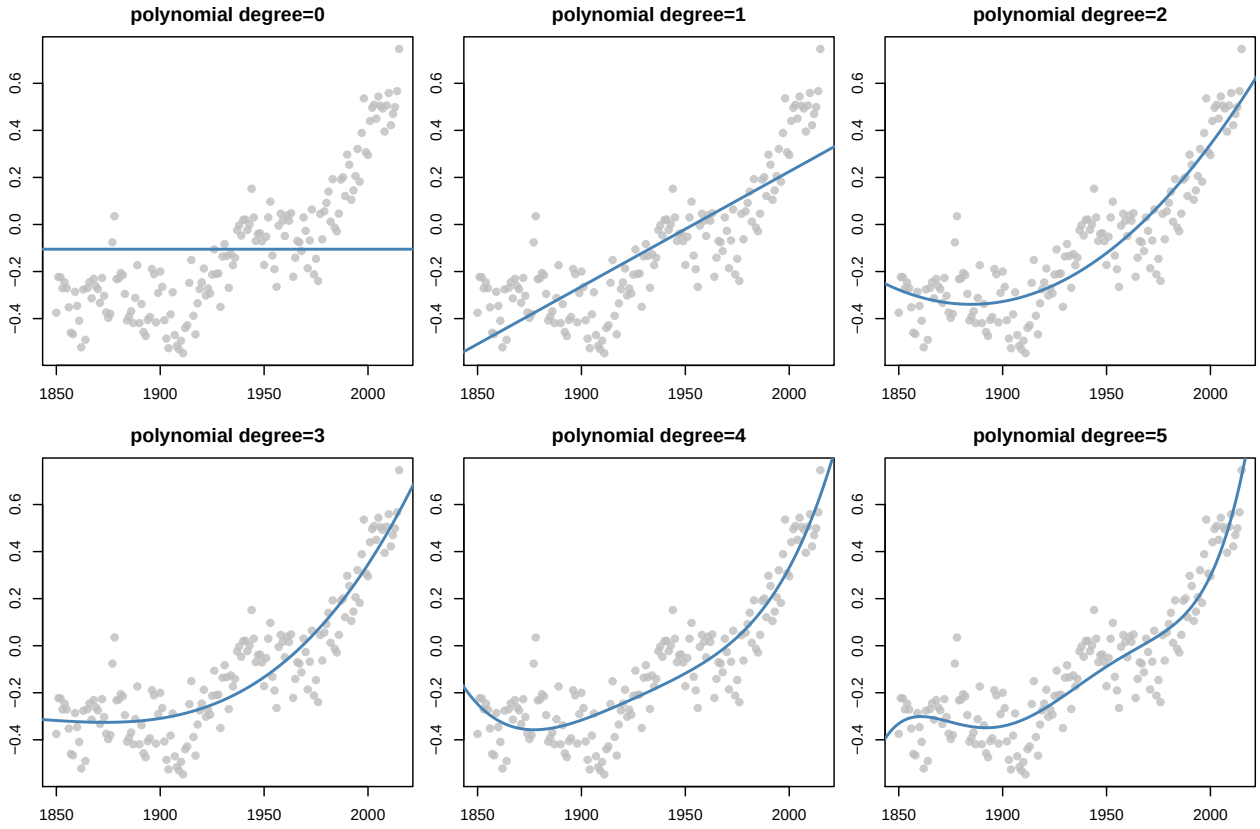
```
plotTemperaturefit <- function(muhat, complexity=NULL) {

  if ( is.null(complexity) ) title = ""
  else title = paste0("polynomial degree=", complexity,"")

plot(temperature[,c("YEAR", "ANNUAL")],
     main= title,
     xlab = "age", ylab = "height",
     pch=19, col= adjustcolor("Grey", 0.8))

xlim =  extendrange(temperature[, "YEAR"])
curve(muhat, from = xlim[1], to = xlim[2],
      add = TRUE, col="steelblue", lwd=2, n=1000)
```

```
}
```

Polynomial Regression (low degree)

- Consider fitting polynomials of degree `0:5`:

```
par(mfrow=c(2,3), mar=2.5*c(1,1,1,0.1))

temppop= getXYpop(xvarname="YEAR", yvarname="ANNUAL",pop=temperature)
dset = 0:5
muhats = lapply(dset, getmuhat, sampleXY=temppop)

for (i in 1:length(dset) ) plotTemperaturefit(muhats[[i]], dset[i])
```



- The APSE calculated over the entire population (for each model) is shown below:

```
dset = 0:5
temp.apse = unlist( lapply(muhats, apse,
                      y= temppop$y, x =temppop$x ))


tab = rbind( 0:5, round(unlist(temp.apse), 4) )
row.names(tab) = c("Polynomial Degree", "APSE")
dimnames(tab)[[2]] = rep("", 6)
tab
```

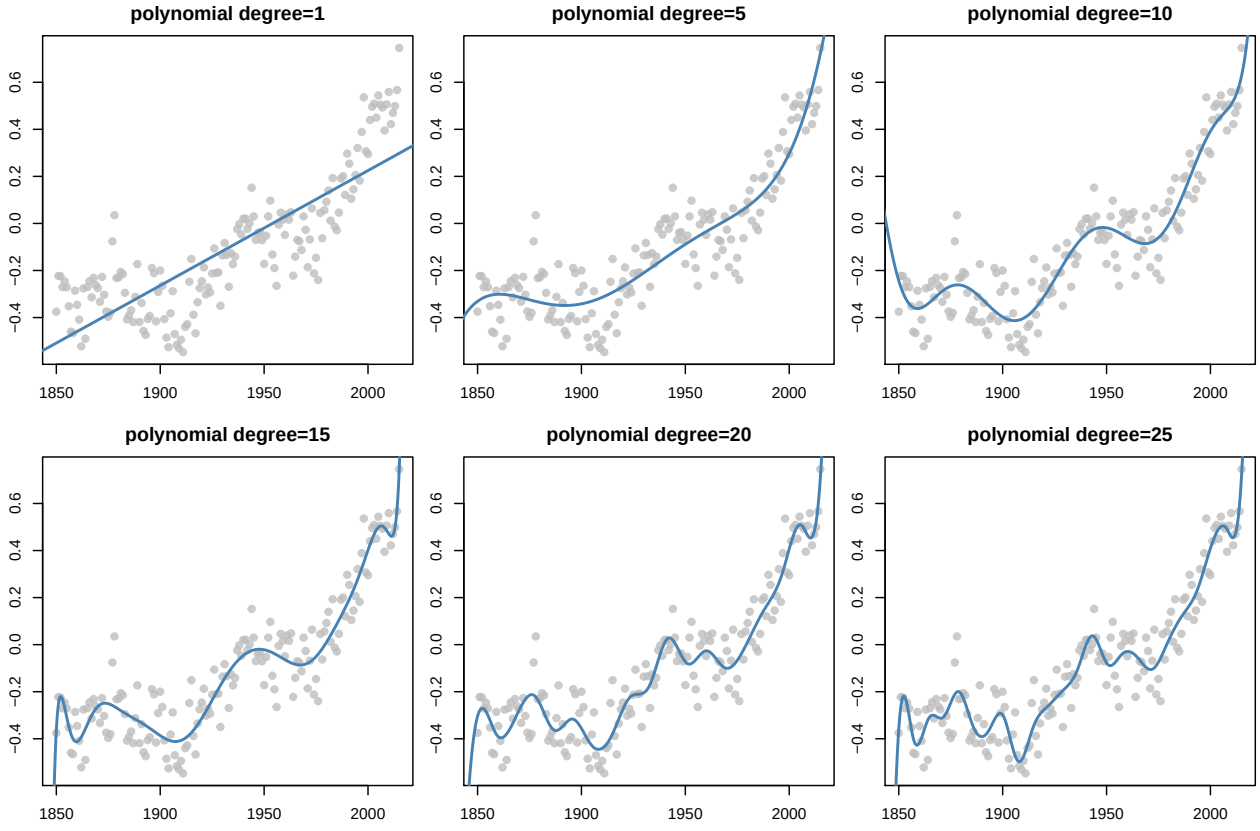
```
##
## Polynomial Degree 0.0000 1.0000 2.0000 3.0000 4.0000 5.0000
## APSE              0.0807 0.0258 0.0148 0.0145 0.0139 0.0132
```

- Which model is best?
- How much do we gain by making the model more complex (i.e., increasing degree of the polynomial)?

Polynomial Regression (high degree)

- Consider fitting polynomials of degrees 1, 5,10,15,20 and 25:

```
par(mfrow=c(2,3), mar=2.5*c(1,1,1,0.1))

dset = c(1, 5,10,15,20,25)

muhats = lapply(dset, getmuhat, sampleXY=temppop  )
for (i in 1:length(dset) ) plotTemperaturefit(muhats[[i]], dset[i])
```



- The APSE calculated over the entire population (for each model) is shown below:

```
temp.apse = unlist( lapply(muhats, apse,
                      y= temppop$y, x =temppop$x ))
```

```
dset = c(1, 5,10,15,20,25)

tab = rbind( dset, round(unlist(temp.apse), 4) )
row.names(tab) = c("Polynomial Degree", "APSE")
dimnames(tab)[[2]] = rep("", 6)
tab
```

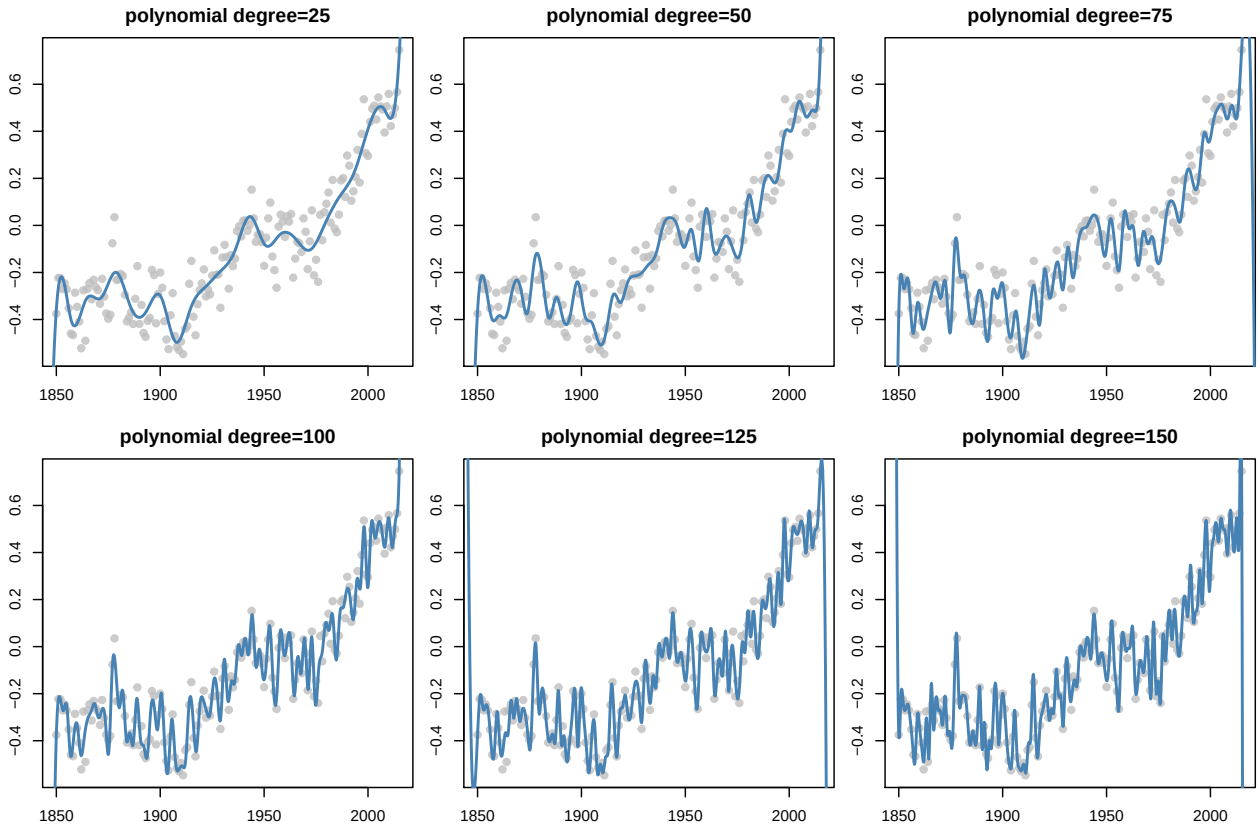
```
##
## Polynomial Degree 1.0000 5.0000 1e+01 15.0000 20.0000 25.0000
## APSE              0.0258 0.0132 9e-03  0.0084  0.0077  0.0071
```

- Which model is best?
- How much do we gain by making the model more complex (i.e., increasing degree of the polynomial)?

Polynomial Regression (higher degree)

- Consider fitting a polynomials of degree 20, 50, 75, 100, 125 and 165 to Global Annual Temperature.

```
par(mfrow=c(2,3), mar=2.5*c(1,1,1,0.1))

dset = c(25, 50, 75, 100, 125, 150)
muhats = lapply(dset, getmuhat, sampleXY=temppop  )

for (i in 1:length(dset)) plotTemperaturefit(muhats[[i]], dset[i])
```



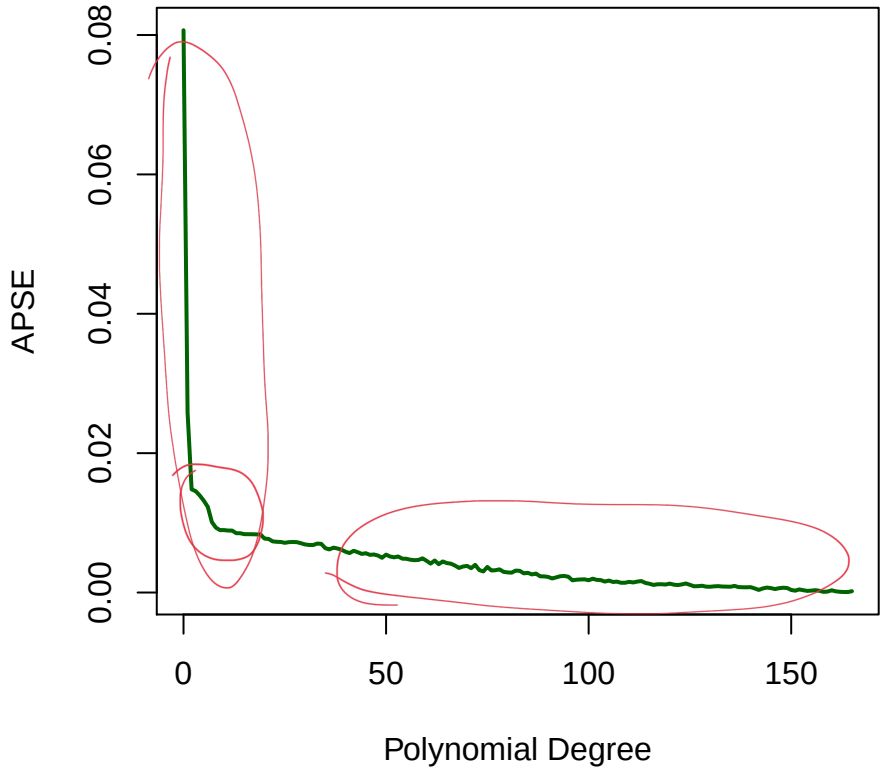
- The APSE calculated over the entire population (for each model) is shown below:

```
dset = c(25, 50, 75, 100, 125, 150)
temp.apse = unlist( lapply(muhats, apse,
                      y= temppop$y, x =temppop$x ))

tab = rbind( dset, round(unlist(temp.apse), 5) )
row.names(tab) = c("Polynomial Degree", "APSE")
dimnames(tab)[[2]] = rep("", 6)
tab
```

```
##
## Polynomial Degree 25.00000 50.00000 75.00000 1.00e+02 1.25e+02 1.5e+02
## APSE               0.00714  0.00544  0.00368 1.76e-03 1.11e-03 3.7e-04
```

- Which model is best?
- How much do we gain by making the model more complex (i.e., increasing degree of the polynomial)?
- What have we learned?

- So what level of *complexity* (i.e. what degree of polynomial) is appropriate?
  - To answer this question, let's visualize the improvement in APSE achieved by iteratively increasing the polynomial's degree:

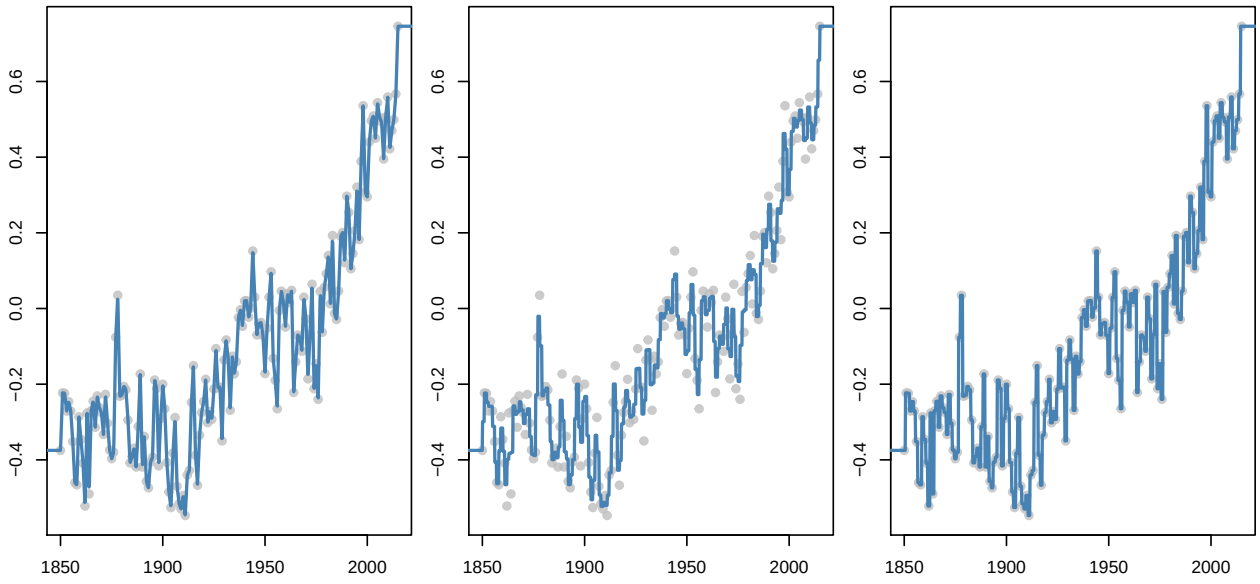


- What APSE value is acceptable?

Fitting Piecewise Functions

- To model this relationship we could also fit the piecewise constant functions:

```
par(mfrow=c(1,3), mar=2.5*c(1,1,1,0.1))

mu.year = getmuFun(pop=temperature,
                   xvarname="YEAR", yvarname="ANNUAL")
plotTemperaturefit(mu.year)

mu.year2 = getmuFun2(pop=temperature,
                   xvarname="YEAR", yvarname="ANNUAL")
plotTemperaturefit(mu.year2)

mu.year3 = getmuFun3(pop=temperature,
                   xvarname="YEAR", yvarname="ANNUAL")
plotTemperaturefit(mu.year3)
```



- Comparing these three plots, what do you observe? → they all behave similarly, and they are probably more complex than we need.
- Is this level of *complexity* appropriate?

5.1.3 Measuring Inaccuracy (Fairly)

- So far, we have been fitting models of various complexities to data, and comparing them on the basis of their **average prediction squared error** (APSE):

$$\frac{1}{N}\sum_{u\in\mathcal{P}}(y_u - \widehat{\mu}(\mathbf{x}_u))^2$$

- **HOWEVER:** In this approach we estimate the predictor function and measure its accuracy using the **exact same set** of observations.

- Our current inaccuracy measure can thus be written as

$$APSE(\mathcal{P},\widehat{\mu}_{\mathcal{P}}) = \frac{1}{N}\sum_{u\in\mathcal{P}}(y_u - \widehat{\mu}(\mathbf{x}_u))^2 = \frac{1}{N}\sum_{u\in\mathcal{P}}(y_u - \widehat{\mu}_{\mathcal{P}}(\mathbf{x}_u))^2$$

  where the notation $\widehat{\mu}_{\mathcal{P}}(\mathbf{x}_u)$ emphasizes the fact that the predictor function was determined from the entire population.

- This isn't the most honest way to estimate a predictor function's accuracy.
  - This method will underestimate the APSE for **predictions** at values of x not existing in the data (i.e., new/different values).
  - The dataset used to fit (**train**) the model is the same dataset we use to evaluate (**test**) the model.

– This (bad) approach leads to a problem known as **overfitting** which will be discussed in more detail below.

Ideally (to provide a fair evaluation of prediction accuracy) we would use different data to *train* vs. *test* the model and our measure of inaccuracy would reflect this.

– We should estimate the predictor function using a sample $\mathcal{S}$ (sometimes called the **training set**)

– AND measure the inaccuracy over the population $\mathcal{P}$, or over the units in the population not included in the sample: $\mathcal{T} = \mathcal{P} \setminus \mathcal{S}$ (sometimes called the **test set**)

$\mathcal{P} = \mathcal{S} \cup \mathcal{T}$ and $\mathcal{S} \cap \mathcal{T} = \emptyset$

- In this case we could write the APSE as

$$APSE(\mathcal{P}, \widehat{\mu}_{\mathcal{S}}) = \frac{1}{N} \sum_{u \in \mathcal{P}} (y_u - \widehat{\mu}_{\mathcal{S}}(\mathbf{x}_u))^2$$

– This notation emphasizes that the estimate of the predictor function $\widehat{\mu}$ is based on a sample $\mathcal{S}$

- Since $\mathcal{P} = \mathcal{S} \cup \mathcal{T}$ and $\mathcal{S} \cap \mathcal{T} = \emptyset$ (i.e., $\mathcal{T}$ is the complement set of $\mathcal{S}$ in $\mathcal{P}$) the APSE as defined in this way can be decomposed into a sum of two pieces:

$$\begin{aligned} APSE(\mathcal{P}, \widehat{\mu}_{\mathcal{S}}) &= \tfrac{1}{N} \textstyle\sum_{u \in \mathcal{P}} (y_u - \widehat{\mu}_{\mathcal{S}}(\mathbf{x}_u))^2 \\ &= \left(\tfrac{n}{N}\right) \tfrac{1}{n} \textstyle\sum_{u \in \mathcal{S}} (y_u - \widehat{\mu}_{\mathcal{S}}(\mathbf{x}_u))^2 \; + \; \left(\tfrac{N-n}{N}\right) \tfrac{1}{N-n} \textstyle\sum_{u \in \mathcal{T}} (y_u - \widehat{\mu}_{\mathcal{S}}(\mathbf{x}_u))^2 \\ &= \left(\tfrac{n}{N}\right) APSE(\mathcal{S}, \widehat{\mu}_{\mathcal{S}}) \; + \; \left(\tfrac{N-n}{N}\right) APSE(\mathcal{T}, \widehat{\mu}_{\mathcal{S}}) \end{aligned}$$

- Given that interest often lies in the quality of the predictions **outside of the sample**

– we might exclusively calculate average prediction squared error over $\mathcal{T}$

$$APSE(\mathcal{T}, \widehat{\mu}_{\mathcal{S}}) = Ave_{u \in \mathcal{T}} (y_u - \widehat{\mu}_{\mathcal{S}}(\mathbf{x}_u))^2.$$

But clearly if $n << N$, the value $APSE(\mathcal{T}, \widehat{\mu}_{\mathcal{S}})$ will not be that different from $APSE(\mathcal{P}, \widehat{\mu}_{\mathcal{S}})$.

Example: Global Temperature Data

- To begin with we will define some functions that will be useful throughout the example

- `getSampleComp` is a function that given a population (`pop`) of size N and a sample `size` n will return an N element vector containing n `TRUE` and $N - n$ `FALSE` values

– `TRUE` indicates the inclusion of a unit in the sample

– `FALSE` indicates the inclusion of a unit in the test set

```
getSampleComp <- function(pop, size, replace=FALSE) {
  N <- popSize(pop)
  samp <- rep(FALSE, N)
  samp[sample(1:N, size, replace = replace)] <- TRUE
  samp
}
```

- `getXYSample` is a function that extracts a given sample of units from a given population and returns a data frame containing just the explanatory variate (x) and the response variate (y)

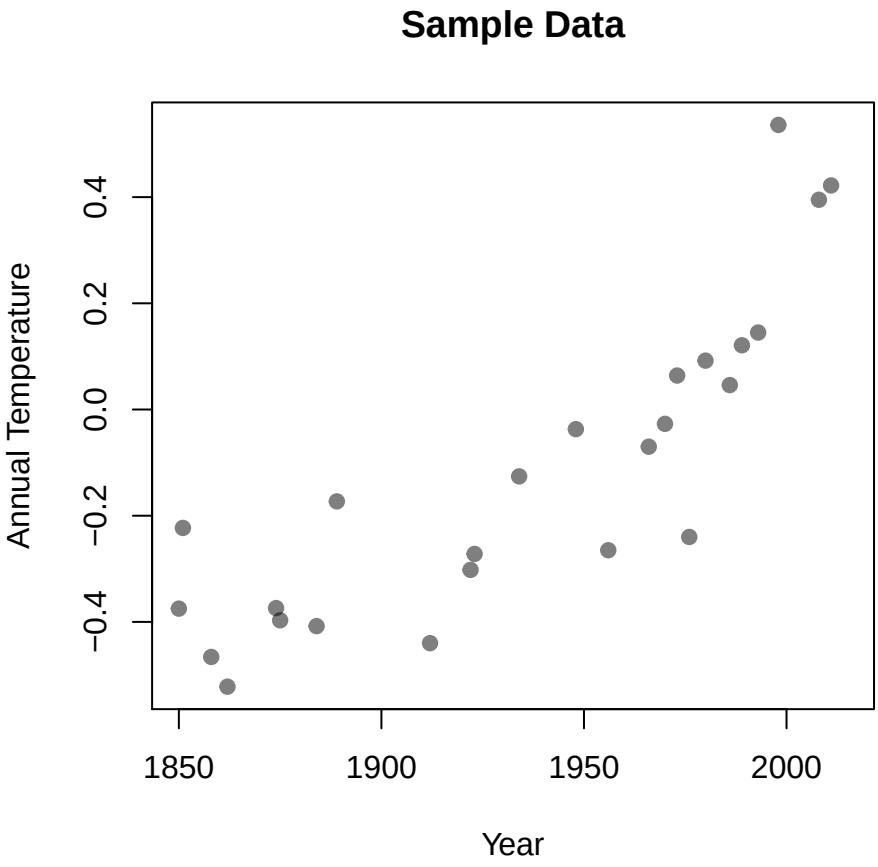
```
getXYSample <- function(xvarname, yvarname, samp, pop) {
  sampData <- pop[samp, c(xvarname, yvarname)]
  names(sampData) <- c("x", "y")
  sampData
}
```

- `popSize` and `sampSize` are functions used by both functions above to calculate the population and sample sizes, respectively.

```
popSize <- function(pop) {nrow(as.data.frame(pop))}
sampSize <- function(samp) {popSize(samp)}
```

- Suppose we have the following sample of $n = 25$ observations.

```
set.seed(341)
sample.temperature <- getSampleComp(temperature, 25)

sample.Data = getXYSample("YEAR", "ANNUAL",
                          samp=sample.temperature, pop=temperature)

plot(sample.Data,
     main= "Sample Data",
     xlab = "Year", ylab = "Annual Temperature",
     pch=19, col= adjustcolor("black", 0.5), xlim=range(temperature$YEAR))
```



- Let's try fitting some models to this sample data, and evaluate them on both the sample and the test data

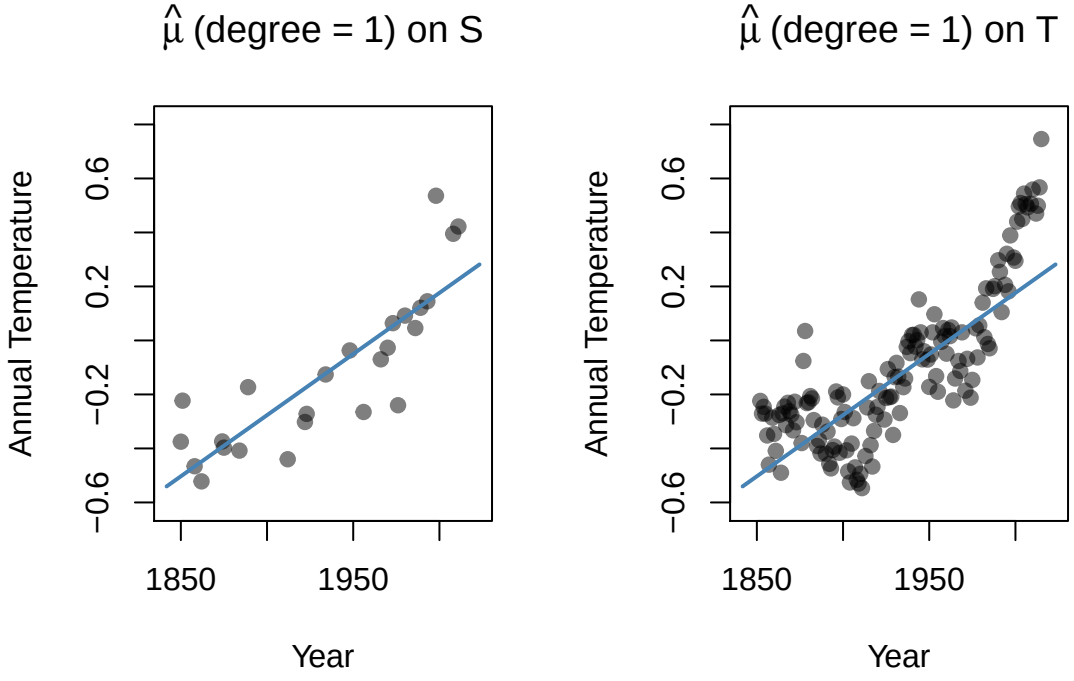
- To do this, the following function will be useful
  - `plotTemperatureSTfit` fits a model to $\mathcal{S}$ and plots both $\mathcal{S}$ and $\mathcal{T}$ with the fitted model overlaid on each

```
plotTemperatureSTfit <- function(muhat=NULL, Sam=NULL, complexity=NULL) {

  if ( is.null(complexity) ){
    titleS <- bquote(hat(mu) ~ "(piecewise) on S")
    titleT <- bquote(hat(mu) ~ "(piecewise) on T")
  }else{
    titleS <- bquote(hat(mu) ~ "(degree =" ~ .(complexity) * ") on S")
    titleT <- bquote(hat(mu) ~ "(degree =" ~ .(complexity) * ") on T")
  }
  title = c(paste0("muhat (degree=", complexity,") on S"),
                paste0("muhat (degree=", complexity,") on T"))

  xlim <- extendrange(temperature[, "YEAR"])
  ylim <- extendrange(temperature[, "ANNUAL"])

  plot(temperature[Sam,c("YEAR", "ANNUAL")],
     main= titleS,
```

```
    xlab = "Year", ylab = "Annual Temperature",
    pch=19, col= adjustcolor("black", 0.5),
    xlim=xlim, ylim=ylim)

  curve(muhat, from = xlim[1], to = xlim[2], add = TRUE,
      col="steelblue", lwd=2, n=1000)

  Tpop = !Sam

  plot(temperature[Tpop, c("YEAR", "ANNUAL")],
     main=titleT,
     xlab = "Year", ylab = "Annual Temperature",
     pch=19, col= adjustcolor("black", 0.5),
     xlim=xlim, ylim=ylim)

  curve(muhat, from = xlim[1], to = xlim[2], add = TRUE,
      col="steelblue", lwd=2, n=1000)

}
```

- Here is a linear predictor function
  - Recall: the line is fitted based on $\mathcal{S}$, and plotted on both $\mathcal{S}$ and $\mathcal{T}$

```
muhat <- getmuhat(sample.Data, 1)

par(mfrow=c(1,2))
plotTemperatureSTfit(muhat, sample.temperature, 1)
```



- The APSE calculated over the sample $\mathcal{S}$ is:

```
apse(y = temperature[sample.temperature,"ANNUAL"],
     x = temperature[sample.temperature,"YEAR"], predfun = muhat)
```

```
## [1] 0.02308721
```

- The APSE calculated over the complement set $\mathcal{T} = \mathcal{P} \setminus \mathcal{S}$ is:

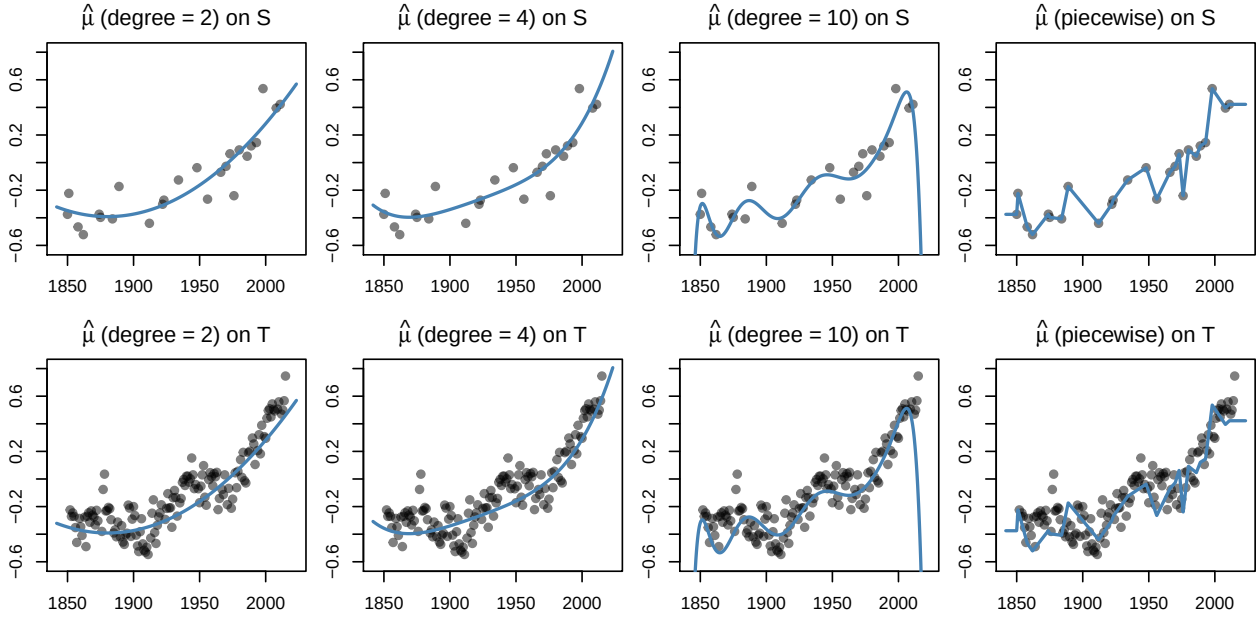
```
Tpop <- !sample.temperature
apse(y = temperature[Tpop,"ANNUAL"], x = temperature[Tpop,"YEAR"], predfun = muhat)
```

```
## [1] 0.0273872
```

- Note that the APSE over $\mathcal{S}$ is *lower* than the APSE over $\mathcal{T}$ (as expected)

- Let's also consider some higher order polynomials

```
dset = c(2,4,10)
muhats    = lapply(dset, getmuhat, sampleXY=sample.Data  )
muhatFun  = getmuFun(sample.Data, "x", "y" )


par(mfcol=c(2,4), mar=2.5*c(1,1,1,0.1))
for (i in 1:length(dset)) plotTemperatureSTfit(muhats[[i]], sample.temperature, dset[i])
plotTemperatureSTfit(muhatFun, sample.temperature)
```



- The APSE calculated over $\mathcal{S}$ and $\mathcal{T}$ for each model (and a function to do so) is shown below:

```
apseST <- function(y, x, sam, predfun) {
  apseS = apse(y[sam], x[sam], predfun)
```

```
  Tpop = !sam
  apseT = apse(y[Tpop], x[Tpop], predfun)

  c(apseS, apseT)
}
```

```
muhats[[4]] = muhatFun

temp.apse = sapply(muhats, apseST,
              y= temppop$y, x =temppop$x, sam=sample.temperature )

rownames(temp.apse) = c("APSE on S", "APSE on T")
colnames(temp.apse) = c(paste("deg=", dset), "piecewise linear ")
round(temp.apse,3)
```

```
##           deg= 2 deg= 4 deg= 10 piecewise linear
## APSE on S  0.013  0.012   0.008             0.000
## APSE on T  0.018  0.017   0.023             0.019
```

Complexity ↑, APSE ↓

Complexity ↑, APSE ↑

- Note the difference in APSEs between $\mathcal{S}$ and $\mathcal{T}$.

Aside: Extrapolation Beyond the Sample

- In the previous plots the `getmuhat` function didn't work very sensibly for x values smaller or larger than what was observed in the sample

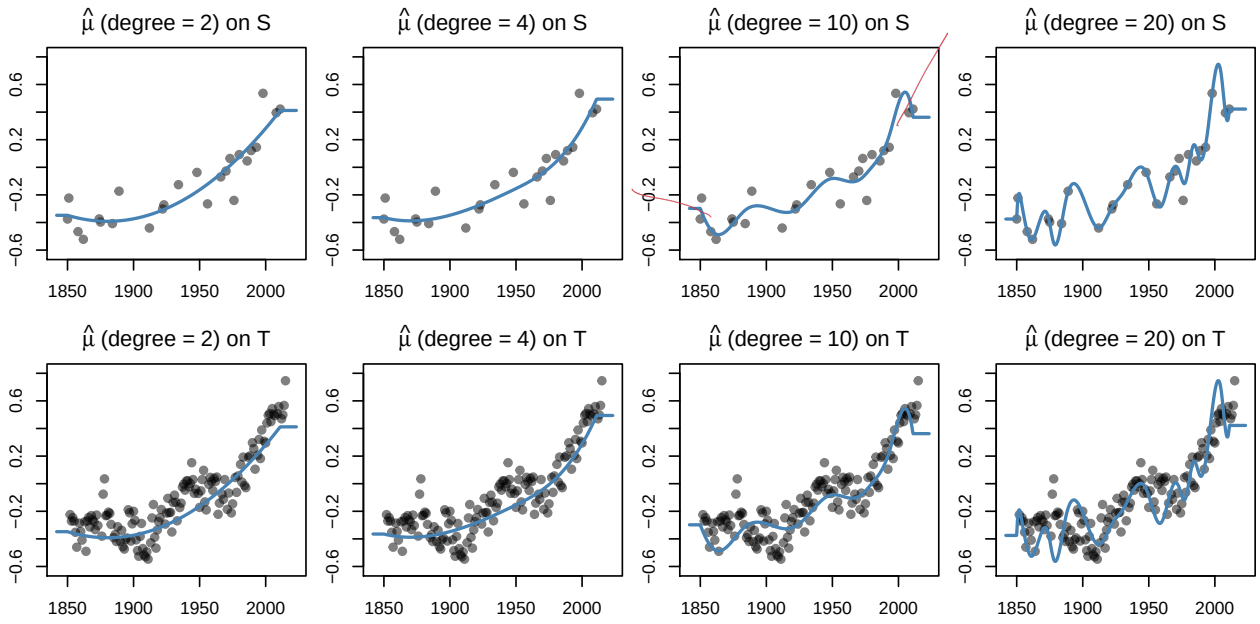
- We can modify the `getmuhat` function so that extrapolation beyond the minimum and maximum of the sample are set to constants.

```
getmuhat.ext <- function(sampleXY, complexity = 1) {
  formula <- paste0("y ~ ",
                    if (complexity==0) "1"
                    else {
                      if (complexity < 3 ) {
                        paste0("poly(x, ", complexity, ", raw = FALSE)")
                        ## due to Numerical overflow
                      } else {
                        ## if complexity >= 20 use a spline.
                        paste0("bs(x, ", complexity, ")")
                      }
                    }
  )

  fit <- lm(as.formula(formula), data = sampleXY)
  tx = sampleXY$x
  ty = fit$fitted.values

  range.X = range(tx)
  val.rY  = c( mean(ty[tx == range.X[1]]),
```

```
                mean(ty[tx == range.X[2]]) )

  ## From this we construct the predictor function
  muhat <- function(x){
    if ("x" %in% names(x)) {
      ## x is a dataframe containing the variate named
      ## by xvarname
      newdata <- x
    } else
      ## x is a vector of values that needs to be a data.frame
    { newdata <- data.frame(x = x) }
    ## The prediction
    ##
    suppressWarnings({
      ypred = predict(fit, newdata = newdata, silent = TRUE)    })
    #val = predict(fit, newdata = newdata)
    ypred[newdata$x < range.X[1]] = val.rY[1]
    ypred[newdata$x > range.X[2]] = val.rY[2]
    ypred
  }
  ## muhat is the function that we need to calculate values
  ## at any x, so we return this function from getmuhat
  muhat
}
```

- Let's recreate the plots from above with this new version of `muhat`

```
dset = c(2, 4, 10, 20)
muhats    = lapply(dset, getmuhat.ext, sampleXY=sample.Data  )
muhatFun  = getmuFun(sample.Data, "x", "y" )

par(mfcol=c(2,4), mar=2.5*c(1,1,1,0.1))
for (i in 1:length(dset)) plotTemperatureSTfit(muhats[[i]], sample.temperature, dset[i])
```

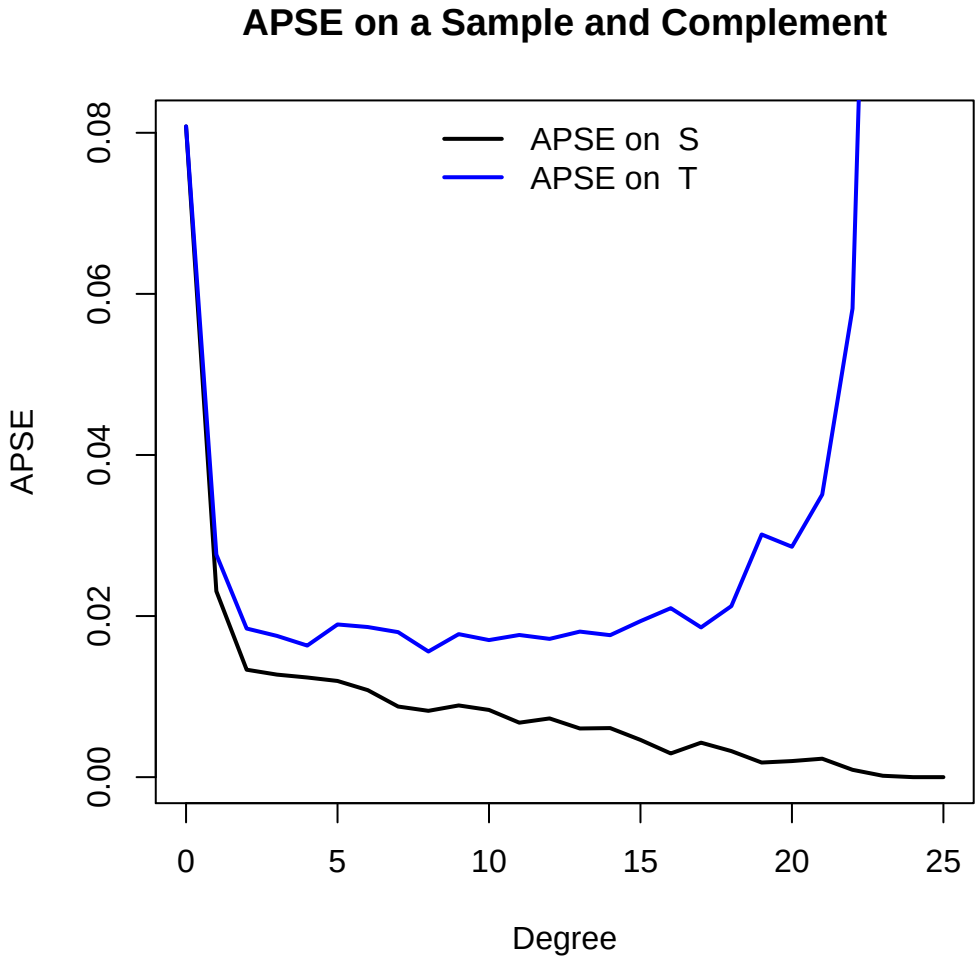


- What else might we do instead of using constant values for extrapolation?

- Let us look at a larger collection of polynomials to pick an appropriate model.

- Comparing the APSE on the sample and complement set we have:

```
dset = 0:25
muhats = lapply(dset, getmuhat.ext, sampleXY=sample.Data  )

temp.apse = sapply(muhats, apseST,
              y= temppop$y, x =temppop$x, sam=sample.temperature )

rownames(temp.apse) = c("APSE on S", "APSE on T")
colnames(temp.apse) = paste("deg=", dset)


plot( dset, temp.apse[1,], type='l', col=1, lwd=2,
      xlab="Degree", ylab="APSE",
      main="APSE on a Sample and Complement" )
lines(dset, temp.apse[2,], type='l', col=4, lwd=2 )
legend( "top", legend= paste("APSE on ", c("S","T")),
        col=c(1,4), lwd=c(2,2), bty='n')
```



- At around degree equal to 4, increasing complexity does not improve prediction accuracy.
  - Increasing complexity will continue to improve predictions on the sample but not on the rest of the population.
  - This effect is what we previously referred to as **overfitting**:
    - the predictor function has been too closely tailored to the peculiarities of the sample.
    - the complexity of the model has been increased so far that it has comprimised the *out-of-sample* prediction performance

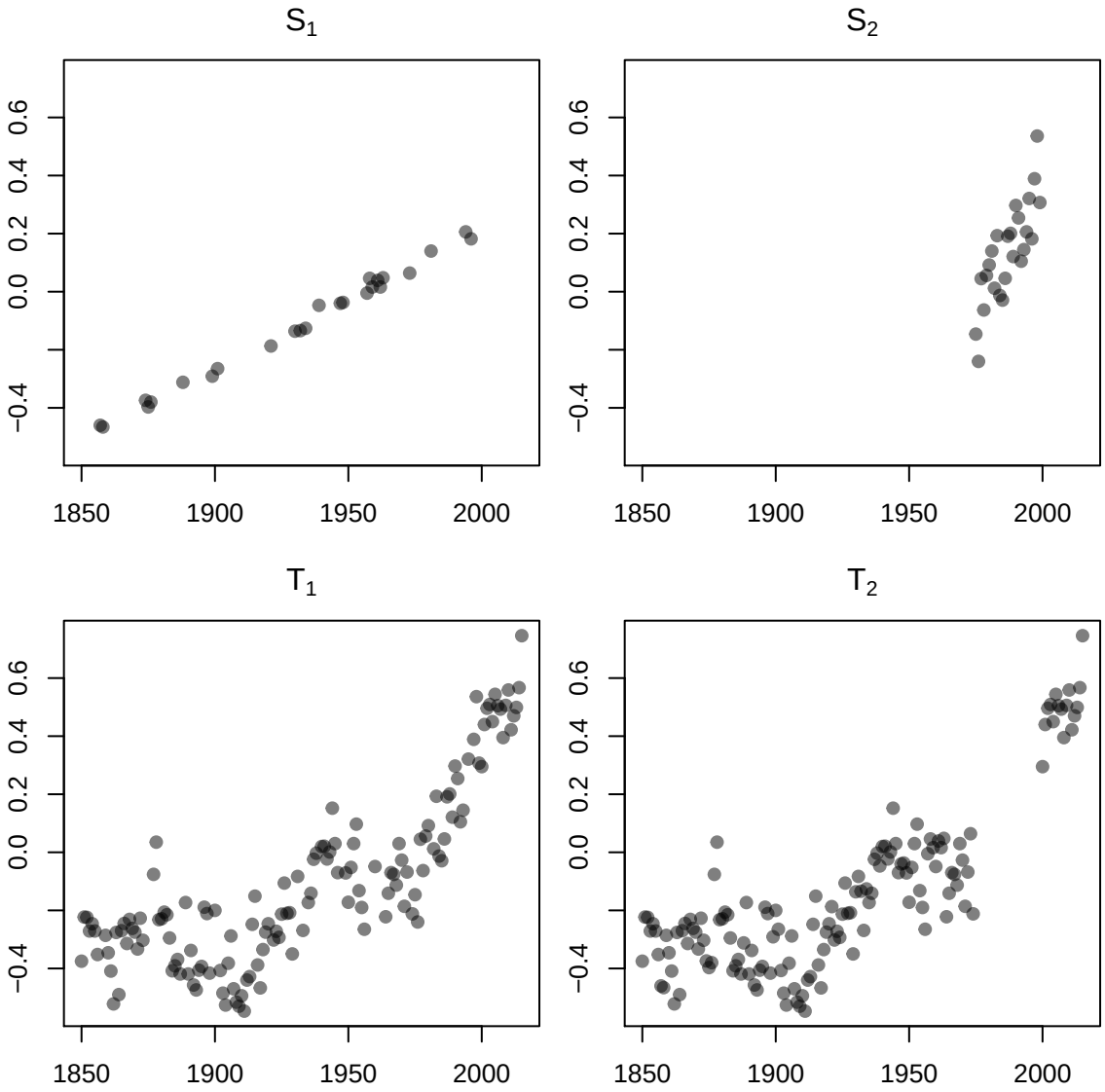
5.1.4 The Importance of a Good Sample

- Since $\widehat{\mu}_{\mathcal{S}}(\mathbf{x})$ is based on a single sample $\mathcal{S}$ the quality of the predictor function depends crucially on the quality of the sample
  - if the sample is not a good/fair representation of the population, then any predictor function is bound to perform poorly

- It is important, then, to recognize that the performance (i.e., APSE) associated with $\widehat{\mu}_{\mathcal{S}}(\mathbf{x})$ could vary quite a lot from one sample to another.

- In practice we tend to assume our sample is a good representation of the population
  - But in case that's not true it's important to choose a predictor function that performs well no matter which sample was used to estimate it.
  - Simpler is often better.

- Here are two partitions of the population $\mathcal{P}$:



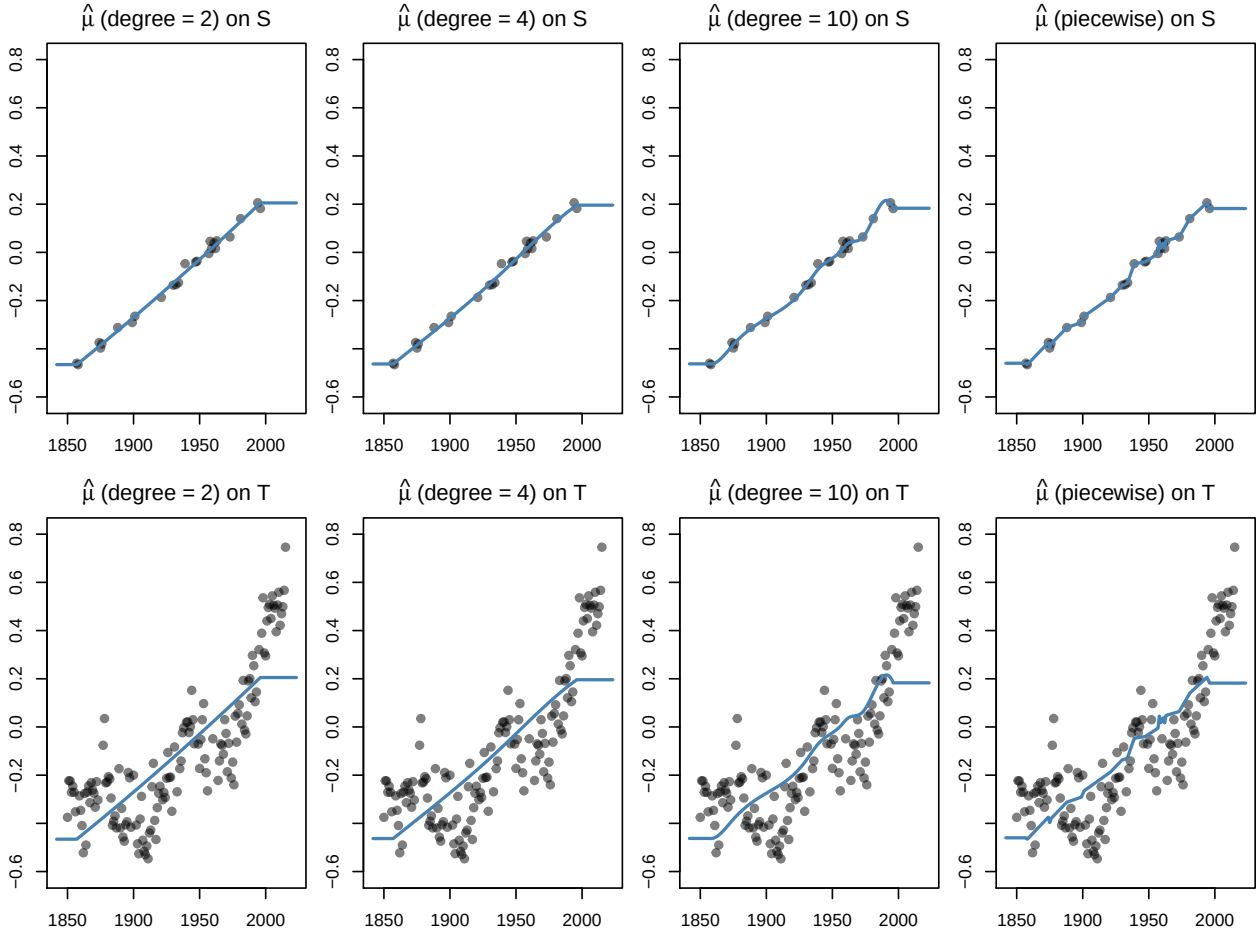
- What do you observe?
- Are both samples *good* representations of the population?

No. S_1 appears to be better than S_2.

Example: Global Temperature Data (Sample 1)

- Here are some polynomials fit to Sample 1.

```
sam1.Data = getXYSample("YEAR", "ANNUAL", samp=sam1, pop=temperature)

dset = c(2,4,10)
muhats    = lapply(dset, getmuhat.ext, sampleXY=sam1.Data  )
muhatFun  = getmuFun(sam1.Data, "x", "y" )


par(mfcol=c(2,4), mar=2.5*c(1,1,1,0.1))
for (i in 1:length(dset)) plotTemperatureSTfit(muhats[[i]], sam1, dset[i])
plotTemperatureSTfit(muhatFun, sam1)
```



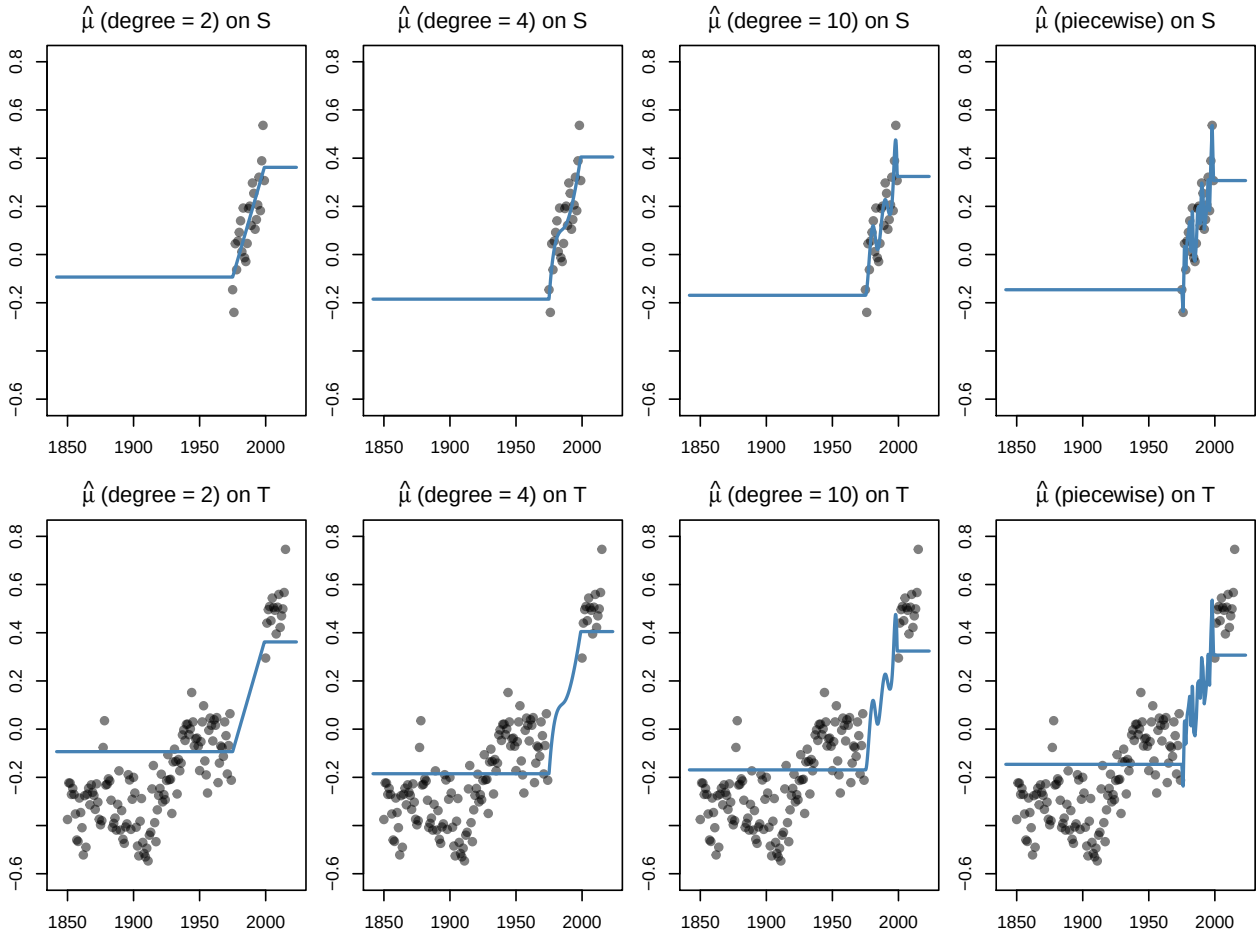
- Which predictor performs better on the test set?

  What would you have chosen had you not seen the test set?

Example: Global Temperature Data (Sample 2)

- Here are some polynomials fit to Sample 2.

```
sam2.Data = getXYSample("YEAR", "ANNUAL", samp=sam2, pop=temperature)

dset = c(2,4,10)
muhats    = lapply(dset, getmuhat.ext, sampleXY=sam2.Data  )
muhatFun  = getmuFun(sam2.Data, "x", "y" )


par(mfcol=c(2,4), mar=2.5*c(1,1,1,0.1))
for (i in 1:length(dset)) plotTemperatureSTfit(muhats[[i]], sam2, dset[i])
plotTemperatureSTfit(muhatFun, sam2)
```



- Which predictor performs better on the test set?
  - What would you have chosen had you not seen the test set?