Tesla Battery Survey

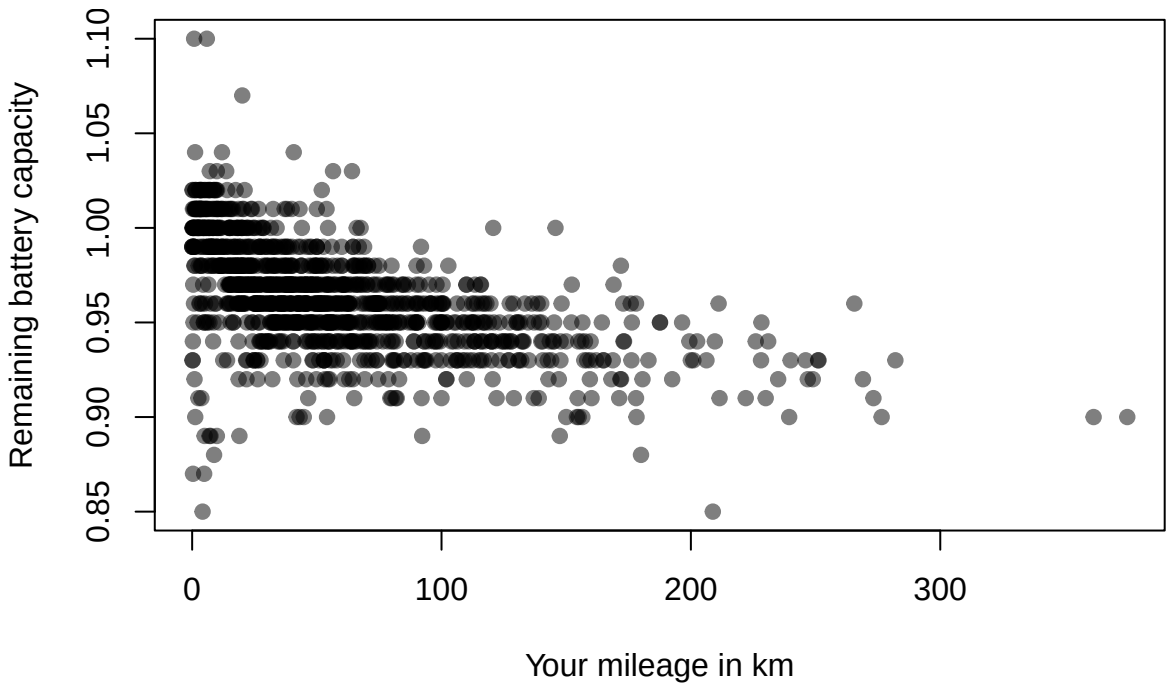
Aiden Ramgoolam

2023-05-21

Battery degradation is one of the biggest concerns for electric car owners and potential buyers, but data from Tesla battery packs have been very reassuring so far. A group of Tesla owners on the Dutch-Belgium Tesla Forum are gathering data from over 350 Tesla vehicles across the world and frequently updating it in a public Google file.
Goal:
We will use the Tesla battery survey to explore the nature of the predictive accuracy of various polynomials.
The file tesla_battery_Survey.csv contains battery and mileages information on variety on Tesla.
We will try to predict Remaining.battery.capacity using Your.mileage.km.

```
filename <- "tesla_battery_Survey.csv"
tesla <- read.csv(file.path(getwd(), filename), header = TRUE)
#summary(tesla)
#data(tesla)
#head(tesla)
tesla$Your.mileage.km = tesla$Your.mileage.km/1000
```

```
plot(tesla$Your.mileage.km, tesla$Remaining.battery.capacity,
     xlab="Your mileage in km", ylab="Remaining battery capacity",
     pch=19, col=adjustcolor("black", 0.5))
```




```
set.seed(341)
popSize  <- function(pop) {nrow(as.data.frame(pop))}
```

```r
sampSize <- function(samp) {popSize(samp)}
```

```r
# This function returns a boolean (TRUE/FALSE) vector representing the
#   inclusion indicators. (This way the complement is also recorded.)
getSampleComp = function(pop, size, replace = FALSE) {
  N    = popSize(pop)
  samp = rep(FALSE, N)
  samp[sample(1:N, size, replace = replace)] = TRUE
  return(samp)
}

# This function returns a data frame with only two variates, relabelled as
#   x and y explicitly.
getXYSample = function(xvarname, yvarname, samp, pop) {
  sampData        = pop[samp, c(xvarname, yvarname)]
  names(sampData) = c("x", "y")
  return(sampData)
}
```

```r
# Load the 'splines' package
library(splines)

getmuhat = function(sampleXY, complexity = 1) {
  if (complexity == 0) {
    fit <- lm(y ~ 1, data = sampleXY)  # Fit only intercept
  } else {
    # Fit a natural spline with 'complexity' degrees of freedom
    fit <- lm(y ~ ns(x, df = complexity), data = sampleXY)
  }

  xmin <- min(sampleXY$x, na.rm = TRUE)
  xmax <- max(sampleXY$x, na.rm = TRUE)

  # Create the predictor function using the fitted model
  muhat <- function(x) {
    x <- as.data.frame(x)            # Convert to data frame, needed by predict
    x$x <- pmax(x$x, xmin)           # *Replace values below xmin with xmin
    x$x <- pmin(x$x, xmax)           # *Replace values above xmax with xmax
    pred <- predict(fit, newdata = x) # Get yhat values from fitted lm model
    return(pred)
  }

  return(muhat)
}
```

```r
getmubar <- function(muhats) {
  function(x) {
    Ans <- sapply(muhats, function(muhat) {muhat(x)})
    apply(Ans, MARGIN = 1, FUN = mean) # Equivalently, rowMeans(A)
  }
}
```

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

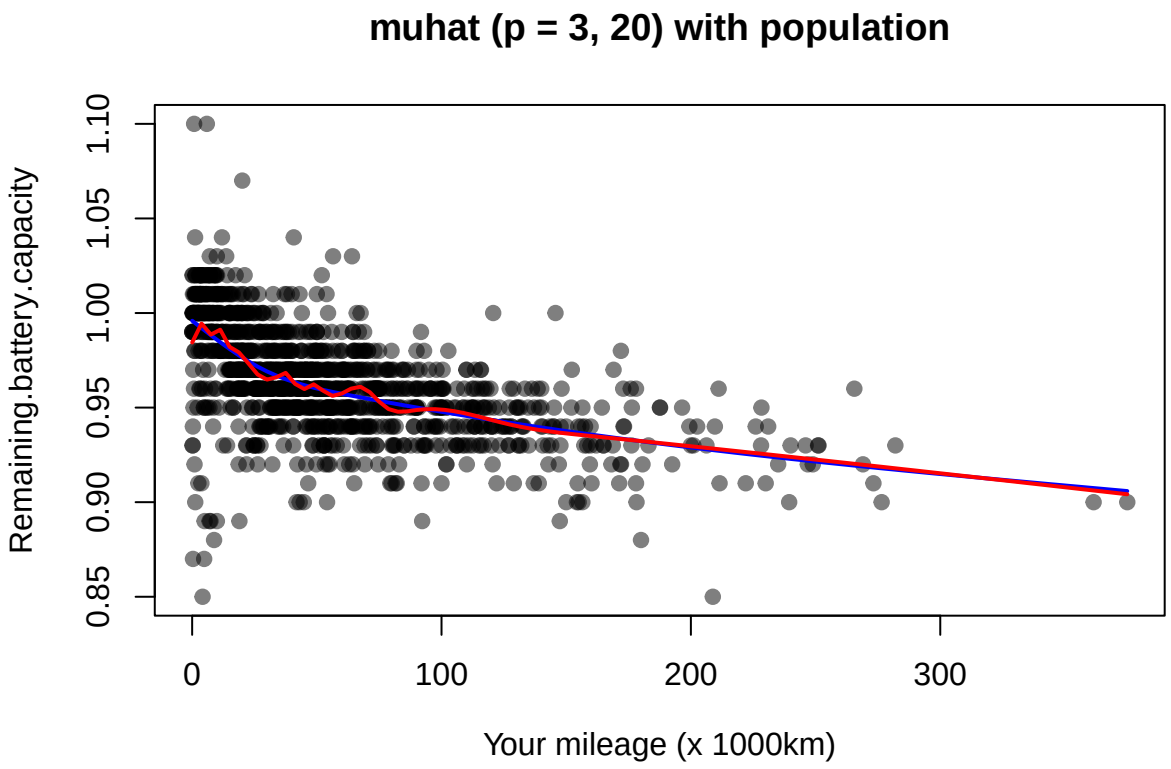
```r
ave_mu_mu_sq <- function(predfun1, predfun2, x, na.rm = TRUE){
  mean((predfun1(x) - predfun2(x))^2, na.rm = na.rm)
}

var_mutilde <- function(Ssamples, Tsamples, complexity){
  # Evaluate predictor function on each sample S in Ssamples
  muhats = lapply(Ssamples, getmuhat, complexity = complexity)

  # Get the average of these, name it mubar
  mubar  = getmubar(muhats)

  # Average over all samples S
  N_S    = length(Ssamples)
  mean(sapply(1:N_S, function(j) {
    # Use muhat function from sample S_j in Ssamples
    muhat = muhats[[j]]
    ## Average over (x_i, y_i) of sample T_j the squares (y_i - muhat(x_i))^2
    T_j   = Tsamples[[j]]
    return(ave_mu_mu_sq(muhat, mubar, T_j$x))
  }))
}
```

```r
bias2_mutilde <- function(Ssamples, Tsamples, mu, complexity){
  # Evaluate predictor function on each sample S in Ssamples
  muhats = lapply(Ssamples, getmuhat, complexity = complexity)

  # Get the average of these, name it mubar
  mubar  = getmubar(muhats)

  # Average over all samples S
  N_S    = length(Ssamples)
  mean(sapply(1:N_S, function(j) {
    ## Average over (x_i, y_i) of sample T_j the squares (y_i - muhat(x_i))^2
    T_j = Tsamples[[j]]
    return(ave_mu_mu_sq(mubar, mu, T_j$x))
  }))
}
```

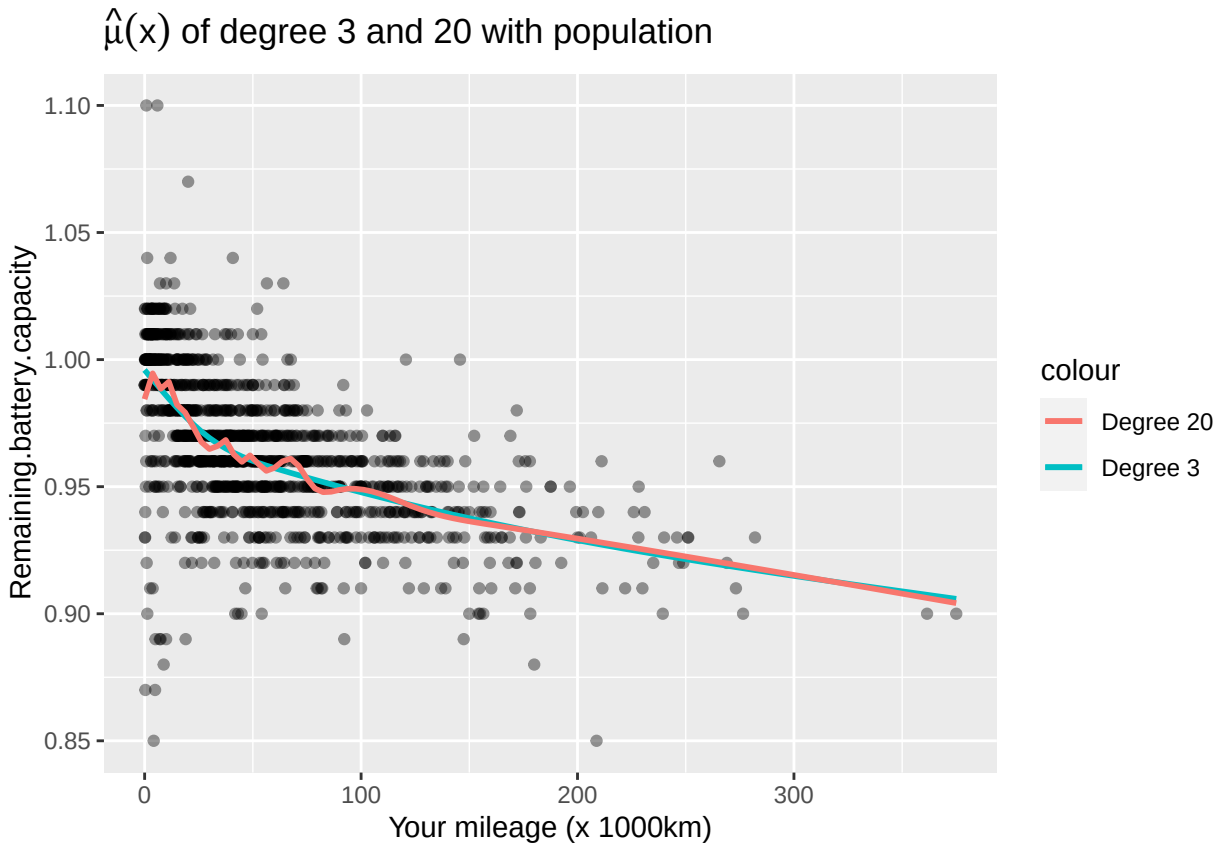
Generate the scatter plot of the data (with shading) and overlay the fitted polynomials with degrees 3 and 20 to the data.

```r
xvarname <- "Your.mileage.km"
yvarname <- "Remaining.battery.capacity"

# Define the population
pop <- data.frame(Your.mileage.km = tesla$Your.mileage.km, Remaining.battery.capacity = tesla$Remaining

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

po= getXYpop(xvarname, yvarname,tesla)
```

```r
muhat3   = getmuhat(po, 3)
muhat20  = getmuhat(po, 20)

plot(pop,
     main = paste0("muhat (p = 3, 20) with population"),
       xlab = expression(paste("Your mileage", " (x 1000km)")),
     ylab = yvarname,
     pch  = 19,
     col  = adjustcolor("black", 0.5))
curve(muhat3,  add = TRUE, col = "blue", lwd = 2)
curve(muhat20, add = TRUE, col = "red", lwd = 2)

library(ggplot2)
```




```r
ggplot(pop, aes(x = Your.mileage.km, y = Remaining.battery.capacity)) +
  expand_limits(x = c(-0.05, 1.05)) +
  geom_point(colour = 'black', alpha = 0.4) +
  stat_function(aes(color = "Degree 3"), fun = muhat3, lwd = 1) +
  stat_function(aes(color = "Degree 20"), fun = muhat20, lwd = 1) +
  ggtitle(expression(paste(hat(mu)(x), " of degree 3 and 20 with population"))) +
  xlab("Your mileage (x 1000km)")
```



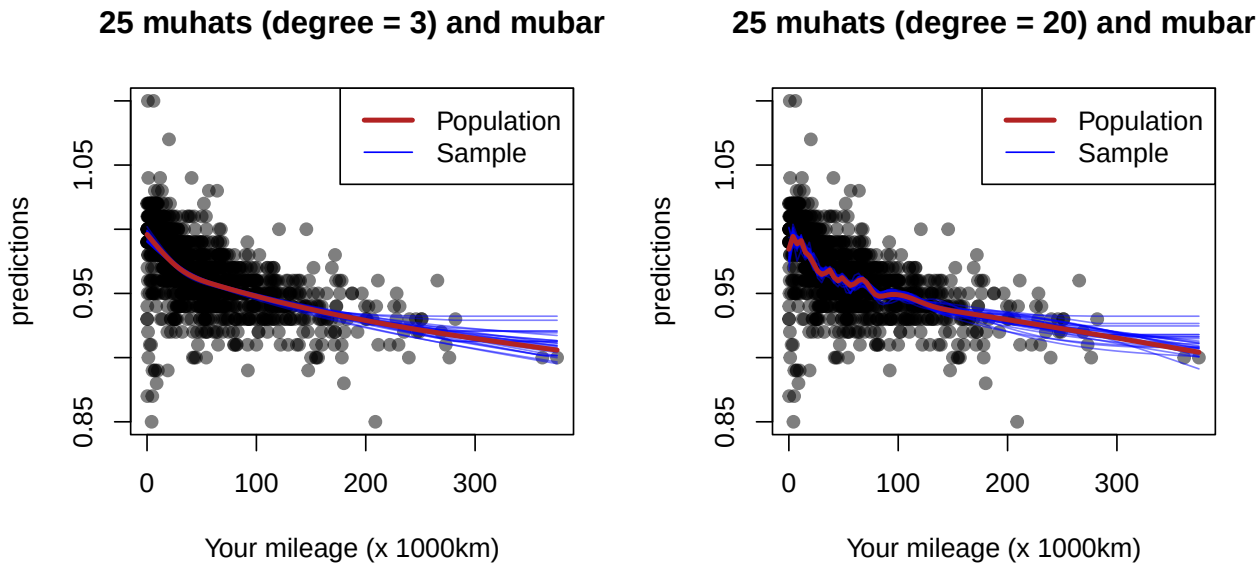
Generate $m = 25$ samples of size $n = 600$. Fit polynomials of degree 3 and 20 to every sample.

```
set.seed(341)
N_S      = 25
n        = 600
samps    = replicate(N_S, getSampleComp(pop, n), simplify = FALSE)
Ssamples = lapply(samps, function(Si) {getXYSample(xvarname, yvarname, Si, pop)})
Tsamples = lapply(samps, function(Si) {getXYSample(xvarname, yvarname, !Si, pop)})
muhats3  = lapply(Ssamples, getmuhat, complexity = 3)
muhats20 = lapply(Ssamples, getmuhat, complexity = 20)
```

Using `par(mfrow=c(1,2))` plot all the fitted polynomials with degree 3 and 20 on two different figures. Overlay the two fitted polynomials of degree 3 and 20 based on the whole population (make the colour of the population curves different from the others to make them stand out).

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

plot(pop[,c(xvarname, yvarname)],
     pch  = 19,
     col  = adjustcolor("black", 0.5),
     xlab = "Your mileage (x 1000km)",
     ylab = "predictions",
```

```
     main = paste0(N_S, " muhats (degree = 3) and mubar")
)

for (f in muhats3) curve(f(x), add = TRUE, col = adjustcolor("blue", 0.5))
curve(muhat3, add = TRUE, col = "firebrick", lwd = 3)

legend("topright",
       legend = c("Population", "Sample"),
       col = c("firebrick", "blue"),
       lty = c(1, 1),
       lwd = c(3, 1))

plot(pop[,c(xvarname, yvarname)],
     pch  = 19,
     col  = adjustcolor("black", 0.5),
     xlab = "Your mileage (x 1000km)",
     ylab = "predictions",
     main = paste0(N_S, " muhats (degree = 20) and mubar")
)

for (f in muhats20) curve(f(x), add = TRUE, col = adjustcolor("blue", 0.5))
curve(muhat20, add = TRUE, col = "firebrick", lwd = 3)

# Add legend for muhat20 and muhats20
legend("topright",
       legend = c("Population", "Sample"),
       col = c("firebrick", "blue"),
       lty = c(1, 1),
       lwd = c(3, 1))
```



*Using `var_mutilde` function, calculate the sampling variability of the function of the polynomials with degree equal to 3 and 20.**

```r
# Set options to prevent scientific notation
options(scipen = 999)

var_complexity3 <- var_mutilde(Ssamples, Tsamples, complexity = 3)
var_complexity20 <- var_mutilde(Ssamples, Tsamples, complexity = 20)

cat("sampling variability with degree equal to 3: ", var_complexity3, "\n")
```

```
## sampling variability with degree equal to 3:  0.000002732045
```

```r
cat("sampling variability with degree equal to 20: ", var_complexity20, "\n")
```

```
## sampling variability with degree equal to 20:  0.00001064541
```

Using `bias2_mutilde` function, calculate the squared bias of the polynomials with degree equal to 3 and 20.

```r
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)
}

muhat <- getmuFun(pop, xvarname, yvarname)
bias_complexity3 <- bias2_mutilde(Ssamples, Tsamples, muhat, complexity = 3)
bias_complexity20 <- bias2_mutilde(Ssamples, Tsamples, muhat, complexity = 20)

cat("squared bias  with degree equal to 3: ", bias_complexity3, "\n")
```

```
## squared bias  with degree equal to 3:  0.0004874026
```

```r
cat("squared bias  with degree equal to 20: ", bias_complexity20, "\n")
```

```
## squared bias  with degree equal to 20:  0.0004796404
```

Generate $m = 25$ samples of size $n = 1000$, and using `apse_all` function, calculate the APSE for complexities equal to `0:10`

```r
#Generate Samples
set.seed(341)
N_S      = 25
n        = 1000
samps    = replicate(N_S, getSampleComp(pop, n), simplify = FALSE)
Ssamples = lapply(samps, function(Si) {getXYSample(xvarname, yvarname, Si, pop)})
Tsamples = lapply(samps, function(Si) {getXYSample(xvarname, yvarname, !Si, pop)})
```

```r
apse_all <- function(Ssamples, Tsamples, complexity, mu){
  ## average over the samples S
  ##
  N_S <- length(Ssamples)
```

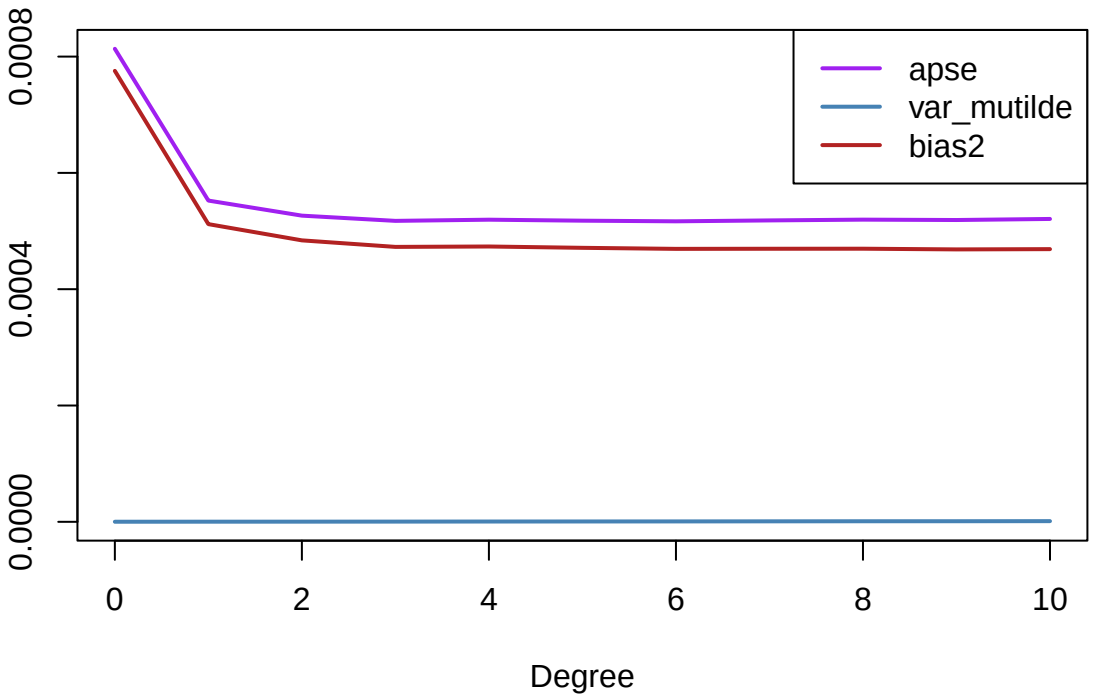
```
  muhats <- lapply(Ssamples,
                   FUN=function(sample) getmuhat(sample, complexity)
  )
  ## get the average of these, mubar
  mubar <- getmubar(muhats)

  rowMeans(sapply(1:N_S,
                  FUN=function(j){
                    T_j         <- Tsamples[[j]]
                    muhat       <- muhats[[j]]
                    ## Take care of any NAs
                    T_j         <- na.omit(T_j)
                    y           <- T_j$y
                    x           <- T_j$x
                    mu_x        <- mu(x)
                    muhat_x     <- muhat(x)
                    mubar_x     <- mubar(x)

                    ## apse
                    ## average over (x_i,y_i) in a
                    ## single sample T_j the squares
                    ## (y - muhat(x))^2
                    apse        <- (y - muhat_x)

                    ## bias2:
                    ## average over (x_i,y_i) in a
                    ## single sample T_j the squares
                    ## (y - muhat(x))^2
                    bias2       <- (mubar_x -mu_x)

                    ## var_mutilde
                    ## average over (x_i,y_i) in a
                    ## single sample T_j the squares
                    ## (y - muhat(x))^2
                    var_mutilde <- (muhat_x - mubar_x)

                    ## var_y :
                    ## average over (x_i,y_i) in a
                    ## single sample T_j the squares
                    ## (y - muhat(x))^2
                    var_y       <- (y - mu_x)

                    ## Put them together and square them
                    squares     <- rbind(apse, var_mutilde, bias2, var_y)^2

                    ## return means
                    rowMeans(squares)
                  }
  ))
}
```

```
complexities = 0:10
apse_vals    = sapply(complexities, function(complexity) {
                      apse_all(Ssamples, Tsamples, complexity = complexity, mu = muhat)
```

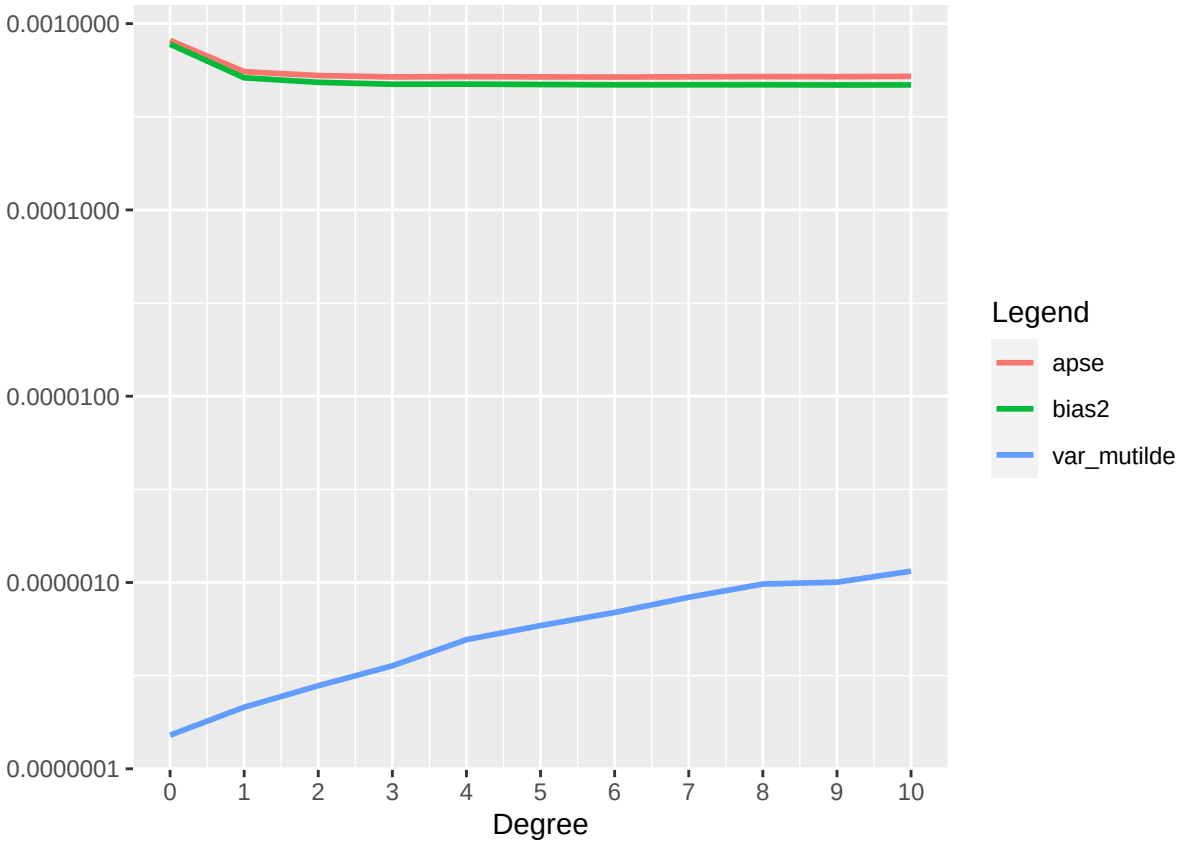
```
                    })

# Print out the results
t(rbind(complexities, apse=round(apse_vals, 10)))
```

```
##       complexities         apse  var_mutilde        bias2        var_y
##  [1,]            0 0.0008133585 0.0000001515 0.0007754422 0.0000403199
##  [2,]            1 0.0005523841 0.0000002141 0.0005119250 0.0000403199
##  [3,]            2 0.0005264704 0.0000002792 0.0004839842 0.0000403199
##  [4,]            3 0.0005175161 0.0000003571 0.0004727329 0.0000403199
##  [5,]            4 0.0005195211 0.0000004935 0.0004734282 0.0000403199
##  [6,]            5 0.0005178098 0.0000005873 0.0004711310 0.0000403199
##  [7,]            6 0.0005167006 0.0000006906 0.0004693328 0.0000403199
##  [8,]            7 0.0005182799 0.0000008331 0.0004695329 0.0000403199
##  [9,]            8 0.0005196260 0.0000009813 0.0004696315 0.0000403199
## [10,]            9 0.0005189800 0.0000010038 0.0004684157 0.0000403199
## [11,]           10 0.0005207853 0.0000011499 0.0004689020 0.0000403199
```

```
library(tidyr)
matplot(complexities, t(apse_vals[1:3,]),
        type = 'l',
        lty = 1,
        lwd = 2,
        col = c("purple", "steelblue", "firebrick"),
        xlab = "Degree",
        ylab = "")
legend('topright',
       legend = rownames(apse_vals)[1:3],
       lty = 1,
       lwd = 2,
       col = c("purple", "steelblue", "firebrick"))
```



```r
rbind(complexities, apse_vals[1:3,]) %>%
  t %>%
  as.data.frame %>%
  pivot_longer(-complexities) %>%
  ggplot(aes(x = complexities, y = value, group = name, colour = name)) +
  geom_line(lwd = 1) +
  xlab("Degree") +
  ylab("") +
  scale_x_continuous(breaks = complexities) +
  scale_y_log10() +
  labs(colour = "Legend")
```



Comment: The polynomial with degree 0 has the highest APSE at 0.0008133585, then it drops to roughly 0.00055 for polynomials with degree 1 and then polynomials with degrees 2 to 10 all have roughly the same aspe 0.00051 (to 2sf), the aspe plateaus.

Instead of randomly constructing sample and test sets we can use k-fold cross-validation.

Create a function creates the k-fold samples from a given population. i.e.- The function has arguments `k` the number of k-fold, `pop` a population, `xvarname` the name of the x variable, and `yvarname` the of the y variable. - The function outputs a list containing the k-fold samples and test samples labelled as `Ssamples` and `Tsamples`.

```r
sample.kfold <- function(k = NULL, pop = NULL, xvarname = NULL, yvarname = NULL) {
  N        = nrow(pop)
  kset     = rep_len(1:k, N)
```

```
  kset     = sample(kset)
  samps    = lapply(1:k, function(k) kset != k)
  Ssamples = lapply(samps, function(Si) getXYSample(xvarname, yvarname, Si, pop))
  Tsamples = lapply(samps, function(Si) getXYSample(xvarname, yvarname, !Si, pop))
  return(list(Ssamples = Ssamples, Tsamples = Tsamples))
}
```

Estimate of the APSE using $k = 5$ fold cross-validation when the `complexity=3`.

```
xvarname <- "Your.mileage.km"
yvarname <- "Remaining.battery.capacity"
kfold.samples = sample.kfold(k = 5, pop = tesla, xvarname  , yvarname)

# Get the muFun using the getmuFun function
po.muFun <- getmuFun(tesla, xvarname, yvarname)

p <- apse_all(kfold.samples$Ssamples, kfold.samples$Tsamples, complexity = 3, mu = po.muFun)
print(p)
```

```
##            apse     var_mutilde           bias2           var_y
## 0.0005353461575 0.0000008535456 0.0004855733993 0.0000424255040
```

```
cat("\n")
```

```
cat("APSE using k=5 fold cross-validation for complexity=3: ", p[1], "\n" )
```

```
## APSE using k=5 fold cross-validation for complexity=3:  0.0005353462
```

```
cat("var_mutilde: ", p[2], "\n" )
```

```
## var_mutilde:  0.0000008535456
```

```
cat("bias2: ", p[3], "\n" )
```

```
## bias2:  0.0004855734
```

```
cat("var_y : ", p[4], "\n" )
```

```
## var_y :  0.0000424255
```

Perform $k = 10$-fold cross-validation to estimate the complexity parameter from the set $0 : 12$. Plot APSE by the complexity.

```
complexities = 0:12
apse_vals    = sapply(complexities, function(complexity) {
                      apse_all(kfold.samples$Ssamples, kfold.samples$Tsamples,
                               complexity = complexity, mu = po.muFun)
                    })

# Print out the results
t(rbind(complexities, apse=round(apse_vals,7)))
```

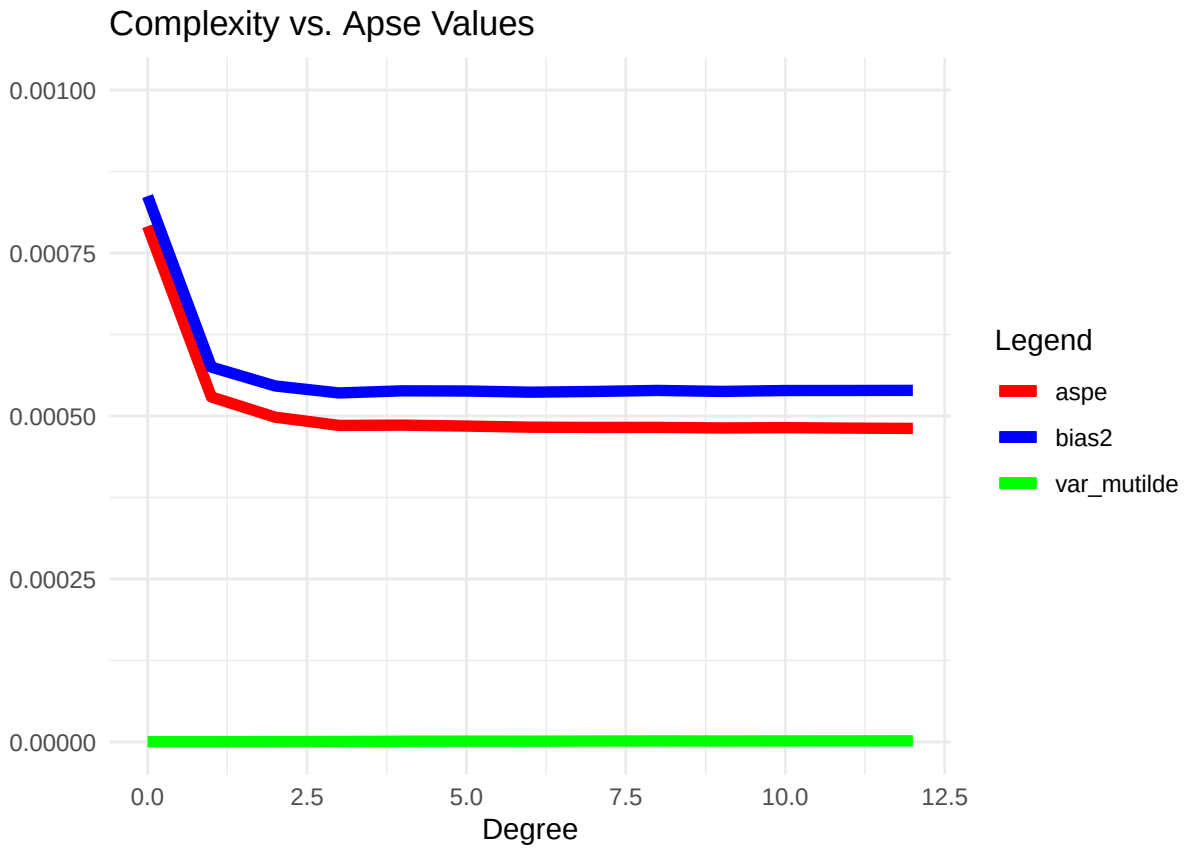
```
##       complexities      apse var_mutilde     bias2     var_y
##  [1,]            0 0.0008373   0.0000004 0.0007912 0.0000424
##  [2,]            1 0.0005750   0.0000004 0.0005292 0.0000424
##  [3,]            2 0.0005461   0.0000006 0.0004982 0.0000424
##  [4,]            3 0.0005353   0.0000009 0.0004856 0.0000424
##  [5,]            4 0.0005386   0.0000012 0.0004860 0.0000424
##  [6,]            5 0.0005383   0.0000013 0.0004845 0.0000424
```

```
##  [7,]            6 0.0005365   0.0000013 0.0004829 0.0000424
##  [8,]            7 0.0005376   0.0000015 0.0004824 0.0000424
##  [9,]            8 0.0005392   0.0000016 0.0004825 0.0000424
## [10,]            9 0.0005377   0.0000016 0.0004817 0.0000424
## [11,]           10 0.0005391   0.0000017 0.0004821 0.0000424
## [12,]           11 0.0005391   0.0000017 0.0004814 0.0000424
## [13,]           12 0.0005393   0.0000018 0.0004810 0.0000424
```

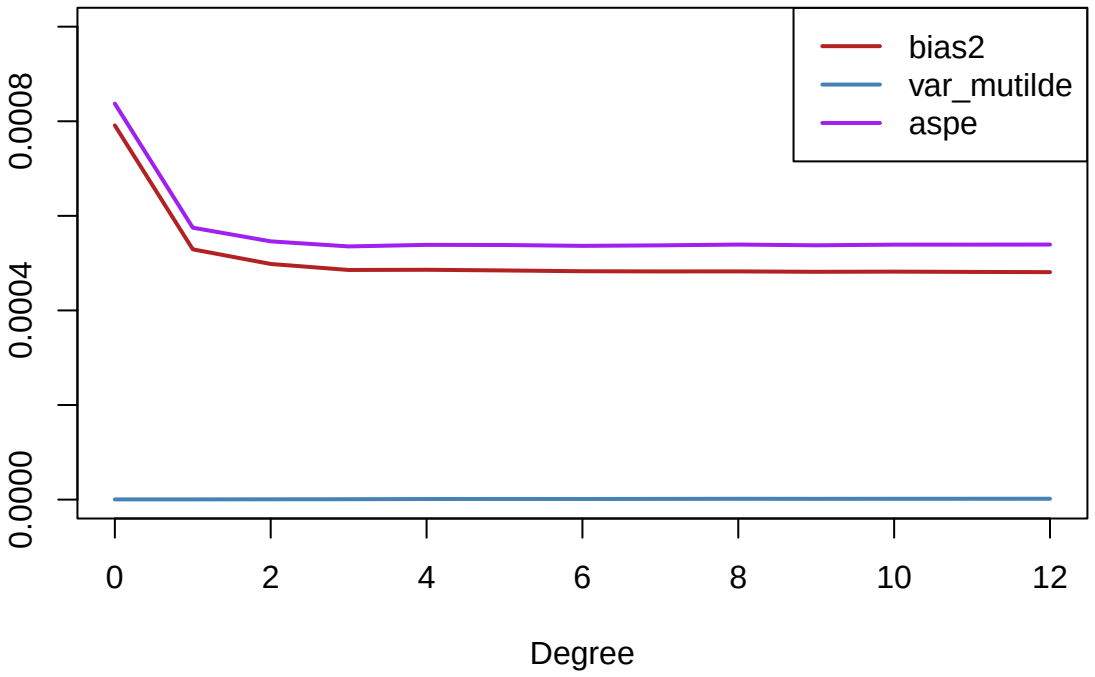
```
complexities[apse_vals[2, ] == min(apse_vals[2, ])]
```

```
## [1] 0
```

```
library(ggplot2)

# Add complexity degrees from 1 to 12 to the existing complexities variable
complexities <- 0:12

# Create a data frame with the apse values
apse_df <- data.frame(complexities, apse = apse_vals[3,], var_mutilde = apse_vals[2,], bias2 = apse_vals

# Plot using ggplot2
ggplot(apse_df, aes(x = complexities, y = apse)) +
  geom_line(aes(color = "aspe"), size = 2) +
  geom_line(aes(y = var_mutilde, color = "var_mutilde"), size = 2) +
  geom_line(aes(y = bias2, color = "bias2"), size = 2) +
  xlab("Degree") +
  ylab("") +
  ylim(0, 0.001) +
  labs(color = "Legend", title = "Complexity vs. Apse Values") +
  theme_minimal() +
  theme(legend.position = "right") +
  scale_color_manual(values = c("aspe" = "red", "bias2" = "blue", "var_mutilde" = "green"))
```




```
plot( complexities, apse_vals[3,], xlab="Degree", ylab="", type='l', ylim=c(0, 0.001), col="firebrick",
lines(complexities, apse_vals[2,], xlab="Degree", ylab="", col="steelblue", lwd=2 )
lines(complexities, apse_vals[1,], col="purple", lwd=2)

# Add legend
legend("topright", legend = c("bias2", "var_mutilde", "aspe"), col = c("firebrick", "steelblue", "purpl
```


bias2
var_mutilde
aspe
0.0000
0.0004
0.0008
0
2
4
6
8
10
12
Degree