testingTrainingValidation

September 23, 2019

1 Training, testing, and validation

The sum squares error **SSE** is a measure of how well a proposed model fits training data. But a major goal of regression, and statistics, is not to fit training data. A regression model is meant to generalize and predict future unseen data points well.

Suppose we have training data X_{train} and the corresponding values of interest y_{train}. Combining X and y together will be denoted D_{train}.

Also assume we have a prediction model $f(X)$

Then the **Mean Squared Error** is defined as

$$\text{MSE}(D, f) = \frac{\sum_{i=1}^{N}[y_i - f(x_i)]^2}{N}$$

If the MSE is computed on training data D_{train} we call this the **training MSE**

Our goal is not to perform well on data we've already collected, but to perform well on data our model was not trained from. We aim to minimize our **test MSE**

$$\text{MSE}(D_{\text{test}}, f)$$

where D_{test} is a data set containing X and y values our model (f) has not trained from.

1.1 $\textbf{MSE}_{\textbf{train}} \neq \textbf{MSE}_{\textbf{test}}$

It may be reasonable to assume minimizing the training MSE should also minimize the test MSE, but this is typically not the case. Let's look at our data from Class03, the polynomial regression data.

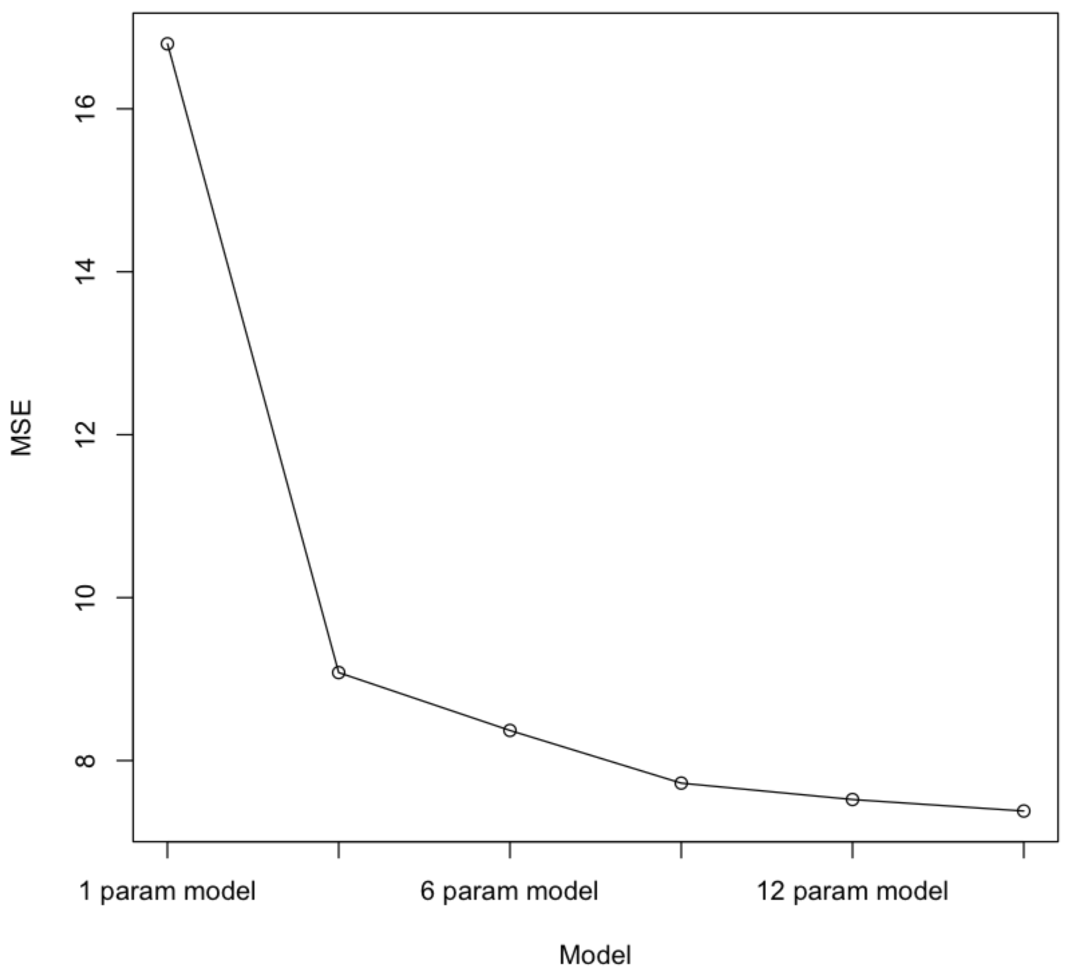
[92]:
```
data <- read.csv('polynomialData.csv')
print(head(data))
```

```
           x         y
1  0.9958723  8.2420054
2 -0.6556163  2.3114202
3 -0.9176787  4.0842076
```

```
4  0.1963727 -5.5386897
5  1.0309346  2.5166174
6  1.2610719 -0.5388713
```

We can lower our training MSE by fitting more and more complicated polynomials.

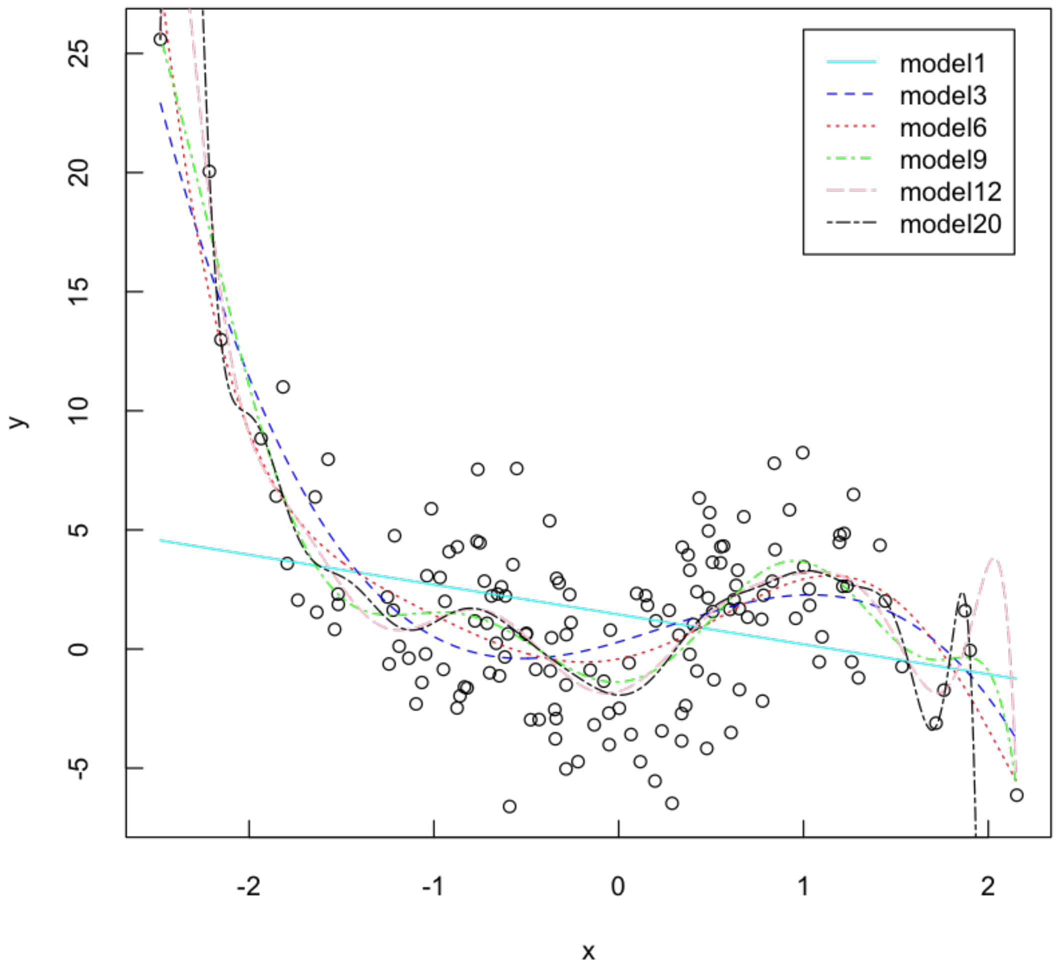
[93]:
```
model1 <- lm(y~x,data=data)
model3 <- lm(y~x+I(x^2)+I(x^3),data=data)
model6 <- lm(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6),data=data)
model9 <-␣
 ↪lm(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8)+I(x^9),data=data)
model12 <-␣
 ↪lm(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8)+I(x^9)+I(x^10)+I(x^11)+I(x^12),data=d
model20 <-␣
 ↪lm(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8)+I(x^9)+I(x^10)+I(x^11)+I(x^12)
                 +I(x^13)+I(x^14)+I(x^15)+I(x^16)+I(x^17)+I(x^18)+I(x^19)+I(x^20)
              ,data=data)


MSE = function(model,data){
     N = nrow(data)
     return( sum((predict(model,data) - data$y)^2)/N )
}

fromString2Model = function(string){
    return(eval(parse(text=string)))
}

i<-1
MSEs <- rep(0,6)
for (model in c("model1","model3","model6","model9","model12","model20")){
    MSEs[i] <- MSE(fromString2Model(model),data)
    i=i+1
}

plot(MSEs,xlab="Model",ylab="MSE",xaxt = "n")
lines(MSEs)
axis(1, at=c(1,2,3,4,5,6)
     , labels=c("1 param model","3 param model","6 param model","9 param␣
 ↪model","12 param model","20 param model"))
```



This looks like a more complicated model, one with more parameters, should always be better. Lets look at the functions graphically.

[94]:
```
# plot data

plot(data$x,data$y
     ,xlab="x"
     ,ylab="y"
     ,tck=0.02
)

# plot model predictions
minX <- min(data$x)
```

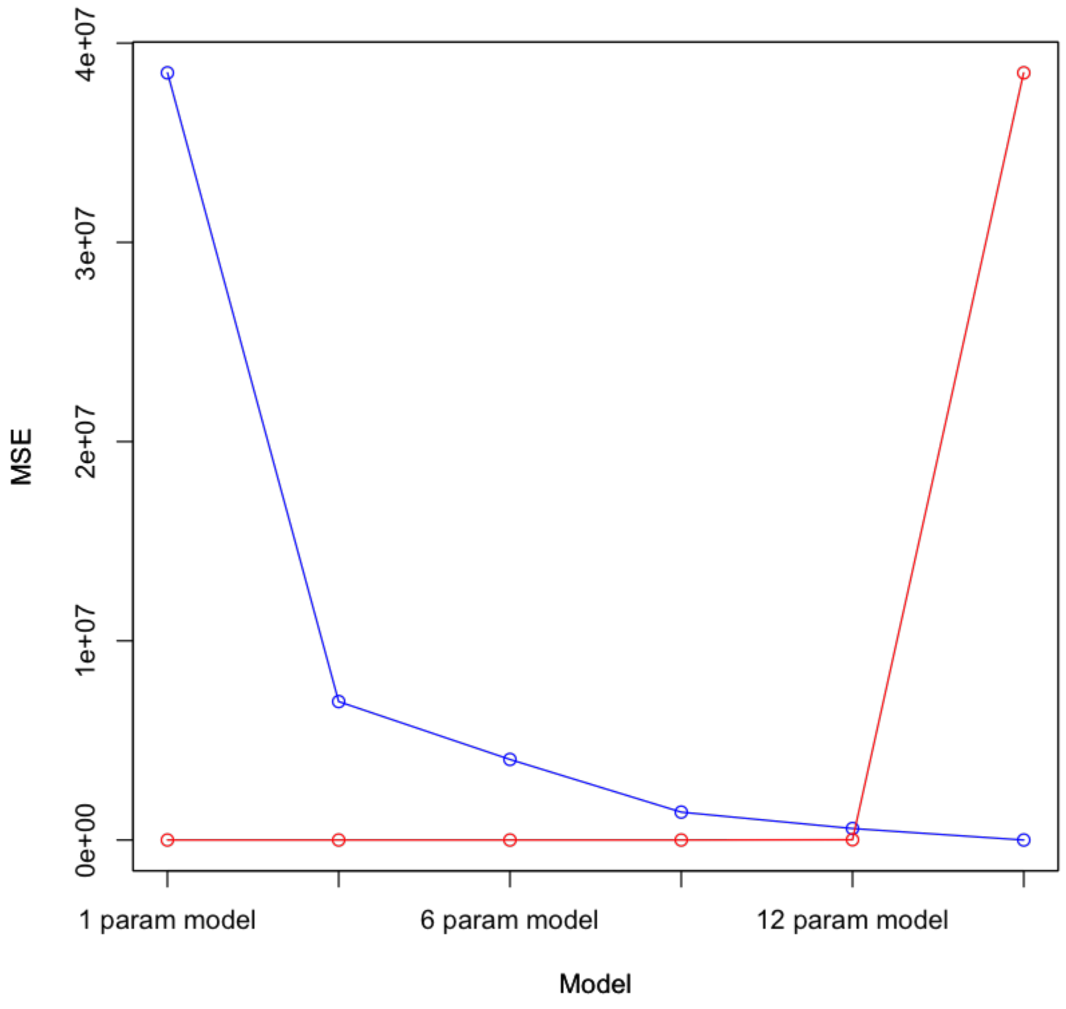
```r
maxX <- max(data$x)
newdata <- data.frame(x=seq(minX,maxX,0.01))

i<-1
colors = c('cyan','blue','red','green','pink','black')
for (model in c("model1","model3","model6","model9","model12","model20")){
    predictions <- predict(fromString2Model(model),newdata)
    lines(newdata$x,predictions,col=colors[i], lty=i)
    i<-i+1
}

legend(1,26,legend=c("model1","model3","model6","model9","model12","model20")
                      ,col=colors
                      ,lty=1:6
)
```



The more complicated models have smaller MSE, but also look like they may be learning the training data to well. We can evaluate the MSE on a set of test data the model hasn't trained on.

[106]:
```
testData <- read.csv('testData.csv')
print(head(data))

i<-1
tstMSEs <- rep(0,6)
for (model in c("model1","model3","model6","model9","model12","model20")){
    tstMSEs[i] <- MSE(fromString2Model(model),testData) # notice change here␣
 ↪from training to test data
    i=i+1
}
plot(MSEs,xlab="Model",ylab="MSE",xaxt = "n",yaxt = "n",col="blue")
lines(MSEs,xaxt = "n",col="blue")

par(new=TRUE)
plot(tstMSEs,xlab="Model",ylab="MSE",xaxt = "n",col="red")
lines(tstMSEs,col="red")

axis(1, at=c(1,2,3,4,5,6)
     , labels=c("1 param model","3 param model","6 param model","9 param␣
 ↪model","12 param model","20 param model"))

#legend()
```

```
           x          y
1  0.9958723  8.2420054
2 -0.6556163  2.3114202
3 -0.9176787  4.0842076
4  0.1963727 -5.5386897
5  1.0309346  2.5166174
6  1.2610719 -0.5388713
```



Though the training error gets better with more model complexity, the more complicated the model the higher the test mse. More complicated models **overfit** the training data. instead of learning to generalize, the model learns fluctuations in the training data.

1.2 Bias-variance tradeoff

The MSE is a function of our target of interest y, our predictions $\hat{y} = \hat{f}_T(X)$ that depend on our data. We saw previously, the MSE computed on our training data doesn't tell us much about our model's ability to predict new data. Instead we would like to measure the MSE on new training data.

Lets assume our variable of interest is generated by a function f. Complicated models will have more potential to learn the true function, f (our signal). Though a complicated model is likely sensitive to our training data. When fit to a different dataset a complicated model could change drastically. Simpler models may be "further" from the true function f but the model is less prone to changing if fit to a different set of training data. The Bias-Variance trade off tries to capture two competing concepts: the potential of a model to closely represent the true data-generating function f and the sensitivity of our model to the training data.

Let $\bar{f}(x)$ be our model's prediction averaged over an infinitely many number of training sets of N_T observations, or $\bar{f}(x) = \sum_{T=1}^{\infty} f_T(x) \Big/ N_T$.

Bias - The bias measures the expected difference between $\bar{f}(x)$ and the true average value $\bar{y} = f(x)$ or Bias=$[\bar{y} - \bar{f}(x)]$.

Variance - The variance measures how our model predictions computing from a finite training set $f_T(x)$ differ from$\bar{f}(x)$ or Variance=$E[f_T(x) - \bar{f}(x)]^2$.

ϵ^2 - $E(y - \bar{y})^2$ an irreducible error made by assumptions about the form of the problem, a quantity we cannot change.

We can decompose our MSE over a single test point as

$$\begin{aligned} \text{MSE} &= E[y - \hat{f}_T(x)]^2 && (1)\\ &= E[y - \bar{y} + \bar{y} - \hat{f}_T(x)]^2 && (2)\\ &= E[y - \bar{y}]^2 + E[\bar{y} - \hat{f}_T(x)]^2 && (3)\\ & && (4) \end{aligned}$$

We broke our MSE into two parts. The first part is ϵ^{2} a value we cannot change since it doesn't involve our prediction model. The second term above can be broken down further by including $\bar{f}$.

$$\begin{aligned} E[\bar{y} - \hat{f}_T(x)]^2 &= E[\bar{y} - \bar{f}(x) + \bar{f}(x) - \hat{f}_T(x)]^2 && (5)\\ &= E[\bar{y} - \bar{f}(x)]^2 + E[\bar{f}(x) - \hat{f}_T(x)]^2 && (6)\\ & && (7) \end{aligned}$$

and now including our third term from above we have

$$\text{MSE} = \epsilon^2 + \text{Bias}^2 + \text{Var}(f) \qquad (8)$$

Our MSE is composed of three different parts. Fixing MSE and ϵ^2, if we reduce our Bias the Variance must increase and vice-versa.

1.3 Cross-Validation

In a best-case scenario you can fit a model to a set of training data and collect additional data for testing. Your model can be fit to training data, used to predict your target variable (y) on the testing set, and those predictions $(\hat{y})$ can be compared to the truth (y).

Because of financial limitations, time burden, and other constraints, statistical modelers are typically unable to collect additional testing data. Instead resort to splitting our data set into training and testing. Our model can learn from the training dataset and test predictions on the test dataset.

Hold-out A straight-forward method is to split our data into a single training **Tr** and testing **Tst** dataset. The model is trained on **Tr** and tested on **Tst**. But splitting our data into a single training/testing dataset may bias our model. We want to assess the model fit independent of how we split our data into training and testing.

K-fold Cross-validation splits the all the data into K distinct sets at random. The total dataset (D) is then a union of K data pieces (P_k)

$$D = \cup_{i=1}^{K} P_k$$

From $i = 1$ to K, remove the ith dataset P_i from the total dataset D. Train the model on $D - P_i$, compute test statistics of the model on P_i, and store those test statistics in a list L. Summary statistics can be computed from L.

For example, we can compute the MSE on every train/test split and append all the MSEs to L. A common summary statistics, called the cv_{error} is the average MSE over all train/test splits.

[18]:
```
data <- read.csv('polynomialData.csv')

#split into K  random pieces
K=10

dataPieces = split(data[sample(nrow(data)),],1:10)
MSES = sapply(dataPieces
       ,function(TestData){
           rowsOfTstData <- as.numeric(row.names(TestData))

           trainingData <- data[-rowsOfTstData,]
           model <- lm(y~x,data=trainingData)

           MSE = function(model,data){
             N = nrow(data)
             return( sum((predict(model,data) - data$y)^2)/N )
            }
            return(MSE(model, TestData))
})

cvError = mean(MSES)
print("cvError")
print(cvError)
```

```
[1] "cvError"
[1] 18.28039
```

[ ]:
```
```