Workshop: Introduction to Statistical Learning Part I

Mohammad Jafari Jozani

February 14, 2021

Contents

1 What is Statistical Learning?

Statistical Learning is considered as an activity or process of gaining knowledge or skill by studying, practicing, being taught, or experiencing something. Statistical learning is a fundamental ingredient in the training of a modern data scientist. For a data scientist it is important to understand the ideas behind the various techniques, in order to know how and when to use them. One has to understand the simpler methods first, in order to grasp the more sophisticated ones. It is important to accurately assess the performance of a method, to know how well or how badly it is working. Sometimes simpler methods perform as well as fancier ones. Statistical learning is an exciting research area, having important applications in science, industry and finance.

There are several applications for Statistical learning. For example:

1. Identify the risk factors for prostate cancer.
2. Classify patients to those having osteoporosis or not having it using DXA images obtained from patients.
3. Predict whether someone will have a heart attack on the basis of demographic, diet and clinical measurements. Another similar problem is predicting Alzheimer's disease years in advance.
4. Customize an email spam detection system.
5. Identify the numbers in a handwritten zip code.
6. Classify a tissue sample into one of several cancer classes, based on a gene expression file.
7. Establish the relationship between salary and demographic variables in population survey data.
8. Classify the pixels in a LANDSAT image, by usage.

In most of these problems we have an independent variable or outcome Y and one or more dependent variable or covariates $\mathbf{X} = (X_1, \ldots, X_p)$. One usually observes these variables for various "subjects". The goal is to figure our what effects do the covariates have on the outcome? How well can we describe these effects? Can we predict the outcome using the covariates?

In general we write

$$Y = f(\mathbf{X}) + \epsilon,$$

where ϵ captures measurement errors and other discrepancies including the effect of some important variables that are not included in the study. Usually it is assumed that $\mathbb{E}[\epsilon|\mathbf{X}] = 0$ with $Var(\epsilon|\mathbf{X}) = \sigma^2$ being fixed but unknown. Also, we assume that $\mathbf{X}$ is given and even if it is random, after observing it we consider it as fixed values. There is another way of looking at this problem when $\mathbf{X}$ is not fixed. Many recent papers are published on this subject and if you are interested let me know to introduce you a few very interesting papers to read.

2 Supervised, reinforcement and unsupervised learning

If we knew $f(\mathbf{X})$ we could use that to predict the Y value for a new individual with a given $\mathbf{X} = \mathbf{x}$. We could also understand which components of $\mathbf{X}$ are important in explaining Y and which are not relevant. If $f(\mathbf{X})$ is not very complicated we might be able to understand how each X_i in $\mathbf{X}$ affects Y. In general we could use the model for inference and/or prediction.

The problem is that $f(\mathbf{X})$ is unknown and needs to be estimated. In a statistical learning course we learn different techniques to estimate $f(\mathbf{X})$ and dig in more to better understand how these methods are developed, what are their properties, for what type of problems they are most effective and how one can extend the methods to obtained new ones. We learn how to evaluate the performance of each method and how to deal with problems where Y is not numeric or $\mathbf{X}$ is very high dimensional.

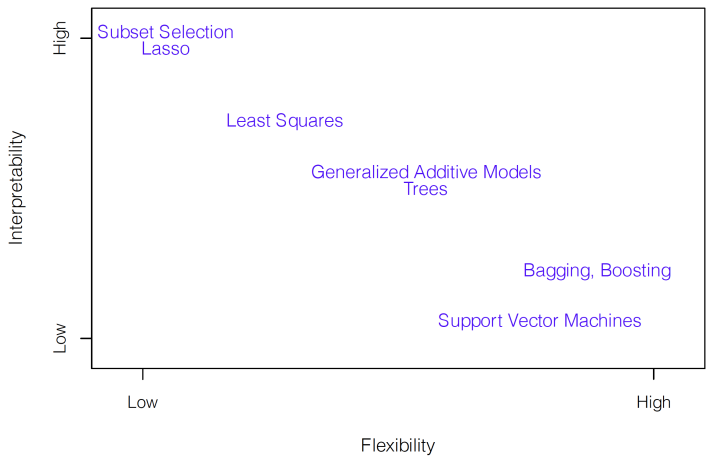


Figure 1: The relationship between flexibility of statistical learning models and their interpretability (Source: Elements of Statistical Learning)

There are different ways to estimate $f(\mathbf{X})$ and you have already seen a few of such techniques in your previous courses. Some of these methods are very complicated and some are simple. In practice there are some trade-offs that you should be aware of as a data scientist:

1. **Prediction accuracy versus interpretability**. For example linear models are easy to interpret but additive models based on cubic splines are not.
2. **Good fit versus over-fit or under-fit**. You should learn how to know when the fit is just right?
3. **Parsimony versus black-box**. We often prefer a simpler model involving fewer variables over a black-box predictor involving them all. For example a deep neural network could provide a good fit and accurate prediction in many situations, but it is not often a useful method due to the lake of parsimony. In many medical applications decision makers want to see parsimony and often do not trust black boxes.

The above problems are called supervise learning as there is always a response variable Y which will supervise us during the estimation (model building) process and will tell us if we have done a good job in estimating $f(\mathbf{X})$.

In other words suppose we fit a model $\hat{f}(\mathbf{x})$ to some training data set $\mathcal{T} = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R},\ i = 1, \ldots, n\}$ and we wish to see how well it performs. One can compute the average squared prediction (estimation) error over $\mathcal{T}$ to get the **training error**:

$$\text{MSE}_{Tr} = \text{Average}_{i \in \mathcal{T}}[y_i - \hat{f}(\mathbf{x}_i)^2],$$

which may be biased toward more overfit models. Instead, if possible we should compute this using a test data that was not used in the training process of $\hat{f}(\mathbf{x})$. Suppose $\mathcal{T}_e = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R},\ i = 1, \ldots, m\}$ is a test data of size m. After finding $\hat{f}(\mathbf{x})$ using the training data one can get a good idea of its performance using the estimated **test error**:

$$\text{MSE}_{Te} = \text{Average}_{i \in \mathcal{T}_e}[y_i - \hat{f}(\mathbf{x}_i)^2],$$

In practice, and as we will show it later in this chapter, increasing the complexity of the fitted model will often decreases the value of the training error MSE_{Tr} but increases the test error MSE_{Te}. As prediction is more important and often is the ultimate goal, finding a model that has a small MSE_{Te} is desirable and often this is not obtained by using very complex models. Such models tend to overfit the training data as they will end up memorizing the data and have a poor performance when they are faced with data that are not similar to the training data set. Imagine that a student start memorizing all the questions in a note without actually learning or understanding the idea behind the topic. Of course student will perform very well on any question from the note but what is going to happen if the student faces a new sets of problems from the same topic but very different from those he or she has memorized?

In general we have three types of problems in a statistical learning course:

1. **Supervised learning**: Here we have a clearly defined response variable (or variables) Y and a set of covariates $\mathbf{X}$ and the goal is to find the relationship between Y and $\mathbf{X}$ in order to perform statistical inference, prediction, etc. The response variable is like a teacher in this type of problems and it is used to supervise us and show us our mistakes until we get good at what we do.

2. **Reinforcement learning**: Here the ultimate goal is to develop learning system that receives a reward signal and tries to learn to maximize the reward signal. We do not consider this type of learning in this course but feel free to consult with the book Reinforcement Learning: An Introduction by Sutton and Barto (2018).

3. **Unsupervised learning**: Here there is no clearly defined objective and we often do not have a predefined response variable Y. So, nothing is there to supervise us and nth ultimate goal is to explore the data and look for interesting patterns in the data. Usually this is the starting point to perform a subsequent supervised learning.

3 Different methods for estimating a regression function

Let us get back to our supervised learning problem and suppose after observing $\mathbf{X}_i = \mathbf{x}_i$ we want to estimate Y_i using $f(\mathbf{x}_i)$. What is the best $f(\cdot)$ that we can use to do this take? Recall that for a general model, and in particular the i-th observation, we have

$$Y_i = f(\mathbf{x}_i) + \epsilon_i$$

with $\epsilon_i = Y_i - f(\mathbf{x}_i)$ such that $\mathbb{E}(\epsilon_i|\mathbf{x}_i) = 0$. Consider the squared error loss function

$$L(Y_i, f(\mathbf{x}_i)) = (Y_i - f(\mathbf{x}_i))^2, \tag{1}$$

as a criterion to calculate the error for using $f(\mathbf{x}_i)$ instead of the true value Y_i. Given the conditional distribution of $Y|\mathbf{X} = \mathbf{x}$, it is easy to see that the best $f(\mathbf{x}_i)$ that can be used is indeed the regression function $m(\mathbf{x}_i) = \mathbb{E}[Y_i|\mathbf{X}_i = xx_i]$. So, the above model can be approximated with

$$\begin{aligned} Y_i &= m(\mathbf{x}_i) + \epsilon_i \\ &= \mathbb{E}[Y_i|\mathbf{X}_i = \mathbf{x}_i] + \epsilon_i. \end{aligned} \tag{2}$$

In other words, Y_i can be considered as the sum of the value of the regression function at $\mathbf{X} = \mathbf{x}_i$ and some error ϵ_i, where expected value of the error is 0. So, one can use $m(\mathbf{x}_i)$ instead of Y_i and be sure that this function is the best function one can use to predict Y_i. This sounds great, but why don't we use this in practice? and if this is that simple, why do we have all these statistical and machine learning techniques such as SVM, deep learning, LDA, QDA, spline regression models, kNN, etc.

The problem is that we do need to know the distribution of Y give $\mathbf{X} = \mathbf{x}$ (for fixed $\mathbf{X}$ strategy) or sometimes the joint distribution of Y given $\mathbf{X}$ (for random $\mathbf{X}$ strategy) to be able to calculate $\mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$. If we knew this we could easily calculate $m(\cdot)$ and the problem was solved.

In practice people address this problem in different ways. For example, one might assume a specific distributional form between Y and $\mathbf{X}$ such as multivariate normality, etc. When $\mathbf{X}$ is high dimensional (which is the case for many modern applications of data science) verifying such assumptions is extremely difficult. Some people have tried to make some parametric assumptions about $m(\mathbf{x})$ itself. For example, one might assume that $m(\mathbf{x}) = \beta_0 + \mathbf{x}^\top \boldsymbol{\beta}$, etc. All these spline, cubic spline, polynomial and additive regression models are attempts to accurately estimate $m(\cdot)$.

Parametric models (e.g., linear models) make strong assumptions about the structure of the relationship between the set of features $\mathbf{X}$ and the response variable Y that might not be true. These models often yield stable but possibly inaccurate predictions. In the real world, most of the data does not follow the typical theoretical assumptions made.

Another approach is to use non-parametric methods. When we say a technique is non-parametric , it means that it does not make any assumptions on the underlying data distribution. In other words, the model structure is determined from the data. One of such methods in the k-Nearest Neighbors (kNN) method that makes very mild structural assumptions and its predictions are often accurate but can be unstable. kNN is a nonparametric approach in statistical learning that is used for classification and regression. In general there are four paradigms of nonparametric approach towards classification and/or regression problems. These include

1. **Local averaging** (e.g., Partitioning estimates, Nadaraya-Watson kernel estimate, kNN, etc.),

2. **Local modeling** (e.g., local polynomials, etc.),
3. **Global averaging** (e.g., least squares), and
4. **Penalized modeling** (e.g., penalized least squares, cubic splines, thin-plate splines, etc.).

We saw that Y_i can be considered as the sum of the value of the regression function at $\mathbf{x}_i$ and some error ϵ_i, where expected value of the error is 0. This motivates the construction of the estimates by *local averaging*, i.e., estimation of $m(\mathbf{x}_i)$ by averaging the y_j values of those data points that their corresponding $\mathbf{x}_j$'s are close to $\mathbf{x}_i$. That is

$$\hat{m}(\mathbf{x}_i) = \sum_{j=1}^{n} W_{n,j}(\mathbf{x}_i; \mathbf{x}_1, \ldots, \mathbf{x}_n) Y_j, \tag{3}$$

where the weights $W_{n,j}(\mathbf{x}_i; \mathbf{x}_1, \ldots, \mathbf{x}_n)$ depend on $\mathbf{x}_1, \ldots, \mathbf{x}_n$, are non-negative weights and get smaller as $\mathbf{x}_j$ gets far from $\mathbf{x}_j$. This helps to estimate the Y_i value corresponding with $\mathbf{x}_i$ by borrowing information from close-by observations but giving more weights to those in the proximity of $\mathbf{x}_i$ and less weights to those that are further away from $\mathbf{x}_i$.

4 kNN as a local averaging method for estimating a regression function

As we said, kNN estimate is an example of a local averaging method that is memory-based and works very well in many applications. One can also rewrite kNN estimators as an example of kernel methods having a data dependent metric. In kNN given a point $\mathbf{x}_0$, we find the k training points $\mathbf{x}_{(r)}$, $r = 1, \ldots, k$, closest in distance to $\mathbf{x}_0$ and the estimate of the y-value associated with $\mathbf{x}_0$ is obtained using the $y_{(r)}$ values in the neighborhood. For example, in a classification problem when $y \in \{0, 1\}$, we simply use a majority vote among the k-neighbors of $\mathbf{x}_0$. If $\mathbf{x}_i$ and $\mathbf{x}_j$ are equidistant from $\mathbf{x}_0$ there will be a tie. There are several rules for tie breaking. For example, one can use randomization. Also, $\mathbf{x}_i$ might be declared closer if $i < j$, i.e., the tie breaking is done by indices. For the sake of simplicity we assume that the probability of ties is zero.

kNN can also be used for regression. In this approach the predicted value of Y at $\mathbf{x}$ is the average (or median) of the values of its k nearest neighbors.

As mentioned above, kNN is a memory-based approach as it requires no model to fit and to make prediction one needs to have access to all the data points. In other words, kNN does not use the training data points to do any generalization and there is no explicit training phase or it is very minimal. This also means that the training phase is pretty fast. Lack of generalization means that kNN keeps all the training data. To be more exact, all (or most) of the training data is needed during the testing phase.

Suppose we have a training data set

$$\mathcal{T} = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R},\ i = 1, \ldots, n\}$$

and suppose $\mathbf{x}_0 \in \mathbb{R}^P$ is a new observation. kNN Algorithm is based on feature similarity. In this approach, how closely out-of-sample features resemble our training set determines how we classify a given data point. To formalize this, let

$$d_i(\mathbf{x}_0) = d(\mathbf{x}_i, \mathbf{x}_0), \quad i = 1, \ldots, n,$$

In the rest of this chapter we use the *Euclidean distance* as our distance measure, that is

$$d_i(\mathbf{x}_0) = ||\mathbf{x}_0 - \mathbf{x}_i|| = \sqrt{\sum_{j=1}^{p} (\mathbf{x}_{0j} - \mathbf{x}_{ji})^2}, \quad i = 1, \ldots, n.$$

Remark 1. *Make sure that you repeat the numerical analysis in this Chapter using other distance measures to see how results are changing with the metrics.*

Remark 2. *Typically, we first standardize each of the features to have the mean* 0 *and variance* 1. *When we have more than one feature, components of* $\boldsymbol{X}$ *might have different scales and magnitudes. For example, suppose we are working with 3 features and one of them takes values between 0 and 1 while the other two take very large values. After calculating the distance measures, there might be cases where the feature with small scale values become uninformative and the algorithm would essentially rely on dimensions whose values are substantially larger.*

Using $d_i(\mathbf{x}_0)$ one can reorder the data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ according to increasing values of $d_i(\mathbf{x}_0)$. To this end, let

$$d_{(1:n)}(\mathbf{x}_0) \leq d_{(2:n)}(\mathbf{x}_0) \leq \ldots \leq d_{(n:n)}(\mathbf{x}_0)$$

be the ordered distance of $\mathbf{x}_i$'s from $\mathbf{x}_0$. In kNN we define the k-nearest neighbors to $\mathbf{x}_0$ as those $(\mathbf{x}_i, y_i) \in \mathcal{T}$ such that $d_i(\mathbf{x}_0) \leq d_{(k:n)}(\mathbf{x}_0)$. Let the indexes of those points be

$$\mathcal{N}_k(\mathbf{x}_0) = \{i \in \{1, \ldots, n\} \ : \ d_i(\mathbf{x}_0) \leq d_{(k:n)}(\mathbf{x}_0)\}.$$

In a 1NN or simply the nearest neighbor (NN) case

$$\mathcal{N}_1(\mathbf{x}_0) = i^* = \text{argmin}_{i=1,\ldots,n} ||\mathbf{x}_0 - \mathbf{x}_i||.$$

The idea is now to use these nearby data points to $\mathbf{x}_0$ to predict its associated value y_0.

The parameter k controls the stability of the kNN estimates. When k is small the algorithm is sensitive to the data and reduces to NN when $k = 1$. When k increases the estimates become more stable. kNN estimators are very simple and have many applications. By varying the number of neighbors k, one can achieve a wide range of flexibility in the estimate function $\widehat{m}(\mathbf{x})$, with small k corresponding to a more flexible fit (low bias and large variance) and large k less flexible (large bias and low variance).

Remark 3. *It may appear that kNN fits have a single parameter* k*, the number of neighbors. The effective number of parameters of the kNN approach is* $\frac{n}{k}$ *and is generally bigger than the number of parameters one need to estimate in a multiple regression (i.e.,* p*), in our formulation, and gets smaller as* k *increases. To get an idea why this is the case note that if the neighborhoods were non-overlapping, there would be* $\frac{n}{k}$ *neighborhood and we would fit one parameter (a mean) in each neighborhood.*

In general, kNN estimators show satisfactory performance in lower dimensions. If special structure between the input $\mathbf{X}$ and output Y known to exist, this can be used to reduce both the bias and the variance of the estimates.

4.1 kNN for Regression

Given a random pair $(\mathbf{X}, Y) \in \mathbb{R}^p \times \mathbb{R}$, recall that the best predictor of Y under the SEL function (1) is given by (2)

$$m(\mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}],$$

which is called the regression function of Y on $\mathbf{X}$. To construct a kNN estimate of $m(\mathbf{x}_0)$, we fix an integer $k \geq 1$ and define $W_{n,i}(\mathbf{x}_0; \mathbf{x}_1, \ldots, \mathbf{x}_n) = \frac{1}{k}$ in (3) for all the k points that are closest to $\mathbf{x}_0$. In other words,

$$\widehat{m}_k(\mathbf{x}_0) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_0)} y_i,$$

where $\mathcal{N}_k(\mathbf{x}_0)$ contains the indices of the k closest points in $\mathbf{x}_1, \ldots, \mathbf{x}_n$ to $\mathbf{x}_0$. The kNN approach assumes the true function $f(\cdot)$ can be well approximated by locally constant function, which results in using a mean in each neighborhood as an estimate.
Note that in $\widehat{m}_k(\mathbf{x})$ two approximations are happening:

1. expectation is approximated by averaging over sample data
2. conditioning at a point $\mathbf{x}_0$ is relaxed to conditioning on some region close to the target point.

Remark 4. *Under mild regularity conditions on the joint distribution of* $(\boldsymbol{X}, Y)$*, one can show that as* $n, k \longrightarrow \infty$ *such that* $\frac{k}{n} \longrightarrow \infty$ *than* $\widehat{m}_k(\boldsymbol{x}) \longrightarrow \mathbb{E}[Y|\boldsymbol{X} = \boldsymbol{x}] = m(\boldsymbol{x})$*. Although this seems to be a very general result, the rate of convergence is decreases exponentially with the dimension* p*, due to the curse of dimensionality.*

Let us visualize kNN estimators using a very simple data set. To this end consider a data set when we have an scalar $X \in [0, 1]$ which is associated with Y through the following model:

$$m(x) = \cos(10x) + 4x^3 - 5x$$

We generate $n = 15$ observations from this model using

$$y_i = m(x_i) + \epsilon_i, \quad i = 1, \ldots, 15,$$

where ϵ_i are iid normal random noises with mean 0 and standard deviation 0.5 resulting in the following scatter plot:

```
# set the seed to reproduce data generation in the future
set.seed(1)
x <- runif(15,0,1)
e <- rnorm(15,0,0.5)
y <- cos(10*x) + 4*x^3 -5*x+e
par(mfrow=c(1, 2))
plot(x, y, pch=20, col="darkgreen")
plot(x, y , pch=20, col="darkgreen")
curve(cos(10*x) + 4*x^3 -5*x, from=0, to= 1,
      col="red", lwd=2, add=TRUE, lty=2)
```

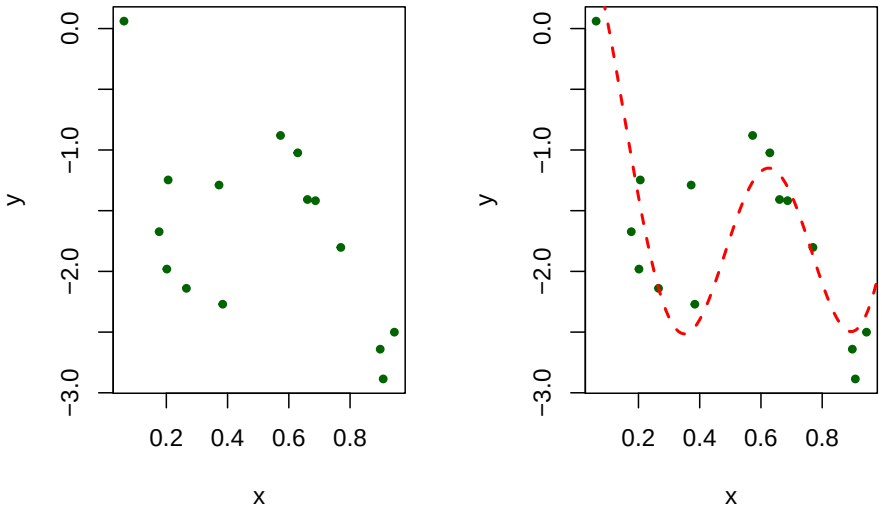


Figure 2: Simulated data from a population on the left where the true regression function is depicted by a red dashed line in the right graph.

Here we only have 15 points. Let us study the behavior of kNN regression estimators with respect to the choice of k. We show the kNN regression estimators for some values of k. To perform kNN for regression, we will need `knn.reg()` from the `FNN` package. Notice that, we do not load this package, but instead use `FNN::knn.reg` to access the function. Note that, in the future, we'll need to be careful about loading the `FNN` package as it also contains a function called `knn`. This function also appears in the `class` package which we will likely use later. The structure of this function is as follows

$$\text{knn.reg(train} = ?, \text{ test} = ?, \text{ y} = ?, \text{ k} = ?)$$

where

- `train=` the x's of the training data
- `test=` the x's at which we would like to make predictions
- `y=` the response for the training data
- `k=` the number of neighbors to consider

```
library(FNN)
par(mfrow=c(2, 2), mar=c(4, 3, 1, 1))
grid2 <- data.frame(x)
for(i in  c(2, 5, 8, 15)){
   pred.values <-  FNN::knn.reg(train = x, test = grid2, y = y, k = i)
   plot(x,y, main=paste("k=", i), pch=20, col="darkgreen")
   curve(cos(10*x) + 4*x^3 -5*x, add=TRUE, lty=2, col="red", lwd=2)
   ORD <-  order(grid2$x)
   lines(grid2$x[ORD],pred.values$pred[ORD], col="blue", lwd=2)
   }
```

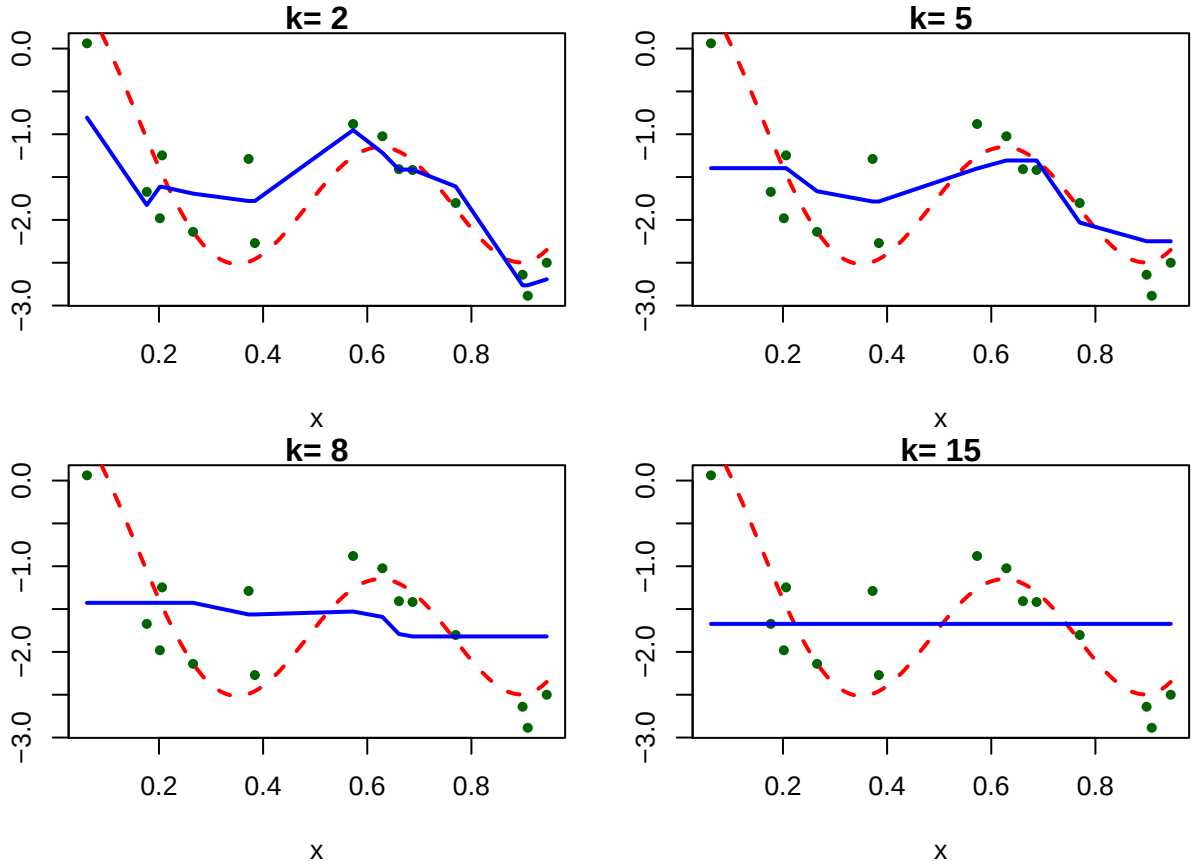


Figure 3: KNN regression estimates fitted to our data points using different values of k

First of all by looking at the true model and what is fitted or might be fitted we can see that *almost all the models that we will end up fitting will be wrong but some of them might be useful*, because there is a very low chance to exactly guess or obtain the true underlying model. Anyway, using the above analysis we see that kNN regression approach is a lazy learner and does not build an explicit model for the relationship between $\mathbf{x}$ and y. For small k is very unstable and when k gets larger it becomes more stable. However, as you see by increasing k we increase the bias and reduce the variability of the estimator.

Also, we observe that KNN regression estimators are very jagged. This can be easily explained by rewriting kNN regression estimator $\widehat{m}_k(\mathbf{x})$ as follow

$$\widehat{m}_k(\mathbf{x}) = \sum_{i=1}^{n} w_i(\mathbf{x}) y_i = (w_1(\mathbf{x}), \ldots, w_n(\mathbf{x})) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix},$$

with

$$w_i(\mathbf{x}) = \begin{cases} \frac{1}{k} & \text{if } \mathbf{x}_i \text{ is one of the } k \text{ neigbours to } \mathbf{x} \\ 0 & \text{Otherwise} \end{cases}.$$

Since $w_i(\mathbf{x})$ is discontinuous as a function of x, therefore $\widehat{m}_k(\mathbf{x})$ will also be discontinuous. Now, let us increase the sample size and see how these estimators behave for larger sample sizes.

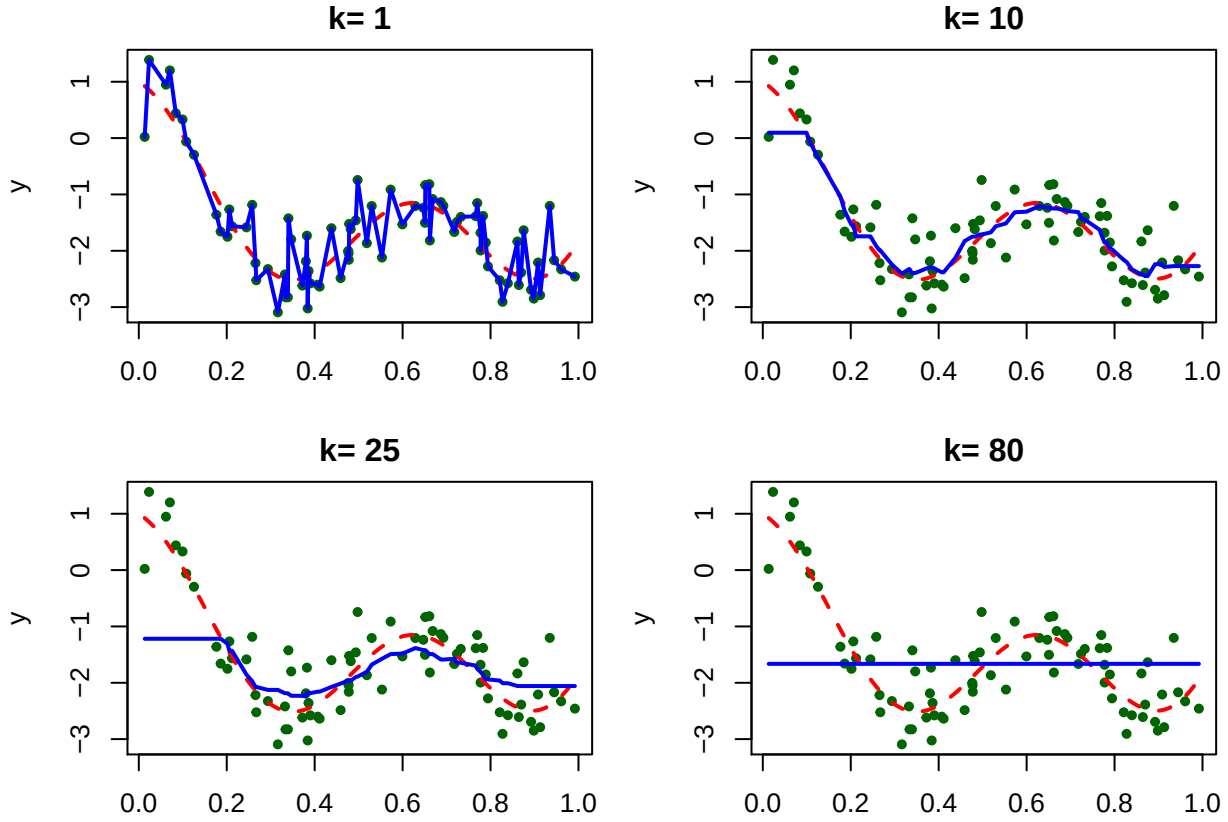


Figure 4: Larger size simulated data from our previous population and its kNN regression estimates with different values of k.

It seems that k plays an important role and needs to be selected carefully. We see that $k = 1$ is clearly over-fitting, as it produced a highly variable model. Conversely, $k = 80$ is clearly under-fitting the data, as $k = 80$ is a very simple, low variance model. In fact, here it is predicting a simple average of all the data at each point.

Choosing k: One can choose k using cross-validation which will be discussed in later chapters. Another method is to set aside a test data and choose a k that works well on the test data. Let us use the mean squared errors as the measure of accuracy for predicting Y using $\hat{m}(X)$.

```
mse <- function(obs, pred) {
  mean((obs-pred)^2)
}
```

Now, we create a test data and we train our model using different values of k and see which values of k produces the smallest MSE for our test data. We generate a test sample of size $m = 60$.

```r
set.seed(2)
test.x <-  runif(60,0,1)
e.test <-  rnorm(60,0,0.5)
test.y <- cos(10*test.x) + 4*test.x^3 -5*test.x+e.test
```

Now, we train our model using different different k values and calculate the training and test MSE's as follow:

```r
knn_reg_test_mse <- function(k, x.train, y.train, x.test, y.test){
  x.test <- data.frame(x.test)
  pred.values <- FNN::knn.reg(train= x.train,
                              test= x.test,
                              y= y.train,
                              k=k)$pred
  mse(y.test, pred.values)
}

knn_reg_train_mse <- function(k, x.train, y.train){
   x.grid <- data.frame(x.train)
   pred.values <- FNN::knn.reg(train= x.train,
                              test= x.grid,
                              y= y.train,
                              k=k)$pred
  mse(y.train, pred.values)
}

k <- 1:30

test_mse <- sapply(k,
                   knn_reg_test_mse,
                   x.train=x,
                   y.train=y,
                   x.test=test.x,
                   y.test = test.y)

train_mse <- sapply(k,
                    knn_reg_train_mse,
                    x.train=x,
                    y.train=y)


y_lim <- c(min(test_mse, train_mse), max(test_mse, train_mse))

par(mfrow=c(1, 2))

plot(k, test_mse, type="b", ylim=y_lim,  pch=20, col="red")
points(k, train_mse, type="b", pch=18, lty=2,  col="green" )
legend("topleft", col=c("green", "red"),
       legend=c("Training Error", "Test Error"),
       pch=c(5, 20))

plot(1/k, test_mse, type="b", ylim=y_lim,  pch=20, col="red")
points(1/k, train_mse, type="b", pch=18, lty=2,  col="green" )
legend("topright", col=c("green", "red"),
       legend=c("Training Error", "Test Error"),
       pch=c(5, 20))
```

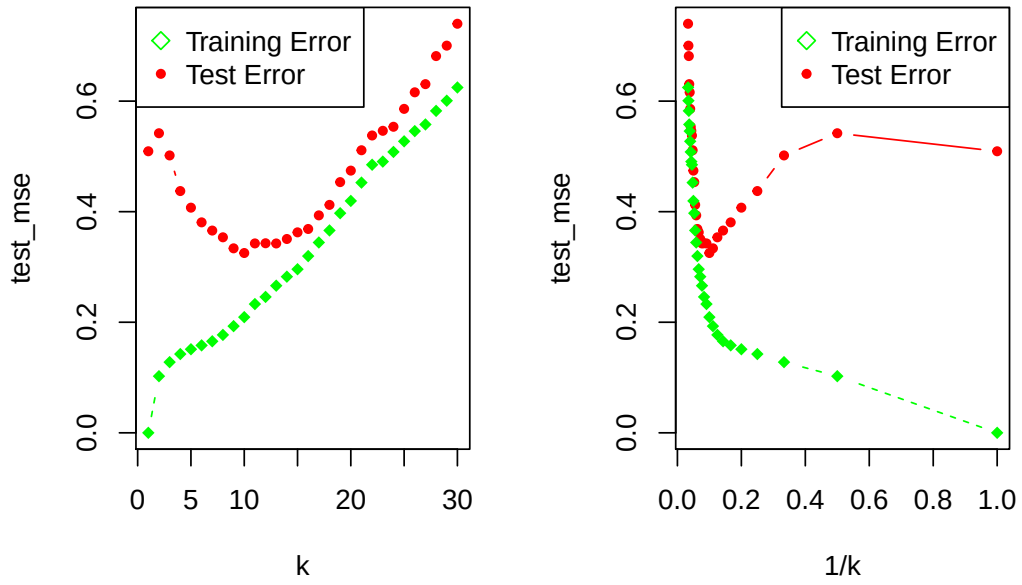


Figure 5: Training and test error estimates as a function of k and 1/k in our kNN regression problem

As we can see the minimum test error happens when \$k=\$10, resulting in an MSE of 0.3252452.

4.2 kNN regression is prone to curse of dimensionality

kNN estimators are universally consistent, which means

$$\mathbb{E}(\widehat{m}_k - m)^2 \longrightarrow 0 \quad \text{as } n \longrightarrow \infty,$$

with no assumption other than $\mathbb{E}(Y^2) < \infty$ provided we take $k = k_n$ such that $k_n \longrightarrow \infty$ and $\frac{k_n}{n} \longrightarrow 0$. For example, one can take $k_n = \sqrt{n}$. However, kNN regression estimators or in general any kNN estimator (in both regression and classification context) could fail if the dimension of the input space is high. This is simply because the nearest neighbors need to be close to the target point and can result in large errors. Assuming that the underlying regression function $m(\mathbf{x})$ in Lipschitz continuous, that is continuous and there exists a non-negative M such that

$$|m(\mathbf{x}_r) - m(\mathbf{x}_s)| \leq M \ ||\mathbf{x}_r - \mathbf{x}_s||_2,$$

the kNN estimate with $K \propto n^{-\frac{2}{2+p}}$ satisfies

$$\mathbb{E}\big(\widehat{m}_k(\mathbf{x}) - m(\mathbf{x})\big)^2 \leq n^{-\frac{2}{2+p}}.$$

Note that this a rather poor error rate as the it depends on the dimension p in a poor way. Given a small $\epsilon > 0$, if one want to have $\mathbb{E}\big(\widehat{m}_k(\mathbf{x}) - m(\mathbf{x})\big)^2 \leq \epsilon$ then one needs to choose sample n such that $n^{-\frac{2}{2+p}} \leq \epsilon$ or

$$n \geq \epsilon^{-\frac{2+p}{2}},$$

which increases exponentially with p. This problem appears to be true for many other problems and it is known as *curse of dimensionality*. In our kNN regression problem, suppose one want to make sure the error is less that $\epsilon = 0.1$ or $\epsilon = 0.05$, the sample size needed for this as a function of the feature dimension is as below:

```
e.bound<-function(e, p){e^(-(p/2+1))}
p<- c(1:10)

plot(p, e.bound(0.1, p), type="l", ylab="Sample Size", col="green", lwd=2)
points(p, e.bound(0.05, p),  type="l", col="red")
legend("topleft", legend=c(expression(epsilon ~"="~0.05),
                           expression(epsilon ~"="~0.1)),
       lwd=c(2, 3), col=2:3)
```

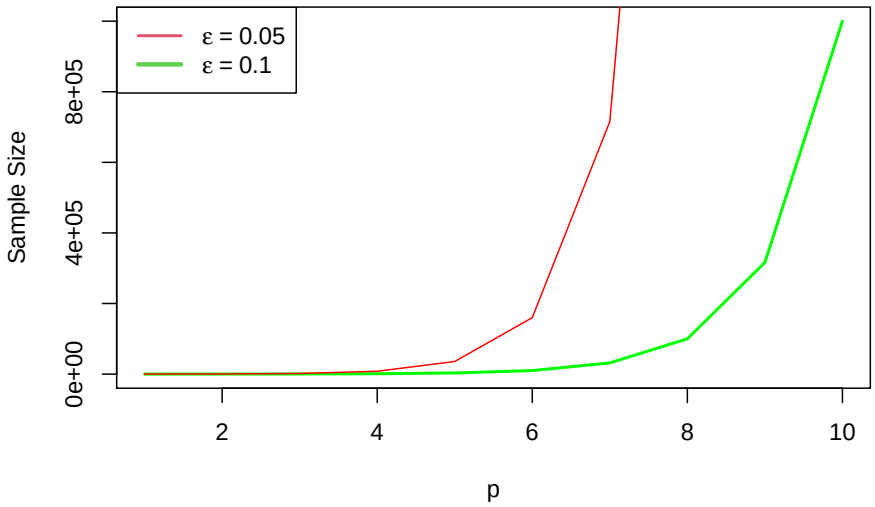


Figure 6: Depicting curse of dimensionality in a kNN regression analysis

To overcome this curse of dimensionality one needs to make more assumptions about the underlying problem in higher dimensions.

4.3 kNN for classification

kNN is very successful in classification when decision boundaries are very irregular. The idea behind kNN for classification is based on the majority vote among observations that are close to the point we are interested in. For $(\mathbf{X}, Y) \in \mathbb{R}^p \times \{0, 1\}$, the regression function $m(\mathbf{x})$ reduces to

$$m(\mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}] = \mathbb{P}(Y = 1|\mathbf{X} = \mathbf{x}),$$

which is the conditional probability of observing class 1, given $\mathbf{X} = \mathbf{x}$. Given i.i.d. observations (i.e., our training data) $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ where $(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \{0, 1\}$, $i = 1, \ldots, n$, that have the same joint distribution as $(\mathbf{X}, Y)$ we can use any nonparametric regression method to form an estimate $\widehat{m}_k(\mathbf{x})$ of $m(\mathbf{x})$, and then define a plug-in classifier via

$$\widehat{C}_k(\mathbf{x}) = \begin{cases} 1 & \text{if } \widehat{m}_k(\mathbf{x}) \geq \frac{1}{2}, \\ 0 & \text{if } \widehat{m}_k(\mathbf{x}) < \frac{1}{2}. \end{cases}$$

As we show later, this estimates the optimal classification rule, called the Bayes classifier given by

$$C_B(\mathbf{x}) = \begin{cases} 1 & \text{if } m(\mathbf{x}) \geq \frac{1}{2}, \\ 0 & \text{if } m(\mathbf{x}) < \frac{1}{2}. \end{cases}$$

The plug-in estimator when we estimate $m(\mathbf{x})$ using the kNN regression is called kNN classifier. Here the regression estimate is

$$\widehat{m}_k(\mathbf{x}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_0)} y_i, \quad \text{with} \quad y_i \in \{0, 1\}.$$

So, $\widehat{C}_k(\mathbf{x}) = 1$ is and only if more than half of the neighbors $\mathcal{N}_k(\mathbf{x})$ of $\mathbf{x}$ are of class 1. In other words, the classifier votes based on the membership of the neighboring points to decide the predicted class.

To better study kNN classification, we use a data set shown in the following graph. To generate the data set, first we generated 10 means μ_k from a bivariate normal distribution

$$N_2((0, 1)^\top, \mathbb{I}_2),$$

and label them BLUE (coded 0). Similarly, 10 more data points were generated and labeled ORANGE (coded 1). Then, for each class we generated 100 observations. This is done by picking an μ_k at random with probability 1/10 and then generating 100 observations from

$$N_2((0,1)^\top, 0.2\mathbb{I}_2).$$

thus leading to a mixture of Gaussian clusters for each class. We have also generated a test data of size $m = 200$ from the same population that is shown below and will be used later for the comparison of the training and test errors as functions of k.

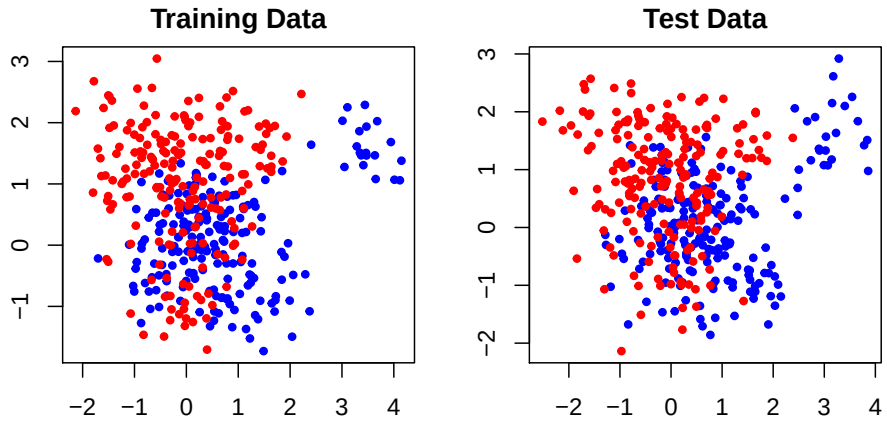
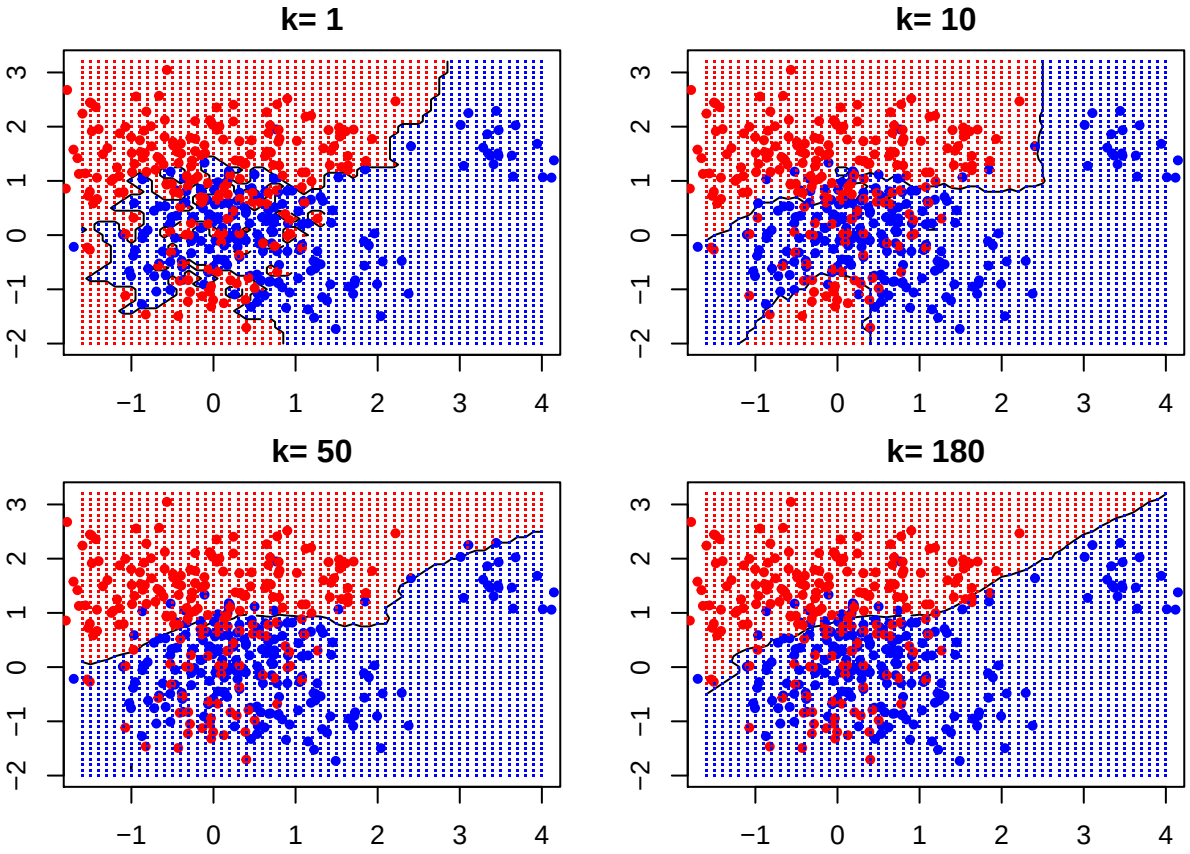


Figure 7: Simulated Training and Test data to bes used for kNN classification

One can easily observe that kNN regions are very irregular and respond to local clusters where one class is dominant. Below we show the performance of kNN classifiers for different values of k. As you can see by increasing k the bias of the classifier increases however the variance decreases with k.



One can compare the performance of kNN classifiers with different values of k. For classification problems when $Y \in \{0,1\}$ we measure the performance of a classifier using a 0-1 loss function rather than a squared error loss. In

this case, the risk of a classifier $C(\mathbf{X})$ is defined as $\mathbb{P}(Y \neq C(\mathbf{X}))$. For a given k, and after finding the corresponding kNN classifier, for each $(\mathbf{X}_j, Y_j)$ in the test data we first calculate $\hat{Y}_j = \widehat{C}_k(\mathbf{X}_j)$ and then estimate the test error as

$$\text{Test Error}_k \approx \frac{1}{m}\sum_{j=1}^{m} \mathbb{I}(\widehat{C}_k(\mathbf{X}_j) \neq Y_j),$$

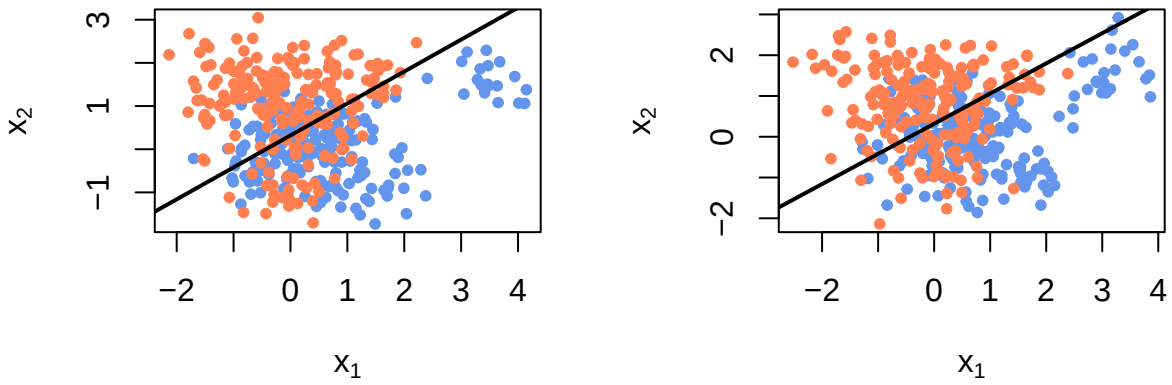
where m is the size of the test points.

There is a very famous result that shows the asymptotic probability of the 1-nearest neighbor classifier is no more than twice the Bayes error (the best possible error). A proof of this will be provided later in a chapter pertinent to classification.

As $\mathbb{P}(Y = 1|\mathbf{X} = \mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$, one might also want to use a linear regression to first fit a line between Y and $\mathbf{X}$, say $\widehat{m}_L(\mathbf{x})$ and then use a classifier

$$\widehat{C}_L(\mathbf{x}) = \begin{cases} 0 & \text{if } \widehat{m}_L(\mathbf{x}) \geq \frac{1}{2}, \\ 1 & \text{if } \widehat{m}_L(\mathbf{x}) < \frac{1}{2}. \end{cases}.$$

The following graph shows the decision boundary of a linear classifier on our training and test data sets.



To this end, using a linear classifier, one can easily see that the mean squared error (Training Error) is given by 0.26 while the mean of the prediction error for an independent sample of size $m = 200$ is given by 0.27.

5 Bias-Variance Trade-off

In practice choosing the k in models we discussed in this section is very important. Consider a kNN regression problem where $Y = m(\mathbf{X}) + \epsilon$ with $\mathbb{E}(\epsilon) = 0$ and $Var(\epsilon) = \sigma^2$. Using the training data points one obtains $\widehat{Y}_i = \widehat{m}(\mathbf{X}_i)$ where

$$\widehat{m}_k(\mathbf{X}_i) = \frac{1}{k} \sum_{j \in \mathcal{N}_k(\mathbf{X}_i)} y_j.$$

This results in a residual sum of squares $SSE(k) = \sum_{i=1}^{n} (Y_i - \widehat{Y}_i)^2$. It is easy to see that we can not use SSE on the training data points to determine k since we would always pick $k = 1$ that results in $SSE(1) = 0$. Such a model is unlikely to predict future data well at all. To choose k one need to be worried about the performance of the estimator on new data points that have not been seen by the model and not been used to train the kNN estimator. To this end, one needs to use expected prediction error (EPE) or simply test error to choose k. The expected prediction error at $\mathbf{x}_0$ also known as *test* or *generalization* error is given by

$$EPE(\mathbf{x}_0) = \mathbb{E}[(Y - \widehat{m}(\mathbf{x}_0))^2 | \mathbf{X} = \mathbf{x}_0].$$

It is easy to see that, $EPE(\mathbf{x}_0)$ can be decomposed as follow

$$\begin{aligned} EPE(\mathbf{x}_0) &= \mathbb{E}[(Y - \widehat{m}(\mathbf{x}_0))^2 | \mathbf{X} = \mathbf{x}_0] \\ &= \sigma^2 + \{Bias^2(\widehat{m}_k(\mathbf{x}_0)) + Var_{\mathcal{T}}(\widehat{m}_k(\mathbf{x}_0))\} \\ &= \sigma^2 + \left(m(\mathbf{x}_0) - \frac{1}{k} \sum_{j \in \mathcal{N}_k(\mathbf{x}_0)} m(\mathbf{x}_j) \right)^2 + \frac{\sigma^2}{k} \end{aligned}$$

Here there are three terms that are contributing to the generalization or test error at point $\mathbf{x}_0$:

1. σ^2 which is irreducible error which is essentially the variance of the new test point. This is beyond our control and is inevitable even if we knew the true $m(\cdot)$.

2. The Bias term that accounts for the squared difference between the true $m(\mathbf{x}_0)$ and the expected value of the estimate.

$$\begin{aligned} Bias(\widehat{m}_k(\mathbf{x}_0)) &= \mathbb{E}[\widehat{m}_k(\mathbf{x}_0)] - m(\mathbf{x}_0) \\ &= \frac{1}{k} \sum_{j \in \mathcal{N}_k(\mathbf{x}_0)} \mathbb{E}[Y_j] - m(\mathbf{x}_0) \\ &= \left(\frac{1}{k} \sum_{j \in \mathcal{N}_k(\mathbf{x}_0)} m(\mathbf{x}_j) \right) - m(\mathbf{x}_0) \end{aligned}$$

   This term most likely increases with k if the true m is smooth. For small k, and for lower dimensions, $m(\mathbf{x}_j)$ for those $\mathbf{x}_j$ in the neighborhood of $\mathbf{x}_0$ will most likely be close to $m(\mathbf{x}_0)$ and so on average the Bias will be small. As k grows, the neighbors are further away from the target and their corresponding m values will be far from $m(\mathbf{x}_0)$ and the Bias will increase. Note that here we assume m is smooth, otherwise the above statements could be wrong.

3. The variance is simply the variance of the average of k (i.i.d) points close to the target, which is $\frac{\sigma^2}{k}$ and it decreases with k.

So, as we observe varying k will have effect on both the Bias and Variance and not the irreducible error. This is called *bias-variance* trade-off in the literature and has an important role in selecting the right model for the underlying problem of interest. Often as the complexity of the model grows, the bias decreases and the variance increases.

However, complex models are often hard to understand and interpret. So, there is always a battle between model complexity and its interpretability and this needs to be taken into account in practical situations.

Typically we would like to choose k (in general the complexity of our model) such that it provides a good trade-off between the bias and variance by minimizing the test error. An obvious estimator of the test error is the training error. Unfortunately, training error always underestimates the test error and does not account for the complexity of the model. There are several ways to estimate the test error which we will discuss some of them in this course including the famous cross validation technique.

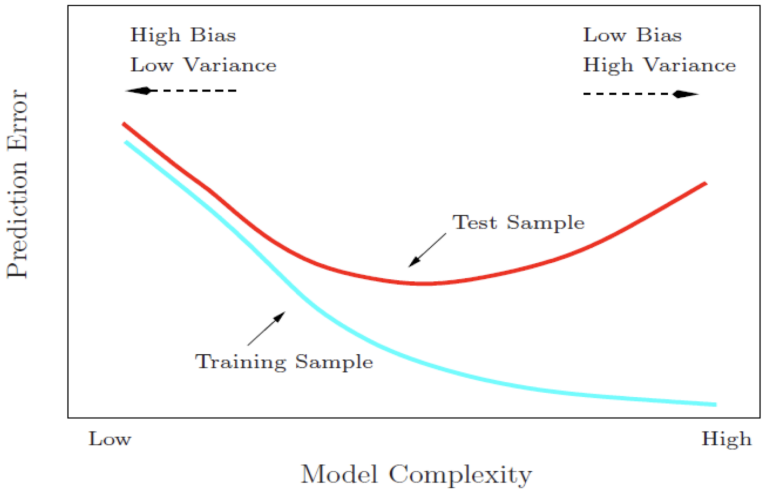


Figure 8: Figure: Training Error Versus Test Error and the Bias-Variance Tradeoff (Source: Elements of Statistical Learning)

Now, we look at the problem of bias-variance trade-off in a more general setting.

6 Reducible and Irreducible Error

Suppose we are given a training data set

$$\mathcal{T} = \{(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, i = 1, \ldots, n\},$$

and the ultimate goal is to estimate $m(\cdot)$ given in (2). Suppose we have an estimate of m given by $\hat{m}(\cdot)$. Suppose $\mathbf{x}$ and y is a new data point independent of our training sample $\mathcal{T}$. We would like to find $\hat{m}(\mathbf{x})$ that is a good estimate of the regression function m, hence predicting Y accurately. We define the **expected prediction error** of predicting Y using $\hat{m}(\mathbf{X})$. A good $\hat{m}$ will have a low expected prediction error.

$$\text{EPE}\left(Y, \hat{m}(\mathbf{X})\right) = \mathbb{E}_{\mathbf{X},Y,\mathcal{T}}\left[\left(Y - \hat{m}(\mathbf{X})\right)^2\right]$$

This expectation is over $\mathbf{X}$, Y, and also $\mathcal{T}$, as $\hat{m}$ is actually random depending on the sampled data *tau*. Conditioning on $\mathbf{X} = \mathbf{x}$, one can write

$$\begin{aligned}
\text{EPE}\left(Y, \hat{m}(\mathbf{x})\right) &= \mathbb{E}_{Y|\mathbf{X},\mathcal{T}}\left[\left(Y - \hat{m}(\mathbf{X})\right)^2 \mid \mathbf{X} = \mathbf{x}\right] \\
&= \underbrace{\mathbb{E}_{\mathcal{T}}\left[\left(m(\mathbf{x}) - \hat{m}(\mathbf{x})\right)^2\right]}_{\text{reducible error}} + \underbrace{\sigma^2}_{\text{irreducible error}}.
\end{aligned}$$

Note that the **reducible error**, is simply the mean squared error of estimating $m(\mathbf{x})$ with $\hat{m}(\mathbf{x})$. The only thing that is random here is $\mathcal{T}$, the data that was used to obtain $\hat{m}$.

$$\begin{aligned}\text{MSE}\left(m(\mathbf{x}), \hat{m}(\mathbf{x})\right) &= \mathbb{E}_{\mathcal{T}}\left[\left(m(\mathbf{x}) - \hat{m}(\mathbf{x})\right)^2\right] \\ &= \underbrace{\left(m(\mathbf{x}) - \mathbb{E}\left[\hat{m}(\mathbf{x})\right]\right)^2}_{\text{bias}^2(\hat{m}(x))} + \underbrace{\mathbb{E}\left[\left(\hat{m}(\mathbf{x}) - \mathbb{E}\left[\hat{m}(\mathbf{x})\right]\right)^2\right]}_{\text{var}(\hat{m}(\mathbf{x}))}.\end{aligned}$$

Also, the **irreducible error** is the the variance of Y given that $\mathbf{X} = \mathbf{x}$, which is the noise that can not be learned by the model itself, even if we knew the tru m, we will still have this error when we use m to predict the value of Y at the point $\mathbf{x}$.

In some situations we might be able to find $\hat{m}$ which is **unbiased**, that is $\text{bias}\left(\hat{m}(\mathbf{x})\right) = 0$, which also has low variance. However, in many cases, this is not possible and there is always a **bias-variance tradeoff**. In many cases, more complex models are less biased but they have large variances. On the other hand, simple modes are mostly biased but they have a low variance. So, choosing the right complexity for the underlying model is a challenging problem.

6.1 Simulation studies

Let us perform some simulation studies to understand the bias-variance trade-off, through simulation. Suppose we would like to train a model to learn the true regression function function

$$m(x) = \cos(10x) + 4x^3 - 5x,$$

using training data that are generated from

$$y = m(x) + \epsilon \quad \text{with} \quad \epsilon \sim N(0, 0.5).$$

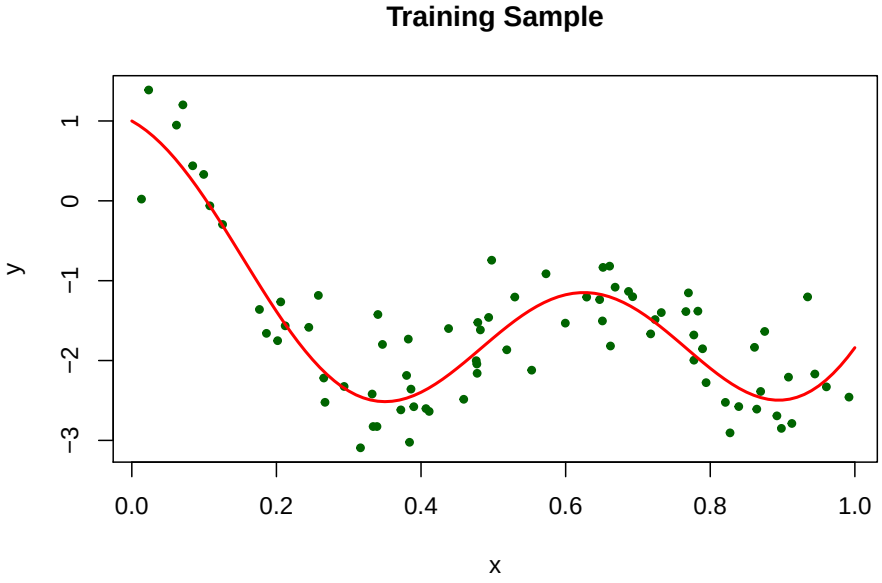
We would like to predict Y given $X = x$, using obtained $\hat{m}(x)$. To this end, we first generate (training and test) samples from the underlying model using the following code:

```
m = function(x) {
cos(10*x) + 4 *x^3 - 5*x
  }

sim_data = function(m, sample_size = 80) {
  x = runif(n = sample_size, min = 0, max = 1)
  eps = rnorm(n = sample_size, mean = 0, sd = 0.5)
  y = m(x) + eps
  data.frame(x, y)
}
```

The scatter plot of our training data is given by

```
set.seed(1)
train_data <- sim_data(m, 80)
plot(train_data, col="darkgreen", main="Training Sample", pch=20)
curve(m, from=0, to =1, col="red", add=TRUE, lwd=2)
```

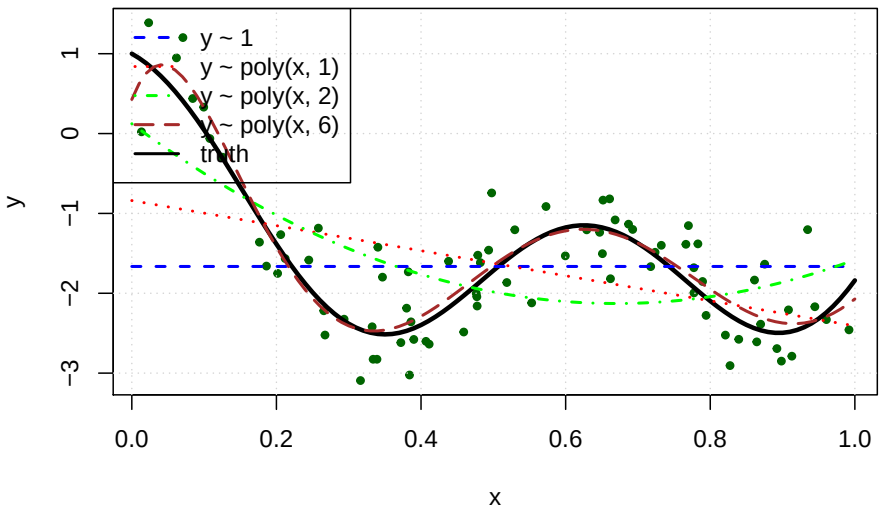


We consider 4 polynomial models to estimate the underlying m:

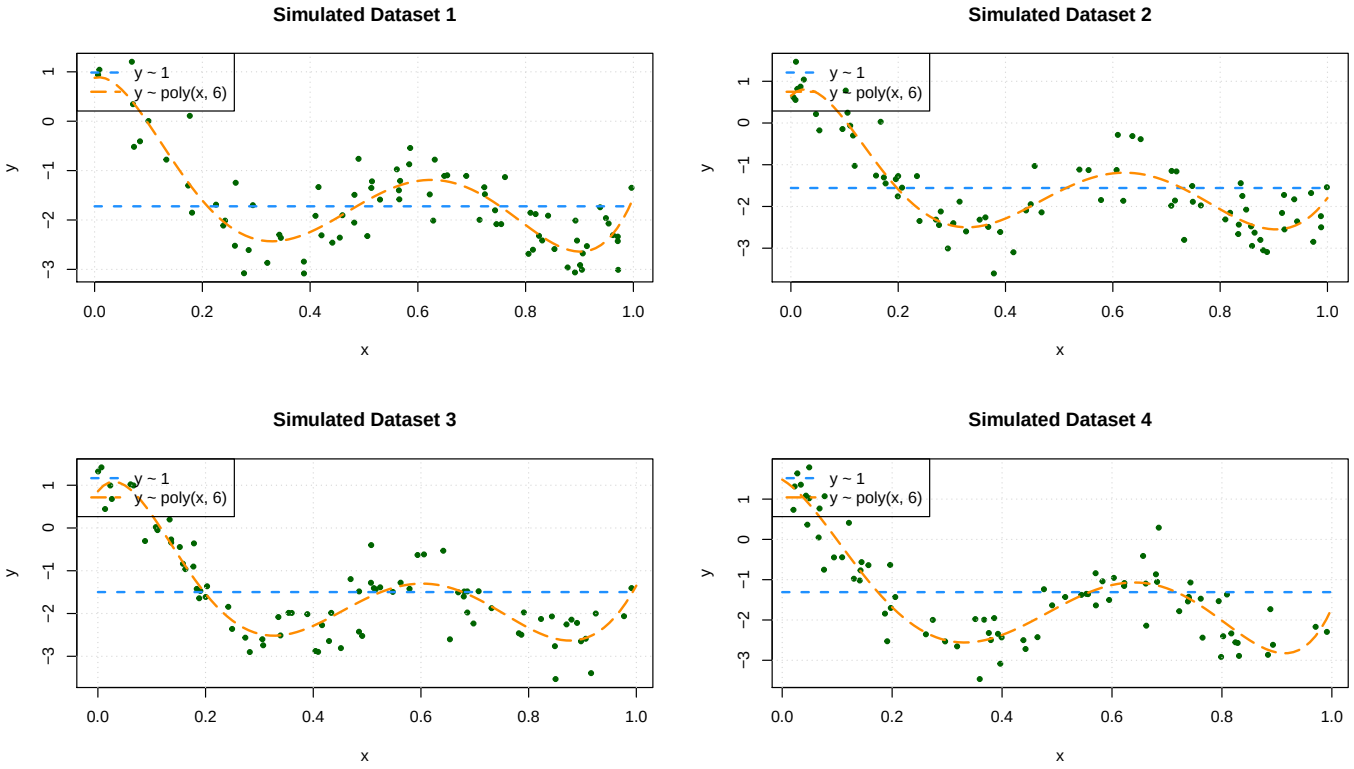
$$\hat{m}_0(x) = \hat{\beta}_0$$
$$\hat{m}_1(x) = \hat{\beta}_0 + \hat{\beta}_1 x$$
$$\hat{m}_2(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2$$
$$\hat{m}_6(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \ldots + \hat{\beta}_6 x^6$$

```
fit_0 = lm(y ~ 1,                   data = train_data)
fit_1 = lm(y ~ poly(x, degree = 1), data = train_data)
fit_2 = lm(y ~ poly(x, degree = 2), data = train_data)
fit_6 = lm(y ~ poly(x, degree = 6), data = train_data)
```

Plotting these four trained models, we see that the zero predictor model does very poorly. The first degree model is reasonable, but we can see that the second degree model fits much better. The ninth degree model seem rather wild.



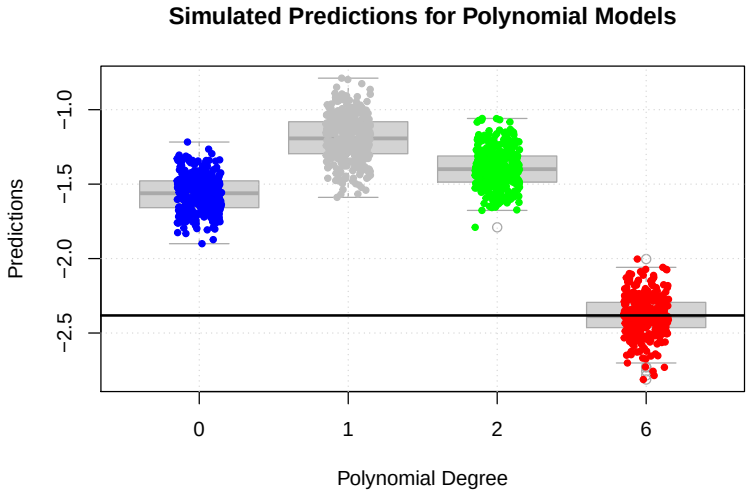
As we see $\hat{m}_0$ is very biased but $\hat{m}_6$ seems to be less biases. Now, let us see, how variable these models are. To this end, we generate 4 different data sets and fit these models. The following figure show the $\hat{m}_0(x)$ and $\hat{m}_6(x)$ fitted to these data sets.



These plots make it clear the difference between the bias and variance of these two models. As we observe $\hat{m}_0$ is clearly biased but nearly the same for each of the data sets, since it has very low variance.

One the other hand $\hat{m}_6$ seem to be unbiased on average, however it is very variable as on each data set $\hat{m}_6$ results in a very different fitted model.

To end up this chapter, we perform a simulation study to understand the relationship between the bias, variance, and MSE for the estimates for $m(x)$ given by these four models at the point $x = 0.3$.



The above plot shows the predictions for each of the 350 simulations of each of the four models of different polynomial degrees. The truth, $m(x = 0.3) = -2.3819925$, is given by the solid black horizontal line.

Two things are immediately clear:

- As complexity *increases*, **bias decreases**. (The mean of a model's predictions is closer to the truth.)
- As complexity *increases*, **variance increases**. (The variance about the mean of a model's predictions increases.)