Fitting & Assessing Models

Arnab Maity - Modified by Justin Post

Code to read in packages used.

library(MASS)  
library(klaR)  
library(tufte)  
library(tidyverse)  
library(caret)  
library(rsample)  
library(ISLR2)  
library(knitr)  
library(AppliedPredictiveModeling)  
library(kableExtra)  
# invalidate cache when the tufte version changes  
options(htmltools.dir.version = FALSE)

# Introduction

In this chapter, we will go into details about training statistical learning models. In the process, we will learn about different methods for splitting the data and resampling techniques, process of tuning hyperparameters, tradeoff between bias and variance, and various criteria for evaluating model performance.

The process of building a statistical model (or multiple models) roughly has the following steps.

1. Split the data into a *training set* and a *test set*.
2. *Tune hyperparameters* (of all the models under consideration) using the training set:
   1. Split the training set further into two sets: one for fitting the model (a *new training set*), and the other for evaluating model performance (known as *validation set* or *holdout set*).
   2. For each candidate value of hyperparameter(s), fit the model using the new training set, and evaluate the fitted model using the validation set using a metric of our choice.
   3. Typically, we repeat stpes a. and b. few times so that we get repeated measurements of model performance for each value of hyperparameters. Using a single validation set often provides highly variable estimate of model performance. The final model performance is taken to be the average of these multiple measurements.
   4. Choose the best value of hyperparameters by optimizing the model perfromance measure obtained in step ‘c’. We would maximize or minimize the model performance criterion depending on the situation. For example, we would minimize criterion like “prediction error”, but maximize “classification accuracy”.
3. Using the best value of hyperparameters, fit the model(s) on the entire training set and estimate the model parameters. This is (are) the *final model(s)* chosen in using the training set.
4. Use the test set to estimate the model performance of the final model(s) from step 3.
5. Again, we may want to repeat steps 1. – 4. a few times to get a reliable estimate of model performance of the final models. For example, we can use cross-validation here to incorporate the uncertainty due to hyperparameter tuning as well.

Notes:

* Most model evaluation criteria focus on prediction. The steps described are geared towards building of predictive models. The fitted model may not be easily interpreted or lend itself to inference.
* There are two points in the algorithm above that we may want to perform repeatedly: these are the inner and outer loops.
* Depending on the situation (sample size, computational cost), we can use any of the resampling and data splitting methods in each of the loops.

# Non-parametric Model: -Nearest Neighbors Regression

One of the simplest nonparametric regression methods is the -Nearest Neighbors (KNN) regression. We will develop our ideas further based on this regression technique. However, these ideas will be applicable in other cases as well.

Assume that we have a regression task and are using our usual, generic, model:

where is an unknown function, and are zero mean random errors with , and independent of . Suppose we have training data . Then, for any given value , KNN regression estimates as follows:

* Identify the observations in the training data such the their values are “nearest” to .
* Estimate by the average values of the points obtained in the previous step.

Formally, suppose denotes the indices of the points whose values are nearest to . Then we have

Note that the predictor can be a scalar as well as a vector, as long as there is a measure of “nearness” available.

## Distance metric

We determine points nearest to by computing a distance metric between and the values of the training data, and taking the points with smallest distance measures.

The most common distance metric is the Euclidean distance: for two vectors and , the Euclidean distance is

This is also known as the -norm of , that is, .

Another popular distance metric is the -norm, , that is,

|  |
| --- |
| L2-norm vs. L1-norm. Given two points (black dots, one located at (0,0) and one located at (1,1)), the L2-norm measures the distance of the straight line between these points. In contrast, L1-norm measures the distance of the path that can only go parallel to the x- and y-axes. Here that is a line going from (0,0) to (1,0) and another line going from (1,0) to (1,1).  L2-norm vs. L1-norm. Given two points (black dots), the L2-norm measures the distance of the straight line between these points (dashed line). In contrast, L1-norm measures the distance of the path that can only go parallel to the x- and y-axes (dotted line). |

The -norm is used when we suspect the data might have outliers or one coordinate may have large values compared to others. This is also useful for binary predictors. The distance is also known as “taxicab” and “Manhattan” distance. The geometry of these distance metrics are shown (simplified) in Figure .

There are other types of distance metrics in literature such as Minkowski, Mahalanobis, Hamming, Cosine distances and so on.

## The hyperparameter

The value of (the number of neighbors) must be chosen using the data. Most often, we do a resampling technique to do so. We’ll discuss this more shortly. For now, let’s look at fitting this model in R.

### Boston Dataset

Let us consider the Boston dataset in the ISLR2 package. The data set contains housing values of suburbs of Boston.

* Goal: Predict the median value of owner occupied homes (in $1000’s, the medv variable) using the lower status of the population (percent, lstat variable).
* A plot of the data is shown in Figure .

|  |
| --- |
| The image is a scatter plot with data points distributed across a two-dimensional grid. The x-axis is labeled 'Lower status of the population (percent),' ranging from 0 to 40, while the y-axis is labeled 'Median value of owner-occupied homes,' ranging from 0 to 50. The plot illustrates a negative correlation between the two variables, with data points densely concentrated at higher median home values with lower population status percentages, and values decreasing as population status percentage increases. The data points are black dots scattered across the plot.  Plot of median housing value vs. percent of population with lower status from Boston data. |

* A snapshot of the R tibble is shown below with only the two variables of interest.

| medv | lstat |
| --- | --- |
| 24.0 | 4.98 |
| 21.6 | 9.14 |
| 34.7 | 4.03 |
| 33.4 | 2.94 |
| 36.2 | 5.33 |

Let us see a KNN fit to the data, with . Here we are not training/testing the model yet – we are simply attempting to understand the role of the hyperparameter and its impact on the fitted model. We can use the function kknn() in the kknn library.

* We use formula notation to specify our response and predictor variable relationship
  + y ~ x

library(kknn)  
knn\_fit <- kknn(medv ~ lstat,   
 train = Boston,   
 test = Boston, k = 30) #fit the model

* Now that we’ve *fit*, *trained*, or *estimated* our model (three ways of saying the same thing!), we can find the predictions from the model for our training observations and add that as a column to a tibble.

#create the predicted values as a new column and append them to our data set  
knn\_estimates <- mutate(Boston, knn\_est = fitted(knn\_fit)) |>  
 arrange(knn\_est)  
knn\_estimates |>   
 select(medv, lstat, knn\_est) |>  
 slice(1:5) |>  
 kable()

| medv | lstat | knn\_est |
| --- | --- | --- |
| 6.3 | 29.97 | 11.49992 |
| 16.5 | 29.93 | 11.50949 |
| 7.4 | 31.99 | 11.54255 |
| 8.1 | 29.68 | 11.55495 |
| 5.0 | 30.59 | 11.55656 |

* Let’s plot the fit!

#plot the data along with the predictions found via knn  
knn\_estimates |>  
 ggplot(aes(x = lstat, y = medv)) +  
 geom\_point(size = 0.5) +  
 geom\_line(aes(y = knn\_est), color = "blue", linewidth = 2)

|  |
| --- |
| The image is a scatter plot with data points distributed across a two-dimensional grid. The x-axis is labeled 'Lower status of the population (percent),' ranging from 0 to 40, while the y-axis is labeled 'Median value of owner-occupied homes,' ranging from 0 to 50. The plot illustrates a negative correlation between the two variables, with data points densely concentrated at higher median home values with lower population status percentages, and values decreasing as population status percentage increases. The data points are black dots scattered across the plot. A blue curve is overlayed. This curve flows roughly through the middle of the data points with some variability evident for lstat values between 5 and 10 but otherwise, pretty smooth. |

**Important** How should we choose ? Consider the fits using varying (1, 30, or 300) in Figure .

|  |
| --- |
| The image consists of three side-by-side scatter plots illustrating the relationship between the median value of owner-occupied homes and the lower status of the population, measured as a percentage. Each plot displays data points as gray dots with a colored line indicating a smoothing or regression fit. The x-axis represents the lower status percentage, while the y-axis represents the median home value. The first plot on the left, labeled 'K = 1,' shows a red line that varies wildy with the data, the second plot in the middle, labeled 'K = 30,' features a green line that is somewhat variable but mostly stays through the middle of the data points, and the third plot on the right, labeled 'K = 300,' presents a blue line that is very smooth but does a poor job representing the middle of the data. The plots generally depict a downward trend, where higher lower status percentages correspond to lower median home values.  Estimated functions form Boston data example for different values of K. |

* We note that for small value of , KNN produces extremely rough estimate of . We are almost interpolating the data – this is an example of **overfitting** the data. While the model is most flexible, and the estimated function does capture the shape of the data (perhaps too much so), such a fit is undesirable as the estimate is much too volatile.
* In contrast, for large value – this is of our sample size – the estimate is smooth, but does not capture the shape of the data. Such a model is not flexible, and undesirable as it may produce a **biased** estimate of , and inaccurate predictions.
* For , it seems the model is flexible enough to capture the overall shape of the data, but stable enough to not overfit the data. Thus we need to discuss a criterion that evaluates the quality of model fit, and enables us to choose (hyperparameters in a regression model in general) properly.

## Regression model evaluation criterion

We evaluate regression models based on how well they predict new observations. Suppose we have new predictor value , and want to predict the response corresponding to . The (squared) prediction error is . However, we want the procedure to provide good predictions across all possible values of when , so we might want to choose a model by minimizing expected prediction error for ,

This strategy works if we are only interested in the specific value . In general, we want a procedure which can predict for all possible values of , not just one specific value. Thus the average performance of the procedure can be measured by taking “average” (expected value) of the previous expected prediction error over possible values of , that is,

The equality in the equation follows by law of iterative expectation: for random variables and , .

Unfortunately, the quantity above can not be directly computed without knowing the probability distribution underlying the data generating process, and hence needs to be estimated using a sample. Suppose we have training set , and a test set . Then, based on the test set, we can estimate the quantity above as

where we have replaced the expected value by a sample average, and the average is taken over the test set. This quantity is called the test *Mean Squared Error (MSE)*. Similar quantity can be computed using the training set as well.

* Consider using KNN regression with .
  + The training MSE is zero (or close to zero depending on how KNN handles ties in the values) since for each in the dataset, the nearest point of is itself. However, 1-NN regression might perform very poorly in a test dataset!
  + Typically, minimizing the training MSE would result in choosing the most flexible model, but *having a low training MSE does not ensure that the test MSE will be low* as well.
* Let’s manually compute the training MSE for our KNN model fit previously

mean((knn\_estimates$medv - knn\_estimates$knn\_est)^2)

[1] 25.19358

* If we used instead we get the training MSE value to be estimated at

knn\_fit\_k1 <- kknn(medv ~ lstat,   
 train = Boston,   
 test = Boston, k = 1) #fit the model  
#add the predicted values to the data set  
knn\_estimates\_k1 <- mutate(Boston, knn\_est = fitted(knn\_fit\_k1)) |>  
 arrange(knn\_est)  
#Find the training MSE  
mean((knn\_estimates\_k1$medv - knn\_estimates\_k1$knn\_est)^2)

[1] 7.064901

* Consider the plot below showing training and testing error for simulated data sets and varying values of .
  + Here we know the true form of the function , and thus can simulate data using it.
  + We can generate many data sets for training purposes and many for testing purposes.
  + We can then fit KNN regression model with different values of
  + Figure shows the results for one such experiment.
    - We see that the test MSE is generally higher that the training MSE.
    - Training MSE keeps increasing as increases (the procedure becomes less flexible).
    - However, the test MSE first decreases and then levels off before increasing slightly.
    - In this experiment, the minimum test MSE is produced for , while lowest training MSE is for .

|  |
| --- |
| The image is a line graph with two plotted datasets, one labeled as 'test' in red dots and the other as 'train' in blue triangles. The x-axis is labeled 'K' and ranges from 0 to 100, while the y-axis is labeled 'Mean Squared Error' ranging from 0 to 1.5. The graph shows two distinct lines with the test dataset starting at a higher error and slightly decreasing before leveling off, and the train dataset starting at a lower error and gradually increasing, stabilizing after a certain point. A legend at the top right identifies the red dotted line as 'test' and the blue triangular line as 'train'.  Training and test MSE for simulated data for different values of K. Larger values of K correspond to less flexibility. |

## Bias-Variance decomposition

To understand the shape of the test MSE curve, we further investigate the form of MSE.

* Recall that we started from the expected prediction error for the test data . Let’s try to ‘decompose’ this into terms corresponding to bias and variance.

The first term is a fixed parameter which we can not control. Even if we have a extremely accurate estimation procedure for so that , we would still have the expected prediction error to be . Thus the term is called the *irreducible error* – it is the variance of the target. We have that . Even if we know true , this variance remains.

The second term is under our control, and depends of the method of estimation of . Minimizing the expected prediction error is equivalent to minimizing the second term. This term can further be decomposed into two parts using similar calculations as above: Recall, that for a random variable , . Also, for an estimator of a parameter , .

Collecting all the terms, we have that Thus the expected prediction error is a combination of the variance and squared bias of the estimator .

* We can now see that the expected prediction error includes terms from a combination of the variance and squared bias of the estimator .
  + We again resort to a simulation experiment to see the relative contribution of the variance and squared bias of to the prediction error.
  + Figure shows one simulated training set of size along with the true function used to generate the data ().

|  |
| --- |
| The image is a scatter plot displayed on a gray background with a grid pattern. The horizontal axis is labeled "x" and ranges from 0.00 to 1.00, while the vertical axis is labeled "y" and ranges from -3 to 3. Multiple data points, represented as black dots, are scattered across the plot. The distribution of the dots shows a general upward trend from left to right, peaking near the center before descending slightly. A black curve, fitted through the data points, runs horizontally across the plot, illustrating this upward and then downward trend.  Simulated data of size n=500. |

* We generate multiple such training sets, and for each set we fit a KNN regression model with , and .
* The test set if a grid of 101 equally spaced points in and we compare the fits to the true curve.
* The estimated functions are shown in Figure .

|  |
| --- |
| Three line plots illustrating data smoothing with K=1 (red, noisy), K=30 (green, moderately smooth), and K=300 (blue, smooth).  Simulated data showing bias and variance of KNN fits. |

In general, this phenomenon holds for various regression models. More flexible models produce estimate with low bias but high variance. Less flexible models do the opposite – estimates have high bias but low variance. Minimizing test MSE tends to choose a model that balances between bias and variance.

### Other Model Metrics

We should be aware that test MSE is not the only metric one can use to evaluate a regression model. A few of the other evaluation metrics are shown below:

* *Root mean squared error (RMSE):* just the square root of MSE. Brings the MSE to the same using as the responses.
* *Mean absolute error (MAE):* average of absolute values of the prediction discrepancies,
* It is more robust the MSE in the sense that it does not emphasize large differences as MSE does.
* *Mean residual deviance:* generalizes the concept of MSE for generalized linear model fitted with maximum likelihood methods (e.g., Poisson and Logistic regression).
* : proportion of variance explained by the model.
* A nice property of is that it will be always between 0 and 1. values close to 0 indicate inadequate model fit, while values close to 1 indicate that the model explains a large amount of variability in the response.

# Data splitting

As we want to have a measure of prediction accuracy based on a test set, we don’t want to use the entire dataset for training and testing. We need a test data set that we can use to evaluate our model’s performance in general.

* We can the *holdout method* or resampling techniques such as *bootstrap* or *v-fold cross validation* to create test set(s) from our data, and validate our models’ performance.
* Let’s go through these ideas!

## Holdout method

The holdout method randomly splits a given dataset into two sets: one for training and one for evaluation (the holdout/validation/test set).

* In practice, , or splits are commonly used for training/test sets. Figure shows the basic layout of the holdout method. In general, we should keep in mind that putting too much data for training results in a small test set, which may not provide a good estimate of the model performance. On the other hand, putting too much data in the test set results in a small traing set, which results in poor model fitting. Other factors such as whether also may impact the split sizes.

|  |
| --- |
| The holdout method. The whole dataset is split into two parts: training and holdout sets.  The holdout method. The whole dataset is split into two parts: training and holdout sets. |

A simple way to create such a split is via *simple random sampling without replacement (SRSWOR)*, that is, by randomly choosing a subset of observations from the data set and putting them aside as the training set. The remaining observations form the holdout set.

* Consider the Boston data again. In base R, we can use the sample() function to perform SRSWOR, as follows.

# set a seed for reproducible results   
set.seed(1234567)  
  
# sample from the row indices to include in the test set  
n <- nrow(Boston)  
index <- sample(x = 1:n,   
 size = round(0.8\*n),  
 replace = FALSE)  
  
# Test and training sets  
train <- Boston[index, ]  
test <- Boston[-index, ]  
  
# Data dimensions  
dim(train)

[1] 405 13

dim(test)

[1] 101 13

* We have split the data – in the example above!
* We’ll look at other R packages that make this process even easier to do and will allow us to easily do stratified sampling and other more advanced splitting techniques.

Ideally, the distribution of in the test set will be similar to that in training set. Figure shows the corresponding distributions (estimated probability densities) of medv for the training/test sets.

|  |
| --- |
| The image is a line graph displaying estimated distributions with the title "Estimated Distributions of Our Response Var." The x-axis is labeled "medv," and the y-axis is labeled "density." The graph features two curves, one orange and one black, that illustrate different distributions. The orange line peaks higher than the black line. Both curves peak around the same x-axis range between 15 and 25 but with varying heights and shapes.  Estimate density functions for medv variable in training (orange) and test (black) sets as obtained using base R, caret, and rsample packages. |

* A disadvantage of SRSWOR is that it does not always preserve the distribution of the response variable.

*Stratified random sampling* is used to explicitly control aspects of the distribution of .

* Useful with data with small sample size or a skewed response distribution.
* Stratified random sampling strategy is to draw sample for each group (strata) of so that the test set represents the distribution of of the whole data.^

For a classification task, if extreme class imbalance is present in the response (say “No” and only “Yes”), we might choose to over-sample the rare class, or under-sample the abundant class, or a combination of both the strategies can be employed. A popular technique in this regard is *Synthetic Minority Over-sampling Technique (SMOTE)*, (N. Chawla et al. SMOTE: Synthetic minority over-sampling technique J. Artif. Intell. Res. (2002). See also, Dina Elreedy, Amir F. Atiya, A Comprehensive Analysis of Synthetic Minority Oversampling Technique (SMOTE) for handling class imbalance, Information Sciences, Volume 505, 2019). which generates synthetic samples from the rare class.

* In particular, SMOTE takes a random observation from the rare class and then finds its nearest neighbors in the rare class.
* Then SMOTE generate new samples using the convex combinations of the original randomly selected observation and one of the the nearest neighbors.
* Many packages allow for SMOTE implementation as a possible sampling strategy.
* The authors of SMOTE also suggest that a combination of SMOTE and under-sampling the majority class works better than just using SMOTE.

Let us now investigate the holdout method using the Boston data.

* First, let’s fit the model with using just the training data

knn\_train\_fit <- kknn(medv ~ lstat,   
 train = train,   
 test = train, k = 30) #fit the model

* Now compute the training set MSE

#training MSE  
train\_MSE <- mean((train$medv - knn\_train\_fit$fitted.values)^2)  
train\_MSE

[1] 23.88902

* Fit again but test on the testing set to obtain the test set MSE

knn\_train\_fit <- kknn(medv ~ lstat,   
 train = train,   
 test = test, k = 30) #fit the model  
#test MSE  
test\_MSE <- mean((test$medv - knn\_train\_fit$fitted.values)^2)  
test\_MSE

[1] 33.26467

* Notice the test set MSE is much larger! It is as we expected – training MSE most likely underestimates the prediction error, while test MSE can be viewed as a reasonable estimate.
  + *It is important to remember that we are operating with the setting - the test MSE might not reflect the best performance the model can have.*

### Selecting an Optimal Tuning Parameter Using Only a Holdout Set

Now let us address the question about choosing the optimal , that is, the value of that gives the best general performance. One option is to use only a holdout set to do so. Once we chose an optimal value of , the model with that would then be fit to the whole data.

Our steps would be:

* Split the data into training and test sets
* For each candidate value of , fit the model in the training set, and compute MSE using the test set.
* Choose the which gives minimum test MSE.
* Fit KNN with optimal K to the full data set.

Then the fully trained model can be used for future predictions. Let’s do this manually for now!

* We’ve already split our data into a train and test set (both are objects in our R environment).
* Now we’ll create a grid of values to fit the model on.

kgrid <- c(1:100)

* Next, we’ll use the lapply() function to apply the kknn() function to each value of our kgrid object.
  + lapply() is a function that allows us to apply a function to a list (or vector)
  + This is like a nicer way of doing a for loop
  + Here we’ll write a quick anonymous (or lambda) function to take the list elements and return the KNN fit

test\_preds <- lapply(X = kgrid,   
 FUN = function(x){  
 kknn(medv ~ lstat,   
 train = train,  
 test = test,  
 k = x)  
 })

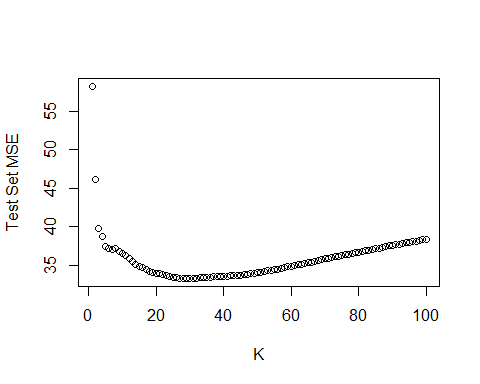
* A list of fitted objects is returned.
* We can apply an MSE calculation to each of these list elements!
  + Here we’ll write a quick anonymous (or lambda) function to take the list elements and return the test set MSE

test\_MSEs <- lapply(test\_preds, FUN = function(x) {   
 mean((test$medv - x$fitted.values)^2)  
 })  
test\_MSEs[1:5]

[[1]]  
[1] 58.25614  
  
[[2]]  
[1] 46.16605  
  
[[3]]  
[1] 39.78201  
  
[[4]]  
[1] 38.76825  
  
[[5]]  
[1] 37.40706

* Let’s plot our test set MSE as a function of .

plot(x = kgrid, y = test\_MSEs,  
 xlab = "K",  
 ylab = "Test Set MSE")



* We can find the optimal programmatically:

## Optimal K  
k\_opt <- kgrid[which.min(test\_MSEs)]  
k\_opt

[1] 30

#show the test MSE corresponding to the optimal K  
test\_MSEs[[k\_opt]]

[1] 33.26467

* Now we refit on the entire data set using this optimal .

knn\_best\_fit <- kknn(medv ~ lstat,   
 train = Boston,   
 test = Boston,   
 k = k\_opt) #fit the model

* To predict for future observations, we could refit using a different test argument. However, we’ll see better packages for doing this type of action across many models soon!

The test MSE of 33.26, equivalently, RMSE 5.77 gives us an unbiased estimate of prediction error of our procedure in unseen test data. This also reflects the added variability due to tuning of the hyperparameter. Note again that for prediction purposes, we will still use the model fitted to the whole data.

The advantage of the holdout method is that it is conceptually and computationally simple. However, this method can produce highly variable test error!

* To see this, we can repeat the hyperparameter tuning procedure a few times. The plot of the test MSE profiles during the tuning process over multiple splits of the data is shown in Figure for 10 training runs. As we see, there is a substantial amount of variability in the test MSE.

|  |
| --- |
| The image is a line graph plotting the test mean squared error (MSE) against varying values of K, ranging from 0 to 100 on the x-axis. The y-axis measures the test MSE from 0 to 60. Ten different lines represent results from “Split 1” to “Split 10,” each with a distinct color. The lines generally show a downward trend initially, reaching a minimum, and then ascending as K increases. Each line fluctuates differently in terms of steepness and slope direction. A legend on the right side matches each line color to its corresponding split.  Test MSE during tuning hyperparameters for 10 runs of the model training. |

* We can apply the train/test idea multiple times to get a more stable estimate. However, if we do this too much, we may end up overfitting.
* Often we split our data into a training and test set and do our tuning over multiple splits on the **training set alone**.
* We save our test set to only be used sparingly on a **final model evaluation**.

## -fold Cross-validation (V-fold CV)

The V-fold CV procedure splits the data into multiple parts, and then cycles through those parts to compute test MSE. In particular, V-fold CV is performed to estimate the test error of a model/procedure as follows:

1. Split the data randomly into (roughly) equal sized disjoint parts, called *folds*. Thus we have fold 1, , fold .
2. For each fold , do:
   1. Set Fold as the test set, and the remaining folds together as the training set.
   2. Train the model using the training set and compute MSE using the test set (Fold ), say . Note: We can use any other performance metric, e.g., MAE, classification accuracy etc. here.
3. The final estimate of test error is formed by taking the average of the MSE values: .

Keep in mind that the model training step can also include tuning hyperparameter(s) as well. Figure shows the layout of -fold CV procedure.

|  |
| --- |
| The image is a diagram illustrating the process of cross-validation, specifically k-fold cross-validation. To the left, there is a labeled block titled "Data," from which three arrows extend to the right. Each arrow leads to a row labeled with sequences of "Test" and "Train" blocks, corresponding to different data folds. The first row shows "Test" in Fold 1 followed by "Train" in subsequent folds. The second row begins with "Train," then "Test" in Fold 2, followed by "Train" in the remaining folds. The last row starts with "Train," with "Test" in the last fold. To the right of each row, the label "MSE" follows, denoting mean squared error for each fold, labeled as MSE1, MSE2, and MSEV. These are converging towards a boxed formula representing the average of these MSE values.  Layout of the V-fold crossvalidation procedure. Data are first randomly split into V equal sized parts, called folds. Each fold is then used as a test set while the remaining folds are used to fit the model. The test error is estimated by taking the average of the MSEs from the V folds. |

Let us apply CV to our Boston data using KNN to select our hyperparameter .

1. Split the data randomly into folds.
2. For each fold , do:
   1. Set Fold as the test set, and the remaining folds together as the training set.
   2. Fit the model using the training set, and evaluate MSE/RMSE using the test set (Fold ), *for each value of the hyperparameter*.
3. From step 2., for each value of hyperparameter, we should have a MSE/RMSE value for each fold ( of them). The final MSE/RMSE for each of the hyperparameter value is calculated by taking the mean of MSE/RMSE values from the folds. Chose the optimal value of the hyperparameter by minimizing the final MSE/RMSE.
4. Use the best hyperparameter value to refit the model on the whole dataset.

For now, we won’t split our data into a training and test set first. We’ll simply use CV to tune our hyperparameter. We’ll further discuss using both a training/test split and cross-validation shortly.

To ease our programming burder, we’ll look at using the caret package in R (similar packages exist in python). caret allows us to easily specify our tuning method, keeps track of all results, and allows us to easily do predictions on new observations.

* Let us tune using 5-fold CV. Figure shows the MSE profile for the tuning process.
* We start by setting a seed for reproducibility and specifying our kgrid for clarity.

set.seed(1001)  
## Set K grid  
kgrid <- expand.grid(k = c(1:100))

* Next, we set up our trainControl() parameters. This is caret’s method for specifying how to train our model. We choose method = cv and number = 5 to do 5 fold cross-validation.

## Training control params  
cv <- trainControl(method = "cv",  
 number = 5)

* Next, we fit the model using caret’s train() function.
  + This takes a formula
  + The data to train on
  + The method or model type ot fit
  + A grid of tuning parameters (if applicable)
  + The method for training

## Fit the model  
knn\_fit <- train(medv ~ lstat,  
 data = Boston,  
 method = "knn",  
 tuneGrid = kgrid,   
 trControl = cv)

* We can run the generic plot() function on the result to see the CV test error profile.

plot(knn\_fit)

|  |
| --- |
| The image depicts a line graph on a white grid background. The x-axis is labeled "#Neighbors" ranging from 0 to 100, and the y-axis is labeled "RMSE (Cross-Validation)" ranging from 5.0 to 7.5. The data points are represented by small circles connected by a line that starts high at the left side, dips to its lowest point around 20 neighbors, and then gradually rises again towards the right side. The graph shows an initial sharp decline followed by a gradual increase in error rate, indicating an optimal region for the number of neighbors.  Results from hyperparameter tuning using 5-fold CV. |

* The bestTune list element tells us the optimal tuning parameter using the default criteria of RMSE

knn\_fit$bestTune

k  
36 36

* Lastly, we’ll refit to the entire data set with the optimal

## Optimum K and model refit on full data   
k\_opt <- knn\_fit$bestTune$k  
knn\_tuned <- train(medv ~ lstat,  
 data = Boston,  
 method = "knn",  
 tuneGrid = expand.grid(k = k\_opt),   
 trControl = trainControl(method = "none"))

We can use the final fitted model for further predictions!

## Leave-One-Out Cross-Validation (LOOCV)

As a special case of -fold cross-validation, consider the case with , where is the sample size of your data. In this case, every observation will be its own fold. Suppose we observe data for . The CV then proceeds as follows:

1. For observation (fold) , do
   * Set the -th observation as the test set, and the remaining as the training set.
   * Fit the model on the training set, and predict (test set)
   * Compute
2. Compute the test MSE as the average of the MSE values from step 1., that is, .

This procedure is known as *leave-one-out cross-validation* (LOOCV).

Let’s do an example using LOOCV in caret.

* In caret we can specify method = "LOOCV" in the trainControl() specification.
* Figure shows the MSE profile for tuning in the Boston data.

## Values of K, and LOOCV specification  
kgrid <- expand.grid(k = 1:50)  
loo <- trainControl(method = "LOOCV")  
## Model fit  
fit <- train(medv ~ lstat,  
 data = Boston,  
 method = "knn",  
 trControl = loo,  
 tuneGrid = kgrid)  
plot(fit)

|  |
| --- |
| The image depicts a line graph on a white grid background. The x-axis is labeled "#Neighbors" ranging from 0 to 100, and the y-axis is labeled "RMSE Leave-One-Out-Cross-Validation" ranging from 5.0 to 7.5. The data points are represented by small circles connected by a line that starts high at the left side, dips to its lowest point around 35 neighbors, and then gradually rises a little towards the right side. The graph shows an initial sharp decline followed by a gradual increase in error rate, indicating an optimal region for the number of neighbors.  Results from tuning K using LOOCV on the whole Boston data. |

A disadvantage of LOOCV is its potential heavy computation cost, especially for large sample size.

* For example, in Boston data (), we have to fit models for *each* value of ! This can be extremely difficult for larger .
* In contrast, holdout and -fold CV procedures are more computationally efficient.

When we estimate the test error, we might have different goals to do so in different situations. When we are interested in evaluating model performance in a test set, the actual value of the test error is of interest. However, when we are tuning a hyperparameter (e.g., K in KNN regression), our primary goal is to find the *minimizer of test error*, rather than test error itself. In the former case, the accuracy of the cross-validation estimates might be an issue. But in the later case, the minimizer might still be valid even if the estimate of the test error itself is not accurate. Examples from several simulation studies have been presented in the textbook (Introduction to Statistical Learning) to examine the point made above.

As a final note on cross-validation, the choice of in -fold cross-validation depends the bias-variance trade-off (See Chapter 5.1.4 of *Introduction to Statistical Learning, second edition* for a detailed discussion) of the procedure. Given a sample size of , the -fold CV uses approximately observation to fit the model. Thus LOOCV effectively uses the whole data to train the model, and therefore produces almost unbiased estimates of the test error. However, a 5-fold CV might produce a biased estimate. On the other hand, in LOOCV the model fits essentially uses the same dataset (any two fits share common training observations), the resulting test MSE values are highly correlated. Averaging the in MSE values LOOCV does not reduce the variance due to them being highly correlated. Thus LOOCV estimates tend to have high variance. In contrast, a 5-fold CV does not have as high level of overlap between the training folds, and produces less variable estimates of test MSE. In practice, we most often use 5-fold or 10-fold cross validation.

## Bootstrapping

Recall that in the holdout method, we used simple random sampling without replacement to create a holdout set smaller than the original data. In contrast, a *bootstrap sample* is a random sample *with replacement* that is of the *same size* as the original data. Since the sampling is performed with replacement, some observations (rows) will be repeated in the bootstrap sample, and therefore a few observations in the original data will not be included in the bootstrap sample. The omitted observations are called *out-of-bag (OOB)* samples.

In statistical learning, we train our model using the bootstrap sample, and test using OOB samples. We do not use a single bootstrap sample however; instead, many bootstrap samples are drawn, and the model is trained/tested repeatedly.

* We can perform bootstrap manually using the bootstraps() function in rsample package. The code below draws 10 bootstrap samples from the Boston data.

set.seed(10)  
# Bootstrap samples  
boot\_sample <- bootstraps(Boston, times = 5)  
boot\_sample

# Bootstrap sampling   
# A tibble: 5 × 2  
 splits id   
 <list> <chr>   
1 <split [506/195]> Bootstrap1  
2 <split [506/180]> Bootstrap2  
3 <split [506/173]> Bootstrap3  
4 <split [506/185]> Bootstrap4  
5 <split [506/192]> Bootstrap5

* We can look at the different splits and obtain training and testing sets using the training() and testing() functions!

# Accessing the first bootstrap sample  
boot\_1 <- training(boot\_sample$splits[[1]])  
dim(boot\_1)

[1] 506 13

# Corresponding out of bag samples to test on  
oob\_1 <- testing(boot\_sample$splits[[1]])  
dim(oob\_1)

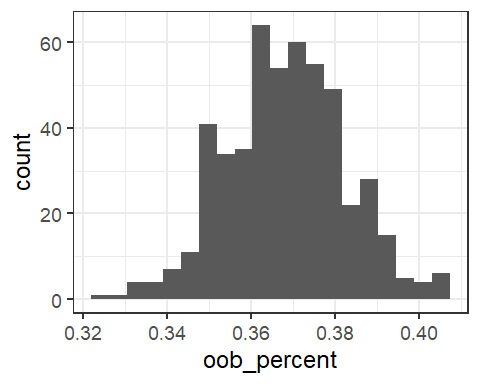
[1] 195 13

* Recall, we want the distribution of our response variable in our training set(s) to mimic the distribution across all the data
* The distribution of our response (medv) in the five bootstrap resamples is compared to the overall distribution in Figure below. We see good agreement overall!

|  |
| --- |
| The image is a density plot graph with a smooth curve illustrating the distribution of a dataset. The x-axis is labeled "medv," ranging from 0 to 50, and the y-axis is labeled "density," ranging from 0.00 to 0.06. The plot features a prominent red line indicating the main density estimation and two black dashed lines outlining the confidence intervals around the red line. The curve shows a sharp peak near 20 on the x-axis, after which it declines steadily and rises slightly around 30 before flattening out towards the end of the range.  Distribution of medv in the Boston data (red solid line), and in 10 bootstrap samples (black dashed lines). |

* As we mentioned, the bootstrap samples will leave a proportion of observations out. On average, the proportion left out here can be estimated empirically! Here we find the sample proportion left out and plot this distribution.

boot\_sample <- bootstraps(Boston, times = 500)  
oob\_percent <- sapply(boot\_sample$splits,  
 function(s){nrow(testing(s))/nrow(training(s))})  
ggplot() +   
 geom\_histogram(aes(x=oob\_percent), bins = 20) +   
 theme\_bw(base\_size = 18)



* On average, we have about 36.83 percent of observations are OOB.

### General Uses of Bootstrapping

The Bootstrap is a general method, and can be used to assess accuracy of statistical procedures. Given a dataset $\cal D$, suppose we want to compute some quantity $S({\cal D})$ based on the whole dataset. We can use bootstrap to assess any aspect of the distribution of $S({\cal D})$ (e.g., mean, variance, quantiles etc.) as follows:

* Draw bootstrap samples from the original data, call them ${\cal D}^\*\_1, \ldots, {\cal D}^\*\_B$.
* For , do
  + Use the -th bootstrap sample, ${\cal D}\_b^\*$ to compute the same quantity you computed based on the original data, $S({\cal D}\_b^\*)$. For example, if we want to compute sample mean of the original data, we would need to compute sample mean using the bootstrap sample as well.
* Use the bootstrap estimates $S({\cal D}\_1^\*), \ldots, S({\cal D}\_B^\*)$ to assess properties of $S({\cal D})$.

Figure shows a layout of using bootstrap as described above.

|  |
| --- |
| The image is a flowchart illustrating a bootstrap process. On the left, a gray box labeled "Data: (D)" and "Goal: S(D)" has three arrows pointing right towards separate gray rectangles labeled, respectively, as "Bootstrap sample 1: D-star 1", "Bootstrap sample 2: D-star 2", and "Bootstrap sample B: D-star B". Each of these rectangles has an arrow pointing further right toward expressions (S(D-star 1)), (S(D-star 2)), and (S(D-star B)) respectively, indicating bootstrap replicates. These replicates converge with arrows pointing toward a blue box labeled "Bootstrap Distribution of (S(D))".  Layout of bootstrap procedure. |

For example, we can examine the the variance of our quantity using the sample variance of the replicates:

$$ \widehat{var}\{S(D)\} = \frac{1}{B-1}\sum\_{b=1}^B[S({\cal D}\_b^\*) - \bar S^\*]^2, $$

where

$$ S^\* = \sum\_{b=1}^BS({\cal D}\_b^\*) / B $$

is the sample mean of the bootstrap replicates.

* Consider the example of fitting KNN regression to Boston data with fixed .

knn\_k30 <- knnreg(medv ~ lstat,  
 data = Boston,  
 k = 30)

Suppose we want to estimate when , that is, we want to estimate the expected value of medv when lstat = 5. We may want to estimate the standard error of this quantity or perhaps the distribution of it!

* We can obtain one estimate of this quantity from a fit on the entire data set.

pred\_k30 <- predict(knn\_k30,  
 newdata = data.frame(lstat = 5))  
pred\_k30

[1] 31.81

* Note: The predicted value of when is the same
* However, the variability of these two quantities is not the same
  + Recall, the variability of the predicted value is represented by bias + + irreducible error.
  + In this case, we are only interested in .
* Let’s use the bootstrap to look at the distribution and standard error of
  + We will draw 200 bootstrap samples from Boston data.
  + For each bootstrap sample, we will fit the KNN procedure with , and compute the estimate.

## Wrap the prediction process in a function for easy use  
knn\_k30\_predict <- function(split){  
 # Input: split from bootstrap using rsample  
 # Output: prediction at lstat = 5  
   
 # Get training set  
 train\_set <- training(split)  
 # fit the KNN model with K = 30  
 knn\_k30 <- knnreg(medv ~ lstat,  
 data = train\_set,  
 k = 30)  
 # Predict at lstat = 5  
 pred <- predict(knn\_k30,  
 newdata = data.frame(lstat = 5))  
 return(pred)  
}  
## Draw bootstrap samples  
B <- 200  
boot\_sample <- bootstraps(Boston, times = B)  
## Apply the prediction function to the bootstrap samples   
## sapply is similar to lapply but simplifies what is returned  
boot\_pred <- sapply(boot\_sample$splits, FUN = knn\_k30\_predict)

Figure shows the bootstrap distribution of .

|  |
| --- |
| The image is a line chart depicting a density plot overlaid with two vertical lines, one red and one black. The x-axis is labeled as "Predicted medv at lstat = 5," with values ranging from 28 to 34. The y-axis is labeled "density" and ranges from 0.0 to 0.2. The density curve is a smooth, bell-shaped curve peaking slightly above 0.2 and centered around the x-axis value of 31. Two vertical dashed lines intersect the x-axis: a red line at approximately 30.5 and a black line at around 31.5. The background is a light grid with horizontal and vertical lines.  Distribution of estimator of E(medv when lstst = 5). Also shown the mean of the bootstrap estimates (red solid lile), and original estimate from the full data (black dashed line), |

* Some summaries of the bootstrap estimates are shown below.

## Summary of bootstrap estimates  
summary\_boot <- summary(boot\_pred)   
tibble(statistic = names(summary\_boot),   
 value = summary\_boot) |>   
 kable()

| statistic | value |
| --- | --- |
| Min. | 27.59 |
| 1st Qu. | 30.41 |
| Median | 31.37 |
| Mean | 31.40 |
| 3rd Qu. | 32.33 |
| Max. | 35.02 |

## Variance/SD of the estimate  
c(variance = var(boot\_pred),  
 sdev = sd(boot\_pred))

variance sdev   
1.931623 1.389828

## MSE  
mean( (boot\_pred - pred\_k30)^2 )

[1] 2.090899

### Tuning with the Bootstrap

In a learning method, we can tune hyperparameters using the bootstrap in a similar method to what we did with cross-validation.

* We fit the model using bootstrap samples, and compute test MSE using OOB samples.
* The best hyperparameter value can be chosen by minimizing test MSE.

In caret this can be done by specifying method = bootstrap the trainControl() function.

set.seed(1001)  
## Values of K, and bootstrap specification  
kgrid <- expand.grid(k = 1:100)  
boot <- trainControl(method = "boot",  
 number = 25)  
## Model fit  
boot\_tuned\_knn <- train(medv ~ lstat,  
 data = Boston,  
 method = "knn",  
 trControl = boot,  
 tuneGrid = kgrid)

Now we can plot() the fitted object to investigate the test MSE values for each .

plot(boot\_tuned\_knn)

|  |
| --- |
| The image is a line graph depicting the relationship between the number of neighbors and the root mean square error (RMSE) using bootstrap resampling. The x-axis is labeled "# Neighbors" and ranges from 0 to 100. The y-axis is labeled "RMSE (Bootstrap)" and ranges from 5.0 to 7.5. The graph shows a curve starting from the top left corner, dipping downwards, reaching its lowest point at approximately 5.0 RMSE around 30 neighbors, and then ascending slightly as the number of neighbors increases to 100. Each data point is marked with a diamond shape, and a smooth line connects these points.  Results from bootstrap (25 reps) tuning of K. |

Compared to -fold cross-validation, bootstrap tends to produce less variable estimates. However, on average only observations get represented in bootstrap samples. Thus bootstrap estimates may have some bias similar to using a 2-fold or 3-fold CV.

# Comparing Competing Models

We discussed specifying our function in multiple ways. For instance, we could choose a simple linear or multiple linear regression model instead of a KNN model. In this section, we look at how to use a train/test split along with cross-validation to choose an optimal model from competing models.

* Remember: We want to touch the test data as little as possible. Otherwise, we may end up training to the test set! Again, if we do this type of overfitting, we may no longer generalize well to a future test set.
* In some cases, especially when we don’t have much data, we may choose to simply use CV or the bootstrap to do our tuning and overall model evaluation! We may also do this when we only have one model form that we are considering

## Competing Models

Let’s stick with the Boston data set. We’ll compare the basic simple linear regression model to the KNN model!

For completeness, we’ll go through the entire process here.

## Train/Test split

First, let’s split our data using the basic simple random sample method. We’ll use caret’s functionality to do so.

set.seed(51)  
index <- createDataPartition(Boston$medv,  
 p = 0.8,  
 list = FALSE,  
 times = 1)  
  
train <- Boston[index, ]  
test <- Boston[-index, ]

## Tuning of KNN model Using 10-fold CV

Now that we have a training set, we can use it to select the tuning parameter for the KNN model via 10 fold CV.

cv <- trainControl(method = "cv",  
 number = 10)  
knn\_fit <- train(medv ~ lstat,  
 data = train,  
 method = "knn",  
 tuneGrid = kgrid,   
 trControl = cv)

Let’s look at our best tuning parameter value and the corresponding CV error.

k\_opt <- knn\_fit$bestTune$k  
k\_opt

[1] 29

knn\_fit$results[k\_opt, ] |>  
 kable()

|  | k | RMSE | Rsquared | MAE | RMSESD | RsquaredSD | MAESD |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 29 | 29 | 5.12014 | 0.6951166 | 3.706729 | 0.825852 | 0.0652974 | 0.4008271 |

Now we fit this to the full training set.

knn\_tuned <- train(medv ~ lstat,  
 data = train,  
 method = "knn",  
 tuneGrid = knn\_fit$bestTune,   
 trControl = trainControl(method = "none"))

## Fitting the SLR Model

We can also fit our SLR model on the training data. There is no need to use CV to fit the model. You might use CV to get a basic understanding of how the model does at predicting though. We won’t do that at this time.

For consistency, let’s use caret to fit this model as well.

slr\_fit <- train(medv ~ lstat,  
 data = train,  
 method = "lm",  
 trControl = trainControl(method = "none"))

We can get an idea about the model fit using summary() on the fitted object.

coef <- summary(slr\_fit)$coefficients  
coef |>  
 as\_tibble() |>  
 kable()

| Estimate | Std. Error | t value | Pr(>|t|) |
| --- | --- | --- | --- |
| 34.5837952 | 0.6269547 | 55.16156 | 0 |
| -0.9570221 | 0.0434102 | -22.04602 | 0 |

## Comparison on Test Set

Now that we have a KNN model and an SLR model, we can compare those two models on the test set to determine an overall ‘best’ model!

* First we get our test set predictions.

knn\_preds <- predict(knn\_tuned, newdata = test)  
slr\_preds <- predict(slr\_fit, newdata = test)

* Now we calculate the MSE!

mean((test$medv-knn\_preds)^2)

[1] 30.88224

mean((test$medv-slr\_preds)^2)

[1] 36.16892

Looks like the KNN model performs better at predicting for this data and choice of predictors!

# Summary

In this chapter we discussed the following main concepts.

* -nearest neighbors for regression.
* Evaluation metrics for regression: MSE/RMSE, MAE, etc.
* Bias-variance trade-off in relation to model flexibility.
* Irreducible error in a regression model.
* Training and test MSE/error
* Data splitting methods: Holdout, -fold CV, Leave-One-Out CV, Bootstrap.
* Hyperparameter tuning methods.
* Test error estimation methods.