Model Building and Data Splitting

Arnab Maity

NCSU Statistics ~ 5240 SAS Hall ~ amaity[at]ncsu.edu

Contents

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:
   a. 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*).
   b. 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.
   c. Typically, we repeat stpes `a.` and `b.` few times so that we get repeated measurements of model performance for each value of hyperparameters.[1] The final model performance is taken to be the average of these multiple measurements.
   d. Choose the best value of hyperparameters by optimizing[2] the model perfromance measure obtained in step `c.`
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.

We should note that most model evaluation criteria focus on prediction. Thus the steps describes above are geared towards building of predictive models. After all, the model provided by the optimal value of hyperparameter, while good for prediction, may not be easily interpreted or lend itself to inference.

[1] Using a single validation set often provides highly variable estimate of model performance.

[2] 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".

There are two points in the algorithm above that we may want to perform repeatedly: these are the inner and outer loops.

Inner and outler loops

The tuning process (Step '2.') can repeated, that is, for each candidate value of the hyperparameter(s), we may use multiple splits of the training set rather than just one holdout set. This is the *inner loop*.

The entire process can be performed repeatedly (Step '5.') to get a better estimate of the test error. This is the *outer loop*.

Depending on the situation (sample size, computationa cost), we can use any of the resampling and data splitting methods in each of the loops.

K-Nearest Neighbors Regression

Before proceeding further, let us introduce one of the simplest non-parametric regression methods – the K-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 model

$$Y = f(X) + \epsilon,$$

where $f(\cdot)$ is an unknown function, and ϵ are zero mean random errors with $var(\epsilon) = \sigma^2$, and independent of X. Suppose we have training data $(Y_i, X_i), i = 1, \ldots, n$. Then, for any given value x_0, KNN regression estimates $f(x_0)$ as follows:

- Identify the K observations in the training data such the their X values are "nearest" to x_0.
- Estimate $f(x_0)$ by the average Y values of the points obtained in the previous step.

Formally, suppose $S_K(x_0)$ denotes the indices of the K points whose X values are nearest to x_0. Then we have

$$\widehat{f}(x_0) = \frac{1}{K} \sum_{i \in S_K(x_0)} Y_i.$$

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

Distance metric

We determine K points nearest to x_0 by computing a distance metric between x_0 and the X values of the training data, and taking the K points with smallest distance measures.

The most common distance metric is the Euclidean distance: for two vectors $w = (w_1, \ldots, w_p)$ and $v = (v_1, ldots, v_p)$, the Euclidean distance is

$$d(w, v) = \sqrt{\sum_{i=1}^{p} (w_i - v_i)^2}.$$

This is also known as the L_2-norm of $w - v$, that is, $||w - v||_2$.

Another popular distance metric is the L_1-norm, $||w - v||_1$, that is,

$$d(w, v) = \sum_{i=1}^{p} |w_i - v_i|.$$

The L_1-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 L_1 distance is also known as "taxicab" and "Manhattan" distance. The geometry of these distance metrics are shown (simplified) in Figure 1.

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

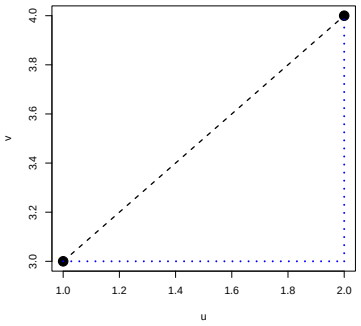


Figure 1: L2-norm vs. L1-norm. Given two points (black dots), the L2-norm measures the distance of the straight line joining them (dashed line). In contrast, L1-norm measures the distance of the path that can only go parallel to the x- and y-axes (dotted line).

The hyperparameter K

Let us consider the `Boston` dataset in the `ISLR2` package. The data set contains housing values of $n = 506$ suburbs of Boston. Suppose we want to predict median value of owner occupied homes (in \$1000's, `medv` variable) using the lower status of the population (percent, `lstat` variable). A snapshot of the data is shown below with only the two variables of interest. A plot of the data is shown in Figure 2.

```
## # A tibble: 6 x 2
##    medv lstat
##   <dbl> <dbl>
## 1  24    4.98
## 2  21.6  9.14
## 3  34.7  4.03
## 4  33.4  2.94
## 5  36.2  5.33
## 6  28.7  5.21
```

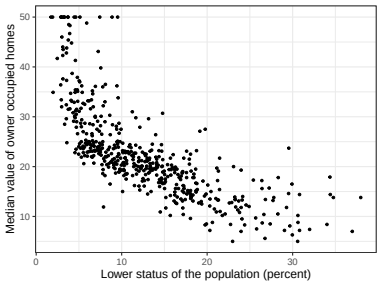


Figure 2: Plot of median housing value vs. percent of population with lower status from Boston data.

Let us see a KNN fit to the data, with $K = 30$.[3] Here we are not training/testing the model yet – we are simply attempting to understand the role of the hyperparameter K and its impact on the fitted model. We can use the function `knnreg()` in the `caret` library.

[3] The value $K = 30$ is chosen arbitrarily for demonstration purposes.

```
library(caret)

# Fit KNN with K=30
knn_fit <- knnreg(medv ~ lstat,
                  data = Boston,
                  k = 30)

# Create prediction grid
xgrid <- list(lstat = seq(2, 37, len=201))

# Perform prediction
fitted_values <- predict(knn_fit, newdata = xgrid)
```

After the fitting the regression, we plot the fitted function $\widehat{f}(\cdot)$ on a grid of 201 equally spaced values in $[2, 37]$ – this interval roughly covers the observed values for `lstat`. The fitted function is shown in Figure 3.

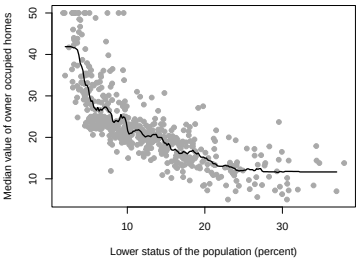


Figure 3: KNN fit to Boston data with K=30.

```
# Plot
plot(Boston$lstat, Boston$medv,
     pch=19,
     col = "darkgray",
     xlab = "Lower status of the population (percent)",
     ylab = "Median value of owner occupied homes")
lines(xgrid$lstat, fitted_values, lwd=2)
```

How should we choose K? To answer this question, we need to investigate how the estimated function changes for different values of K. We show three fitted functions for $K = 1$, 30, and 300 in Figure 4.

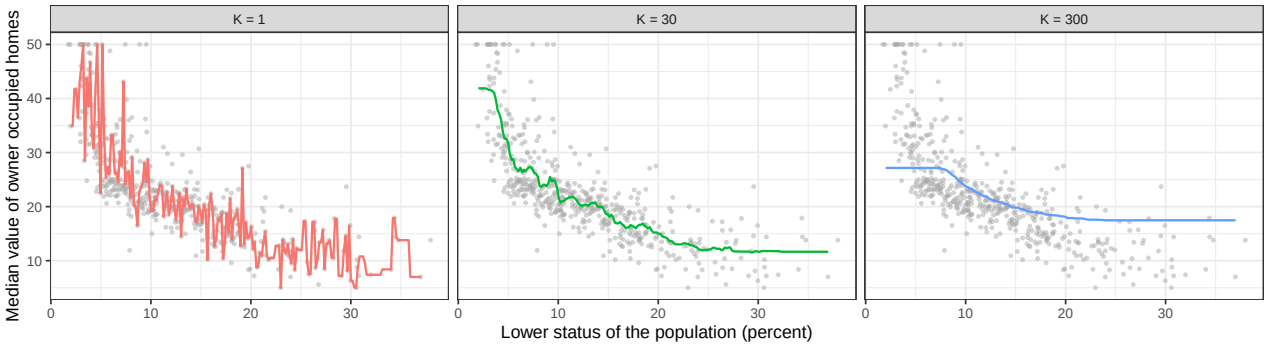


Figure 4: Estimated functions form Boston data example for different values of K.

We note that for small value of $K = 1$, KNN produces extremely rough estimate of $f(\cdot)$. 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 $K = 300$ – this is 60% 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 biased estimate of $f(\cdot)$, and inaccurate predictions.

For $K = 30$, 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 K (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 x_0, and want to predict the response Y corresponding to x_0. The (squared) prediction error is $\{Y - \widehat{f}(x_0)\}^2$. However, we want the procedure to provide good predictions across all possible values of Y when $X = x_0$,[4] so we might want to choose a model by minimizing expected prediction error[5] for $X = x_0$,

$$E[\{Y - \widehat{f}(x_0)\}^2 | X = x_0].$$

This strategy works if we are only interested in the specific value $X = x_0$. In general, we want a procedure which can predict for all possible values of X, 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 x_0, that is,[6]

$$E\left(E[\{Y - \widehat{f}(x_0)\}^2 | X = x_0]\right) = E[\{Y - \widehat{f}(X)\}^2].$$

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 $(Y_i, X_i), i = 1, \ldots, n$, and a test set $(Y_i, X_i), i = n+1, \ldots, n+m$. Then, based on the test set, we can estimate the quantity above as

$$\frac{1}{m} \sum_{i=n+1}^{n+m} (Y_i - \widehat{f}(X_i))^2,$$

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.

[4] Note that when $X = x_0$, the response Y is not just a single number – it is a random variable. For example, if we assume $\epsilon \sim N(0, \sigma^2)$, we have $Y|X = x_0 \sim N(f(x_0), \sigma^2)$. Thus, for $X = x_0$, the response could be any realization from this distribution.

[5] Also known as generalization error – see *Elements of Statistical Learning* by Hastie, Tibshirani and Friedman, 2009, for more details.

[6] The equality in the equation follows by law of iterative expectation: for random variables W and Z, $E[E(W|Z)] = E(W)$.

Mean Squared Error (MSE)

MSE: Given responses Y_i and their predictions $\widehat{Y}_i = \widehat{f}(X_i)$, the Mean Squared Error is defined as

$$Average_i(Y_i - \widehat{Y}_i)^2$$

Training/Test MSE: If the MSE is computed on the training set (the set used to fit the model), then the resulting quantity is called *Training MSE*:

$$MSE_{\text{train}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$$

MSE computed on an external test data (independent of the training data) is called *test MSE*:

$$MSE_{\text{test}} = \frac{1}{m}\sum_{i=n+1}^{n+m}(Y_i - \widehat{Y}_i)^2$$

Using training MSE to evaluate model performance is often misleading and results in overfitting the data. As an example, consider using KNN regression with $K = 1$. The training MSE is zero (or close to zero depending on how KNN handles ties in the X values).[7] 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.

In general, when we evaluate a model or algorithm, we do not care about how it performs in the training set. Instead, we are interested in its performance on new unseen data (test data) independent of the training data. In other words, we want a method that can be generalized to new data. Thus, a better option to evaluate a model is the *test MSE*.

To visualize this phenomenon, let us conduct a simulation study where we know the true form of the function $f(\cdot)$, and thus can simulate data using it. We can simulate a training set and a test set. We can then fit KNN regression model with different values of K, and for each case compute the training and test MSE. Figure 5 shows results for one such experiment. We see that the test MSE is generally higher that the training MSE. Training MSE keeps increasing as K 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 $K = 50$, while lowest training MSE is for $K = 1$.[8]

[7] Since for each X_i in the dataset, the nearest point of X_i is itself. Thus the prediction $\widehat{Y}_i = Y_i$, resulting in (near) zero training MSE!

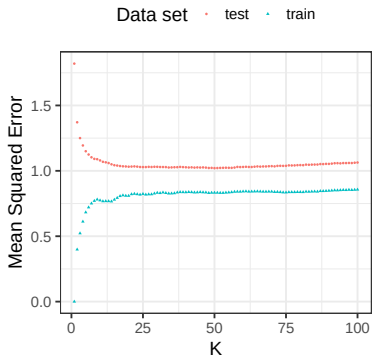


Figure 5: Training and test MSE for simulated data for different values of K. Larger values of K correspond to less flexibility.

[8] For more such examples and detailed discussion, see Chapter 2 of *An Introduction to Statistical Learning* by James et al.

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 $E[\{Y - \widehat{f}(X)\}^2 | X = x_0]$ for the test data x_0. Some algebra gives us[9]

$$\begin{aligned} E[\{Y - \widehat{f}(x_0)\}^2 | x_0] &= E[\{Y - f(x_0) + f(x_0) - \widehat{f}(x_0)\}^2 | X = x_0] \\ &= E[\{Y - f(x_0)\}^2 | X = x_0] + E[\{\widehat{f}(x_0) - f(x_0)\}^2] \\ &= var(\epsilon) + E[\{\widehat{f}(x_0) - f(x_0)\}^2] \\ &= \sigma^2 + E[\{\widehat{f}(x_0) - f(x_0)\}^2]. \end{aligned}$$

[9] The cross-product term in the second line can be shown to be zero for test data set, under the assumption that the test data is independent of the training data.

The first term σ^2 is a fixed parameter which we can not control. Even if we have a extremely accurate estimation procedure for $f(\cdot)$ so that $\widehat{f}(X) \approx f(X)$, we would still have the expected prediction error to be σ^2. Thus the term σ^2 is called the *irreducible error* – it is the variance of the target.[10]

[10] We have that $\sigma^2 = var(Y|X)$. Even if we know true f, this variance remains.

The second term is under our control, and depends of the method of estimation of $f(\cdot)$. 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:[11]

$$\begin{aligned} &E[\{\widehat{f}(x_0) - f(x_0)\}^2] \\ &= E\left([\widehat{f}(x_0) - E\{\widehat{f}(x_0)\}]^2\right) + E\left([E\{\widehat{f}(x_0)\} - f(x_0)]^2\right) \\ &= var\{\widehat{f}(x_0)\} + [Bias\{\widehat{f}(x_0)\}]^2. \end{aligned}$$

[11] Recall, that for a random variable W, $var(W) = E[\{W - E(W)\}^2]$. Also, for an estimator $\widehat{\Theta}$ of a parameter θ, $Bias(\widehat{\Theta}) = E(\widehat{\Theta}) - \theta$.

Collecting all the terms, we have that

$$E[\{Y - \widehat{f}(x_0)\}^2 | x_0] = \sigma^2 + var\{\widehat{f}(x_0)\} + [Bias\{\widehat{f}(x_0)\}]^2.$$

Thus the expected prediction error is a combination of the variance and squared bias of the estimator $\widehat{f}(x_0)$.

We again resort to a simulation experiment to see the relative contribution of the variance and squared bias of $\widehat{f}(x_0)$ to the prediction error. Figure 6 shows one simulated training set of size $n = 500$ along with the true function used to generate the data. We generate multiple such training sets, and for each set we fit KNN regression with $K = 1$, 30 and 300. The test set if a grid of 101 equally spaced points in $[0.01, 0.99]$.

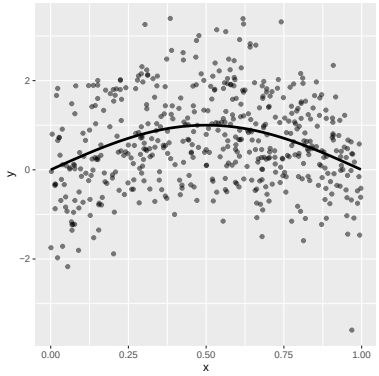


Figure 6: Simulated data of size n=500.

The estimated functions are shown in Figure 7. We notice that for $K = 1$ (the most flexible situation), the estimated functions have high variance, but on average captures the true function well producing low bias. In contrast, for $K = 300$ (least flexible case), the estimates have much less variance but show high bias. For $K = 30$, it seems both the bias and variance are balanced. Thus, when looking at expected prediction error, or its sample version computed by test MSE,

the fit with $K = 1$ results in high MSE due to variance dominating the low bias. The $K = 300$ case results in higher MSE that $K = 30$ due to high bias even though the variance is small. The fit with $K = 30$ seems to balance both bias and variance.

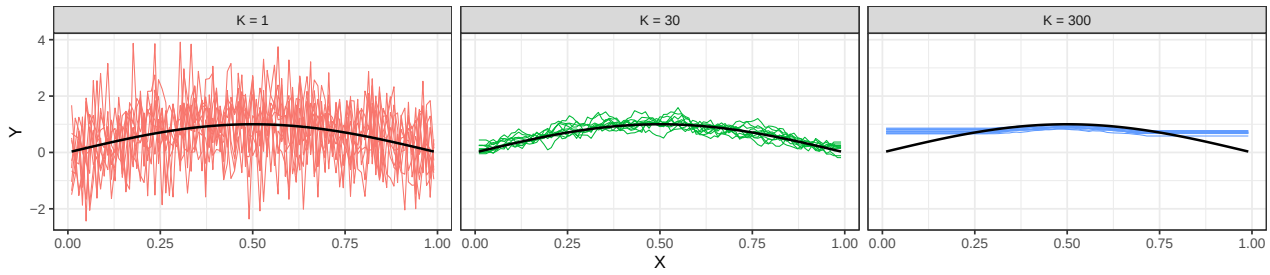


Figure 7: Simulated data showing bias and variance of KNN fits.

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.

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,

$$MAE = n^{-1} \sum_i |Y_i - \widehat{Y}_i|.$$

  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).
- R^2: proportion of variance explained by the model.

$$R^2 = 1 - \frac{\sum_i (Y_i - \widehat{Y}_i)^2}{\sum_i (Y_i - \bar{Y})^2}.$$

  A nice property of R^2 is that it will be always between 0 and 1. R^2 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

Let us re-examine KNN regression fit to `Boston` data. Suppose we use $K = 30$.

```
# Fit KNN with K=30
knn_fit <- knnreg(medv ~ lstat,
                  data = Boston,
                  k = 30)

# Predictions
pred <-  predict(knn_fit,
                 newdata = data.frame(lstat = Boston$lstat))

# Training MSE
MSE_train <- mean( (Boston$medv - pred)^2 )
MSE_train
```

```
## [1] 25.88429
```

So we see that we have a training MSE about 25.88. However, as we have discussed so far, relying on training MSE is not a good idea. We want to know how the model performs on independent test data. Also, is $K = 30$ a good choice? Both both these issues, 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 from our data, and validate our models performance.

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).[12] In practice, $80\% - -20\%$, $70\% - -30\%$ or $60\% - -40\%$ splits are commonly used for training/test sets. 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 $p > n$ also may impact the split sizes. Figure 8 shows the basic layout of the holdout method.

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.

[12] Various authors use different terminology here. We will use these names interchangeably.

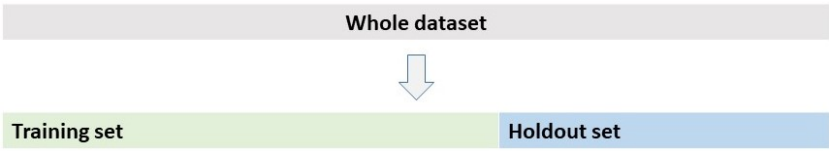


Figure 8: The holdout method. The whole dataset is split into two parts: traing and holdout sets.

Consider the `Boston` data again. In base `R`, we can use the `sample()` function[13] to perform SRSWOR, as follows.

[13] See `?sample` for details.

```
# 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 80% – 20% in the example above.

The following code chunk shows examples of SRSWOR using `caret` and `rsample`, if we want to split the data manually.[14]

[14] Various packages such as `caret`, `mlr3` and `h20` etc. have holdout methods built into their system so that we often do not have to do the data splitting manually.

```
# Using caret
library(caret)
index <- createDataPartition(Boston$medv,
                             p = 0.8,
                             list = FALSE,
                             times = 1)

train.2 <- Boston[index, ]
test.2 <- Boston[-index, ]
```

```
# Using rsample
library(rsample)
index <- initial_split(Boston,
                       prop = 0.8)

train.3 <- training(index)
test.3 <- testing(index)
```

Ideally, the distribution of Y in the test set will be similar to that in training set. Figure 9 shows the corresponding distributions (estimated probability densities) of `medv` for the training/test sets using each of the methods described above.

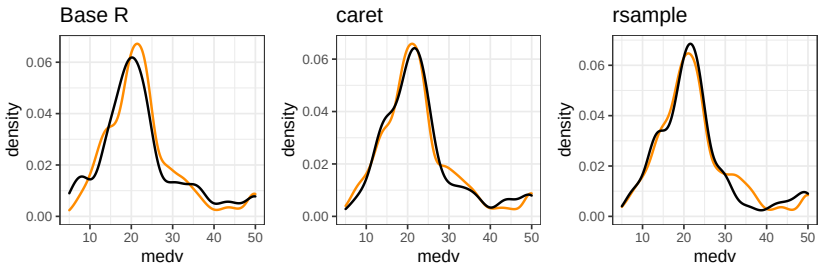


Figure 9: Estimate density functions for 'medv' variable in training (orange) and test (black) sets as obtained using base R, caret, and rsample packages.

A disadvantage of SRSWOR is that it does not always preserve distribution of the response variable. For example, in a classification problem with two classes ('Yes' and 'No'), we might have 70% individuals in 'Yes' group and the remaining 30% in the 'No' group. Performing SRSWOR in the data may lead to a test set with over-representation/under-representation of the groups.[15] In this case, a stratified sampling strategy is appropriate.

Stratified random sampling is used to explicitly control aspects of the distribution of Y. This is useful with data with small sample size or skewed response distribution. Stratified random sampling strategy is to draw sample for each group (strata) of Y so that the test set represents the distribution of Y of the whole data.[16] We can use the `initial_split()` function as before for this purpose but with an extra argument `strata`.

If extreme class imbalance is present in the data (say 90% "No" and only 10% "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)*,[17] which generates synthetic samples from the rare class. In particular, SMOTE takes a random observation from the rare class and then finds its

[15] This issue can arise in a regression problem where Y might have a skewed distribution. The ideal test set should contain both small and large values of Y. SRSWOR can not guarantee this.

[16] For continuous Y, we might split Y into multiple groups based on its quantiles, and sample from each group.

[17] 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.

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. The `caret` package has SMOTE implementation as a possible sampling strategy. Authors of SMOTE also suggest that a combination of SMOTE and undersampling the majority class works better than just using SMOTE.

Let us now investigate the holdout method using the `Boston` data. Recall, that for $K = 30$, the training MSE was approximately 25.88.

```
set.seed(1001)

# (Using rsample) train/test sets (80/20)
index <- initial_split(Boston,
                       prop = 0.8)

train <- training(index)
test <- testing(index)

# Fit knn on training set with K = 30
knn_fit <- knnreg(medv ~ lstat,
                  data = train,
                  k = 30)

# Predictions on test set
pred <- predict(knn_fit,
                newdata = data.frame(lstat = test$lstat))

# Test MSE
MSE_test <- mean( (test$medv - pred)^2 )
MSE_test
```

```
## [1] 31.31455
```

Thus the test MSE is 31.31, which is slightly higher that the training MSE. It is as we expected – training MSE most likely underestimates the prediction error, while test MSE can be viewed as a reasonable estimate of the same. *It is important to remember that we are operating with the setting $K = 30$ - the test MSE might not reflect the best performance the model can have.*

Now let us address the question about choosing the optimal K, that is, the value of K that gives the best general performance. For the full data set, we can tune K using holdout method, and fit the resulting model to the whole data. In particular,

(a) Split the data into training and test sets

(b) For each candidate value of K, fit the model in the training set, and compute MSE using the test set.

(c) Choose the K which gives minimum test MSE.

(d) Fit KNN with optimal K to the full data set.

Then the trained model can be used for future predictions.

```
set.seed(1001)
## (Using rsample) train/test (80/20)
index <- initial_split(Boston,
                       prop = 0.8)
train_set <- training(index)
test_set <- testing(index)

## Fit KNN using train for different values of K
## and compute MSE on the test set
# Candidate values of K
kgrid <- c(1:100)
# vector to store mse values for different k
mse <- rep(NA, length(kgrid))
# run through all k values
for(kn in kgrid){
  fit <- knnreg(medv ~ lstat,
                data = train_set,
                k = kn)
  pred <- predict(fit,
                  newdata = test_set)
  mse[kn] <- mean((pred - test_set$medv)^2)
}

## Optimal K
k_opt <- kgrid[which.min(mse)]

## Refit training set with optimal K
fit_final <- knnreg(medv ~ lstat,
                    data = Boston,
                    k = k_opt)
```

It turns out that the optimal choice of K is $K_{opt} = 35$. The plot of MSE profile obtained form the tuning process is shown in Figure 10.

Before proceeding further, let us take a look into the `caret` package, and implement the procedure using `caret`'s functionality. As we will see, much of the code above can be streamlined. The plot of MSE

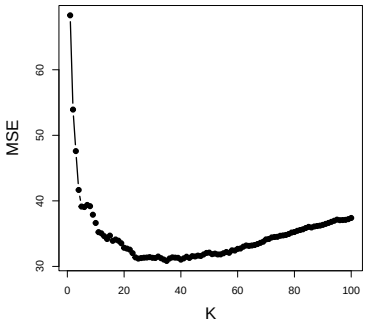


Figure 10: MSE profile for tuning K.

profile obtained form the tuning process is shown in Figure 11 for this run.

```
set.seed(1001)
## Candidate values of K
kgrid <- expand.grid(k = c(1:100))

## Parameters governing training process
hold <- trainControl(method = "LGOCV",
                     p = 0.8,
                     number = 1)

## Training the model
knn_fit <- train(medv ~ lstat,
             data = Boston,
             method = "knn",
             tuneGrid = kgrid,
             trControl = hold
             )

## Plot Root MSE (RMSE)
plot(knn_fit, lwd=2, pch=19)
```

In the code block above[18], first we setup the grid of values for the hyperparameter using the `expand.grid()` function. This creates a dataframe with the candidate values. This can be done for multiple hyperparameters as well

Next, we use the `trainControl()` function to create parameter specification for the training process.[19] The option `LGOCV` is the hold-out method, `p=0.8` specifies the size of training set, and `number = 1` specifies how many times this process is repeated.

Finally, the `train()` function performs the training according to the specifications. The argument `method = "knn"` ensures that we are running KNN regression.

Now we ask again: how does this entire procedure work in general, that is, can we estimate the generalization error of this procedure *including the tuning of the hyperparameter*? Here also, we can use a holdout approach:

- Split the data into training/test set.
- Apply the procedure described above (including tuning), that is, all the steps (a) – (d) to the training set to get the final model.
- Compute MSE using the test set.

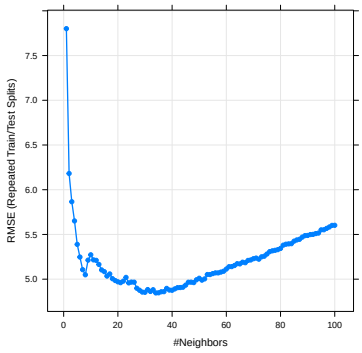


Figure 11: MSE profile for tuning K using caret.

[18] Execute these lines of code yourself and examine the output of each step to better understand the process.

[19] It does not actually train the model yet. It just creates a blueprint for the process.

Let us use `caret` again to perform these steps. Recall we have already set the candidate values in `kgrid`, and the training specifications in `hold` in the previous code block. Also, we have split the data into training/test sets before – they are stored in `train_set` and `test_set`, respectively. Thus we present only code that is new.

```
## Training the model on train_set
knn_fit <- train(medv ~ lstat,
             data = train_set,
             method = "knn",
             tuneGrid = kgrid,
             trControl = hold
             )
plot(knn_fit)
```

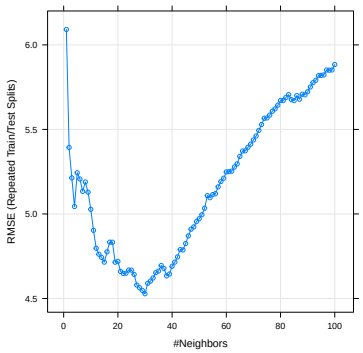


Figure 12: MSE profile for tuning K using holdout method for tuning in the training set (80 percent of the whle data).

```
## Optimal K, and refit on the training set
# optimal K
k_opt <- knn_fit$bestTune$k
# Refit with optimal K
knn_fit <- train(medv ~ lstat,
             data = train_set,
             method = "knn",
             tuneGrid = expand.grid(k = k_opt),
             trControl = trainControl(method = "none")
             )

## Predict test_set and compute MSE
pred <- predict(knn_fit, newdata = test_set)
MSE_test <- mean( (test_set$medv - pred)^2 )
MSE_test
```

```
## [1] 31.31455
```

The test MSE of 31.31, equivalently, RMSE 5.6 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 tuning process is shown in Figure 13 for 10 training runs. As we see, there is a substantial amount of variability in the test MSE.

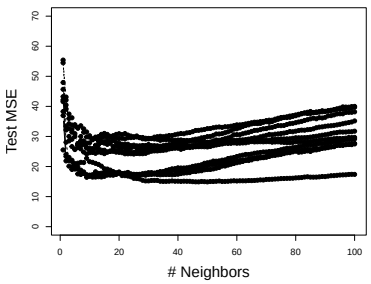


Figure 13: Test MSE during tuning hyperparameters for 10 runs of the model training.

Another possible disadvantage is that the holdout method may overestimate the test error since we are fitting the statistical model with only a subset of the whole data.

Both these issue might be solved if we repeat the inner/outer loops a few times, and take average of the resulting MSE values. Resampling techniques such as cross-validation provides a natural way to do so.

V-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 V (roughly) equal sized disjoint parts, called *folds*. Thus we have fold 1, ..., fold V.
2. For each fold $\ell = 1, \ldots, V$, do:
   a. Set Fold ℓ as the test set, and the remaining folds together as the training set.
   b. Train the model using the training set and compute MSE[20] using the test set (Fold ℓ), say MSE_ℓ.
3. The final estimate of test error is formed by taking the average of the V MSE values: $\frac{1}{V}\sum_{\ell=1}^{V} MSE_\ell$.

[20] We can use any other performance metric, e.g., MAE, classification accuracy etc. here.

Keep in mind that the model training step can also include tuning hyperparameter(s) as well. Figure 14 shows the layout of V-fold CV procedure.

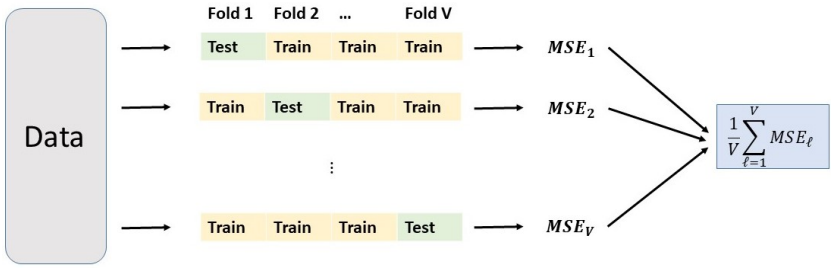


Figure 14: Layout of the V-fold cross-validation procedure. Data are first randomly split into V equal sized parts, called folds. Each fold is then used as a test set while the remaining folds are used to fit the model. The test error is estimated by taking the average of the MSEs from the V folds.

Let us apply CV in practice. Recall, we started our discussion of data splitting by fitting a KNN regression with $K = 30$, and used holdout method to estimate the test error of the procedure. Now we use the 5-fold CV to do the same. Since we have fixed $K = 30$ (no tuning), there is no inner loop, and the out loop is 5-fild CV.

We again use `caret` as follows.

```
set.seed(1001)
## Set K=30
kgrid <- expand.grid(k = 30)

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

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

```
## k-Nearest Neighbors
##
## 506 samples
##   1 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 404, 405, 406, 404, 405
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.245283  0.6792958  3.773841
##
## Tuning parameter 'k' was held constant at a value of 30
```

```
# test MSE
knn_fit$results$RMSE^2
```

```
## [1] 27.51299
```

The estimate test error is 27.51 for the 30-NN regression fit. The `train()` function, by default, returns RMSE, rather than MSE. The RMSE of of each of the 5 folds can be obtained using the `resample` component of `knn_fit`:

```
knn_fit$resample
```

```
##       RMSE  Rsquared      MAE Resample
## 1 5.410063 0.6572638 3.630137    Fold1
## 2 4.812217 0.7555945 3.615484    Fold2
## 3 5.063268 0.7138431 4.003944    Fold3
## 4 5.822570 0.6101198 4.020697    Fold4
## 5 5.118295 0.6596576 3.598944    Fold5
```

Note that the best reported RMSE is the average of the 5 RMSE values above. If we want the average MSE, we have to perform the computation manually, which is slightly different than computing MSE from the best RMSE.[21]

[21] (Average of RMSE)^2 is not the same as (average of RMSE^2)$^{1/2}$.

```
mean(knn_fit$resample$RMSE^2)
```

```
## [1] 27.63243
```

For the rest of the chapter, we will use RMSE as is default in `caret`.

We can tune hyperparameters using V-fold CV as well:

1. Split the data randomly into V folds.
2. For each fold $\ell = 1, \ldots, V$, do:
   a. Set Fold ℓ as the test set, and the remaining folds together as the training set.
   b. Fit the model using the training set, and evaluate MSE/RMSE[22] using the test set (Fold ℓ), *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 (V of them). The final MSE/RMSE for each of the hyperparameter value is calculated by taking the mean of V MSE/RMSE values from the V 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.

[22] We can use any other performance metric, e.g., MAE, classification accuracy etc. here.

Continuing from the previous example, let us tune K using 5-fold CV, using `caret`. Figure 15 shows the MSE profile for the tuning process.

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

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

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

plot(knn_fit)
```

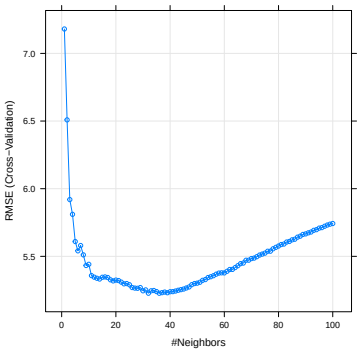


Figure 15: Results from hyperparameter tuning using 5-fold CV.

```
## 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.

The code above does not estimate the test error of the tuned model. If we want to estimate the test MSE/RMSE of the tuned model, we can follow the same strategy as in with holdout method. We can use either holdout or V-fold CV in the outer loop. Unfortunately, `caret` can only perform the inner loop computation for tuning, so we need to manually create the holdout set or the CV folds for the our loop. The easiest way to create folds is to use the `rsample` package and `vfold_cv()` function.

```
folds <- vfold_cv(Boston, v=5)
folds
```

```
## #  5-fold cross-validation
## # A tibble: 5 x 2
##   splits            id
##   <list>            <chr>
## 1 <split [404/102]> Fold1
## 2 <split [405/101]> Fold2
## 3 <split [405/101]> Fold3
## 4 <split [405/101]> Fold4
## 5 <split [405/101]> Fold5
```

The column named `splits` contain the folds. We can access each fold by using the `training()` and `testing()` functions. For example, we can obtain fold 1, and the corresponding training set (the remaining folds) as follows.

```
fold_1 <- testing(folds$splits[[1]])
training_1 <- training(folds$splits[[1]])
dim(fold_1)
```

```
## [1] 102  13
```

```
dim(training_1)
```

```
## [1] 404  13
```

Now we can apply the model fitting/tuning procedure on each of the training set and compute test MSE/RMSE on the folds. The final test error can be estimated by taking average of the MSE/RMSE values form the folds.

```
## Wrap the procedure (including tuning) in a function
tuned_knn_cv <- function(data_split){
  # Input: data_split is a v_fold cv split
  #        obtained using vfold_cv function

  # train and test sets from data splits
  train_set <- training(data_split)
  test_set <- testing(data_split)
  # Set K grid
  kgrid <- expand.grid(k = c(1:100))
  # Training control params
  cv <- trainControl(method = "cv",
                     number = 5)
  # Fit the model on train_set
  knn_fit <- train(medv ~ lstat,
                   data = train_set,
                   method = "knn",
                   tuneGrid = kgrid,
                   trControl = cv)
  # Optimum K and model refit on full train_set
  k_opt <- knn_fit$bestTune$k
  knn_tuned <- train(medv ~ lstat,
                   data = train_set,
                   method = "knn",
                   tuneGrid = expand.grid(k = k_opt),
```

```
                    trControl = trainControl(method = "none"))
  # Predict test_set and compute test_mse
  pred <- predict(knn_tuned,
                  newdata = test_set)
  test_mse <- mean( (test_set$medv - pred)^2 )
  return(test_mse)
}
## Apply the process above to each split
mse_folds <- lapply(folds$splits, tuned_knn_cv)
MSE_test <- mean(unlist(mse_folds))
MSE_test
```

```
## [1] 28.26914
```

An advantage of V-fold CV is that every observation in the data will be used once as a part of test set, and $V-1$ times as a part of training set. Another advantage of V-fold CV is that it provides test MSEs which have much less variability than those from holdout method. To visualize this phenomenon, we repeated the 5-fold CV based tuning process 10 times – the resulting MSE profiles are shown in Figure 16. We can see that the CV estimated MSE values have much less variance compared to holdout method shown in Figure 13.

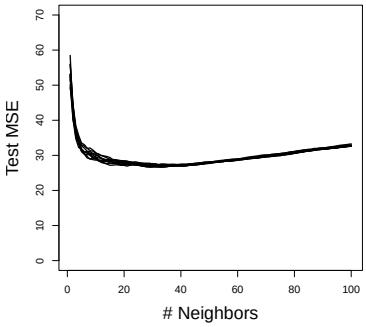


Figure 16: Results from hyperparameter tuning using 5-fold CV, repeated 10 times.

Leave-One-Out Cross-Validation (LOOCV)

As a special case of V-fold cross-validation, consider the case with $V = n$, where n is the sample size of your data. In this case, every observation will be its own fold. Suppose we observe data (Y_i, X_i) for $i = 1, \ldots, n$. The CV then proceeds as follows:

1. For observation (fold) $i = 1, \ldots, n$, do
   - Set the i-th observation (Y_i, X_i) as the test set, and the remaining $n-1$ as the training set.
   - Fit the model on the training set, and predict Y_i (test set)
   - Compute $MSE_i = (Y_i - \widehat{Y}_i)^2$
2. Compute the test MSE as the average of the n MSE values from step 1., that is, $\frac{1}{n}\sum_{i=1}^{n} MSE_i$.

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

There are two advantages of LOOCV over the holdout method. First, the holdout method fits the models on a smaller subset of the full data (e.g., 80% of whole data, even less if another loop/tuning is involved). This may introduce bias in estimation of test error – the holdout method often overestimates the test error due to the fact that

the model is trained using a smaller sample. In contrast, LOOCV trains the model using $n - 1$ observations, which is effectively the entire dataset, and thus reducing estimation bias.

The second advantage of LOOCV is that, there is no random splitting of the data since LOOCV cycles through every observation systematically. Thus results from running LOOCV multiple times will give the same answer, whereas running the holdout method multiple times on the same dataset may give (very) different results.

In `caret` we can specify `method = "LOOCV"` in the `trainControl()` specification to perform LOOCV. Figure 17 shows the MSE profile for tuning K 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)
```

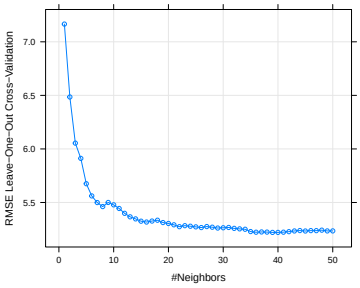


Figure 17: Results from tuning K using LOOCV on the whole Boston data.

A disadvantage of LOOCV is its potential heavy computation cost, especially for large sample size. For example, in `Boston` data ($n = 506$), we have to fit $n - 1 = 505$ models for *each* value of K! This can be extremely difficult for larger n. In contrast, holdout and V-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. Figure 18 shows true test MSE, and the estimates using 10-fold CV and LOOCV for a few simulation scenarios.

We can observe that estimates from 10-fold CV and LOOCV are very similar. However, the quality (bias) of the estimates changes depending on the scenario. On the other hand, even though sometimes the CV estimate underestimate the true test error, the minimizer of

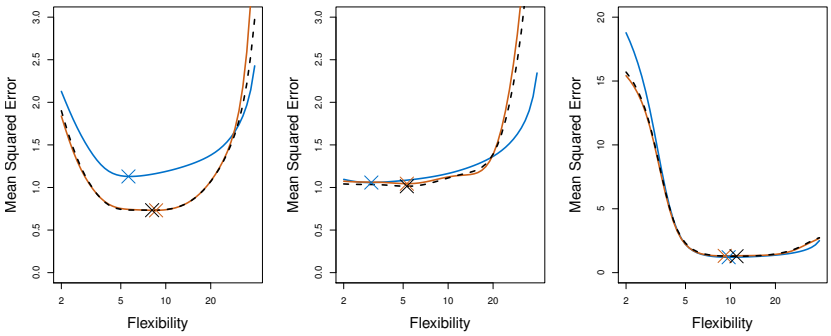


Figure 18: Comparison of CV estimate of test error and its minimizer compared to true test error in several simulation studies. Shown are the true test MSE (blue), LOOCV estimate (black dashed line), and the 10-fold CV estimate (orange), along with their minimum (cross). Figure and caption adapted from Introduction to Statistical Learning, Figure 5.6.

the CV estimates are very close to the minimizer of the true test error. Thus they tend to correctly identify the flexibility (e.g., how small-/large K should be in KNN) of the procedure.

As a final note on cross-validation, the choice of V in V-fold cross-validation depends the bias-variance trade-off[23] of the procedure. Given a sample size of n, the V-fold CV uses approximately $(V-1)n/V$ observation to fit the model. Thus LOOCV effectively uses the whole data to rain the model, and therefore produces almost unbiased estimates of the test error. However, a 5-gold CV might produce a biased estimate. On the other hand, in LOOCV the n model fits essentially uses the same dataset (any two fits share $n-2$ common training observations), the resulting test MSE values are highly correlated. Averaging the n 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.

[23] See Chapter 5.1.4 of *Introduction to Statistical Learning, second edition* for a detailed discussion.

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.

Bootstrap and Out-Of-Bag samples

Bootstrap sample: A random sample drawn with replacement of the original data.

Out-Of-Bag sample: The observations not included in the bootstrap sample.

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.

```
# Bootstrap samples
boot_sample <- bootstraps(Boston, times = 5)
boot_sample
```

```
## # Bootstrap sampling
## # A tibble: 5 x 2
##   splits            id
##   <list>            <chr>
## 1 <split [506/181]> Bootstrap1
## 2 <split [506/176]> Bootstrap2
## 3 <split [506/183]> Bootstrap3
## 4 <split [506/189]> Bootstrap4
## 5 <split [506/186]> Bootstrap5
```

```
# Accessing the bootstrap sample
boot_1 <- training(boot_sample$splits[[1]])
dim(boot_1)
```

```
## [1] 506  13
```

```
## [1] 181  13
```

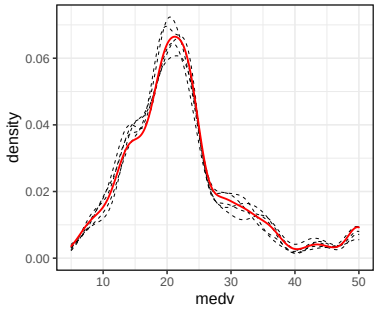


Figure 19: Distribution of 'medv' in the Boston data (red solid line), and in 10 bootstrap samples (black dashed lines).

As with holdout sample, we might want to check whether the distribution of Y in the bootstrap samples is similar to that of the original data. Figure 19 shows distributions of `medv` from 5 bootstrap samples and that of the original data. We can see that the distributions are quite similar.

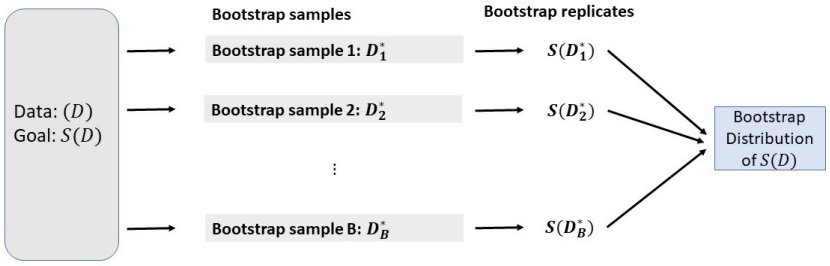
Let us look at the size of the OOB samples that we can use as a test set. We generated 500 bootstrap samples from the `Boston` data –

the percentage of observations that are OOB are shown in Figure 20. On average, we have about 36.72 percent of observations are OOB.[24]

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

- Draw B bootstrap samples from the original data, call them $\mathcal{D}_1^*, \ldots, \mathcal{D}_B^*$.
- For $b = 1, \ldots, B$, do
  - Use the b-th bootstrap sample, $\mathcal{D}_b^*$ to compute the same quantity you computed based on the original data, $S(\mathcal{D}_b^*)$. 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 $S(\mathcal{D}_1^*), \ldots, S(\mathcal{D}_B^*)$ to assess properties of $S(\mathcal{D})$.

Figure 21 shows a layout of using bootstrap as described above.



For example, we can examine the distribution of $S(D)$ by estimating it by using the bootstrap replicates $S(\mathcal{D}_1^*), \ldots, S(\mathcal{D}_B)$ (e.g., a histogram or a density estimate). We can estimate the variance of $S(D)$ using the sample variance of the replicates:

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

where $S^* = \sum_{b=1}^{B} S(\mathcal{D}_b^*)/B$ is the sample mean of the bootstrap replicates.

Consider the example of fitting KNN regression to `Boston` data with fixed $K = 30$.

[24] Interested readers: can you verify this number theoretically? Think about the probability that the i-th observation being included in a typical bootstrap sample. Then see how this probability changes for different values of sample size n.

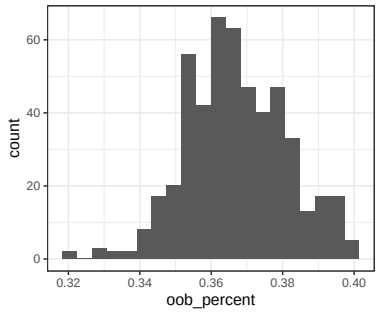


Figure 20: Percent of original observations in OOB sample.

Figure 21: Layout of bootstrap procedure.

```
knn_k30 <- knnreg(medv ~ lstat,
                  data = Boston,
                  k = 30)
```

Suppose we want to estimate $f(x)$ when $x = 5$, that is, expected value of `medv` when `lstat = 5`. The estimation is shown below.

```
pred_k30 <- predict(knn_k30,
                    newdata = data.frame(lstat = 5))
pred_k30
```

```
## [1] 31.81
```

Note that, predicted value of Y when $X = 5$ is the same $\widehat{f}(5)$, the estimated value of $f(x)$ when $x = 5$.[25] What is the standard error of this estimate? What is the distribution of the estimator? We can use bootstrap to answer these questions.

We will draw 200 bootstrap samples from `Boston` data. For each bootstrap sample, we will fit the KNN procedure with $K = 30$, and compute the estimate – this is all according to Figure 21.

[25] Even though the predicted Y and estimated $f(5)$ values are the same, their variability is not the same. Recall, variability of the prediction is represented by expected prediction error at $x = 5$ is $\text{bias}^2(\widehat{f}(5)) + var(\widehat{f}(5)) +$ irreducible error. In this case, we are only interested in $var(\widehat{f}(5))$.

```
## 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)
  # KNN 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
boot_pred <- sapply(boot_sample$splits, knn_k30_predict)
```

Figure 22 shows the bootstrap distribution of $\widehat{f}(5)$. Some summaries of the bootstrap estimates are shown below.

```
## Summary of bootstrap estimates
summary(boot_pred)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   28.25   30.33   31.28   31.28   32.21   34.84
```

```
## Variance/SD of the estimate
c(variance = var(boot_pred),
  sdev = sd(boot_pred))
```

```
## variance     sdev
## 1.623432 1.274140
```

```
## MSE
mean( (boot_pred - pred_k30)^2 )
```

```
## [1] 1.891878
```

In a learning method, we can tune hyperparameters using bootstrap as before – 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)

plot(boot_tuned_knn)
```

Figure 23 shows the RMSE profile for tuning K using bootstrap.

Compared to V-fold cross-validation, bootstrap tends to produce less variable estimates. However, on average only 63.2% observations get represented in bootstrap samples. Thus bootstrap estimates may have some bias similar to using a 2-fold or 3-fold CV.

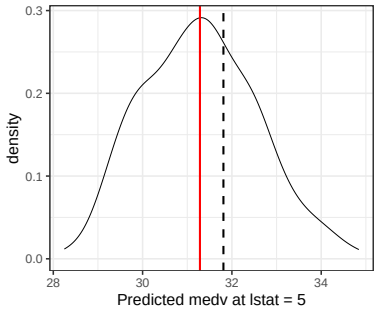


Figure 22: Distribution of estimator of E(medv when lstst = 5). Also shown the mean of the bootstrap estimates (red solid lile), and original estimate from the full data (black dashed line),

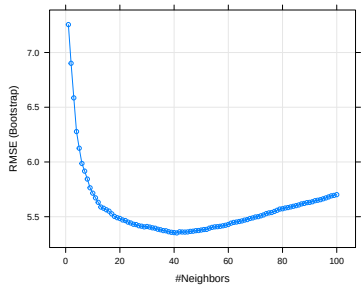


Figure 23: Results from bootstrap (25 reps) tuning of K.

K-Nearest Neighbors Classification

The discussion and techniques presented so far also apply to classification setting, the main difference is model evaluation metric. To demonstrate the ideas clearly, let us introduce a simple classification technique based of K-nearest neighbors.

Suppose we have training data (Y_i, X_i) for $i = 1, \ldots, n$, where Y_i is categorical variable denoting class label of X_i. For a given predictor x_0, KNN classifier predicts the class label as follows:

- Identify the K observations in the training data such the their X values are "nearest" to x_0.[26]
- Predict the class label corresponding to x_0 as the class having the majority vote, that is, having the most number of points among the K neighbors obtained form previous step.

[26] See the discussion about distance metric in KNN regression section.

Formally, we can think of the process as estimating the conditional probability of $P(Y|X)$. Suppose that we have J classes, that is, Y can take values $1, \ldots, J$.[27] suppose $S_K(x_0)$ denotes the indices of the K points whose X values are nearest to x_0. Then for a data point x_0, KNN estimates the conditional probability that the class label is j given $X = x_0$ as[28]

$$\widehat{P}(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in S_K(x_0)} I(Y_i = j),$$

for each $j = 1, \ldots, J$. Thus, for each of the J classes, we compute the proportion of the K neighbors belonging to that class. We classify x_0 to the class that has the highest estimated probability.

[27] These are not numeric values. They are merely labels for J classes.

[28] Here $I(\cdot)$ denotes the indicator function. For any event A, we define $I(A) = 1$ if A is true, 0 otherwise.

Bayes classifier

The motivation behind estimating the conditional probabilities $P(Y = j|X)$ is from minimizing test error rate. Similar to regression, given a new independent test point X with label Y, we can define expected prediction error for classification as

$$E[I(Y \neq \widehat{Y})],$$

where $\widehat{Y}$ is the prediction from a classifier. Notice that Y depends on X, and $\widehat{Y}$ depends on both X and the training set. We want a classifer that minimizes the expected prediction error. It can be shown[29] that the optimal classifier is the one that predicts a new observation x_0 by $\widehat{Y}$ such that

> $\widehat{Y} = j$ if $P(Y = j|X = x_0)$ is maximum among $P(Y = 1|X = x_0), \ldots, P(Y = J|X = x_0)$.

[29] Interested readers can consult *Elements of Statistical Learning* by Hastie et al. (2017).

The optimal classifier is called the *Bayes classifier*.

Bayes classifier and error

Bayes Classifier: Classifies an observation to the most probable class using the discrete conditional distribution of $P(Y|X)$

Bayes error rate: misclassification error rate of the Bayes classifier. For a given x_0, Bayes error is $1 - max_\ell P(Y = \ell | X = x_0)$. The overall Bayes rate is $1 - E[max_\ell P(Y = \ell | X)]$.

Thus every classification problem has a corresponding Bayes classification rule and associated Bayes error rate. The Bayes rate is analogous to the irreducible error that we encountered in the regression setting.

Unfortunately, we can not directly use the Bayes classifier since we do not know the distribution of $Y|X$. Different classifiers use different estimators of such conditional distributions – KNN uses proportion of points in the K nearest neighbors belonging to each class as the estimator, as discussed before.

Evaluating a classifier

To evaluate the performance of the classifier, instead of test MSE, we can use *classification accuracy* or *misclassification error rate*.[30]

[30] In the definitions below, $|S|$ denotes the *cardinality* of S, that is, the number of observations in S.

Accuracy/error rate of a classifier

Accuracy: the proportion of points correctly classified to their respective classes. With a set of observations S,

$$\text{Accuracy} = \frac{\text{Total correct classification}}{\text{Total number of points}} = \frac{1}{|S|} \sum_{i \in S} I(Y_i = \widehat{Y}_i).$$

We can compute *training* and *test accuracy* depending on whether S is the training or testing set.

Misclassification error rate: The proportion of points wrongly classified.

$$\text{Error rate} = \frac{\text{Total incorrect classification}}{\text{Total number of points}} = \frac{1}{|S|} \sum_{i \in S} I(Y_i \neq \widehat{Y}_i).$$

As before, we can compute *training* and *test error rate*.

As with regression setting, here too we aim to maximize *test* accuracy or minimize test error. minimizing training error is undesirable since it will lead to overfitting the data. For example, consider $K = 1$,

a 1-NN classifier. Since each X_i is the closest neighbor to itself, the training error would be zero. Figure 24 shows training and test error rates from a simulation study (figure adapted from the textbook *Introduction to Statistical Learning*).

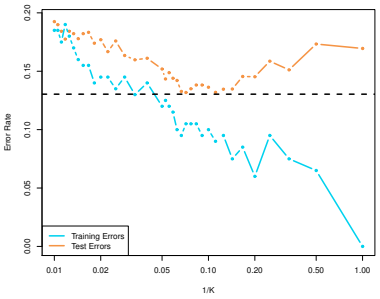


Figure 24: Training and test error rates for a KNN classifier based on 200 training and 5000 test observations. The error rates are plotted against 1/K. The black dashed line shows the Bayes error rate. Figure adapted from Introduction to Statistical Learning.

Role of hyperparameter in classification

As with KNN regression, the hyperparameter K determines how flexible the KNN method is. However, the idea of flexibility is subtle in this case. Consider a two class problem – a classification problem with two classes. A classifier will attempt to create regions using the predictors so that a new data point could be classified into a class depending on which region it fall into. The boundary that separates the these regions is effectively the classification rule for that classifier. Figure 25 shows a classification problem with two classes. A certain classifier creates two regions (red and blue) so that a new data point will be classified to red/blue classes if it falls in the corresponding region. The *decision boundary*, is the boundary of the regions, a straight line in this example.

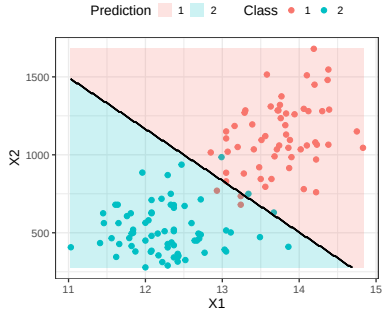


Figure 25: Two-dimensional classification.

How would the decision boundary look like for the Bayes classifier? For a two class problem, the Bayes classifier assigns the most probable class to a new data point. Thus for a new x_0, it will be assigned to class 1 if $P(Y = 1|X = x_0) > P(Y = 2|X = x_0)$, assign to class otherwise. Equivalently, x_0 will be assigned to class 1 if $P(Y = 1|X = x_0) > 0.5$. Thus the decision boundary of the Bayes classifier is the set of all x such that $P(Y = 1|X = x) = 0.5$.

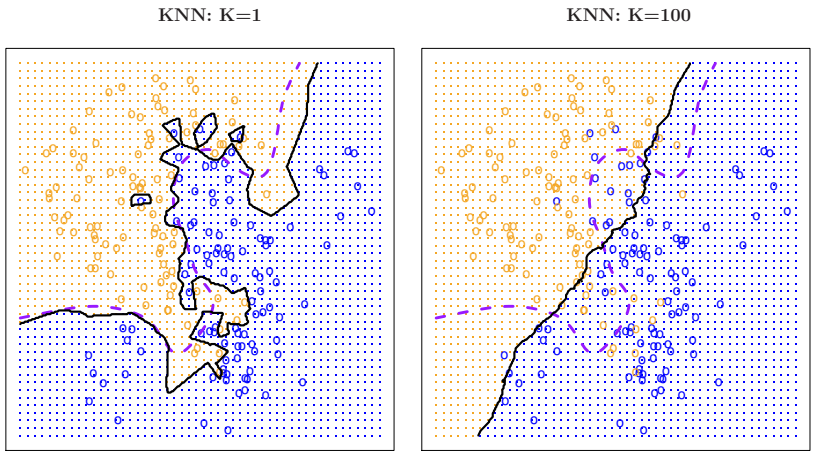


Figure 26: Impact of K on decision boundaries of KNN classifiers. The Bayes dicision boundary is shown using purple dashed line. Image adapted from *Introduction to Statistical Learning*.

The value of K in a KNN classifier determines how smooth or rough the decision boundary is (this is analogous to estimating $f(\cdot)$ in a regression problem). Figure 26 shows decision boundaries for

KNN classifier for different values of K in a simulated data along with that of the Bayes classifier. For small value of K (in this example, $K = 1$), the boundary is extremely rough. Although it follows the Bayes boundary closely, it is overly flexible (uses local features) and tries to discover patterns that do not conform to the Bayes boundary. This is an example of overfitting a classification problem. In contrast, for a large value of K (such as $K = 100$), the decision boundary is much smoother but does not capture the shape of the Bayes boundary.[31] Large value of K results in a non-flexible (uses global features but averages over local ones) classifier that perhaps captures the overall trend of the Bayes boundary, but misses the details. In fact, as K grows, the decision boundary will get closer to a straight line.

Therefore, we need to tune K so that the "optimal" K will result in a decision boundary that is not too rough but also sufficiently captures the shape of the Bayes boundary. Firgure 27 shows one such example with $K = 10$. In practice, we might choose K by minimizing the test error rate or equivalently maximizing test accuracy.

[31] Again we see the bias-variance trade-off here.

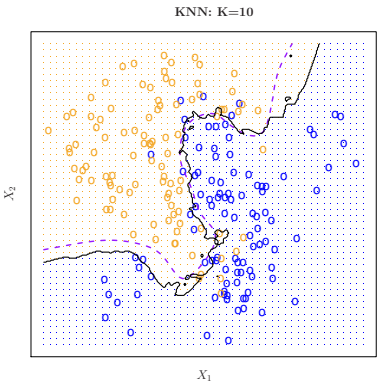


Figure 27: Decision boundary for $K = 10$ using simulated data presented in Figure 26. The Bayes dicision boundary is shown using purple dashed line. Image adapted from *Introduction to Statistical Learning*.

Building a classifier

Consider the `wines` data set available at the UCI machine learning repository.[32]. The dataset contains quantities of 13 constituents found in each of the three types (cultivars) of wines.

[32] `https://archive.ics.uci.edu/ml/datasets/wine`; also available with the textbook Applied Multivariate Statistics with R by Zelterman

```
# Read the data
wines <- read.table("data/Wines.txt", header = TRUE)
wines$Class <- as.factor(wines$Class)
```

A snapshot of the full data is shown below. The goal is to find a *rule* that can assign a specimen of wine to its region. In other words, we want to predict the classes (regions) based on the predictors (13 variables).

```
## # A tibble: 178 x 14
##    Class Alcohol Malic   Ash Alcal    Mg Phenol  Flav  Nonf Proan Color   Hue
##    <fct>   <dbl> <dbl> <dbl> <dbl> <int>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 1        14.2  1.71  2.43  15.6   127   2.8   3.06  0.28  2.29  5.64  1.04
##  2 1        13.2  1.78  2.14  11.2   100   2.65  2.76  0.26  1.28  4.38  1.05
##  3 1        13.2  2.36  2.67  18.6   101   2.8   3.24  0.3   2.81  5.68  1.03
##  4 1        14.4  1.95  2.5   16.8   113   3.85  3.49  0.24  2.18  7.8   0.86
##  5 1        13.2  2.59  2.87  21     118   2.8   2.69  0.39  1.82  4.32  1.04
##  6 1        14.2  1.76  2.45  15.2   112   3.27  3.39  0.34  1.97  6.75  1.05
##  7 1        14.4  1.87  2.45  14.6    96   2.5   2.52  0.3   1.98  5.25  1.02
##  8 1        14.1  2.15  2.61  17.6   121   2.6   2.51  0.31  1.25  5.05  1.06
##  9 1        14.8  1.64  2.17  14      97   2.8   2.98  0.29  1.98  5.2   1.08
```

```
## 10 1        13.9  1.35  2.27  16      98   2.98  3.15  0.22  1.85  7.22  1.01
## # ... with 168 more rows, and 2 more variables: Abs <dbl>, Proline <int>
```

```
# classes of wine
table(wines$Class)
```

```
##
##  1  2  3
## 59 71 48
```

For this demonstration, we will only consider two predictor variables, `Alcohol` and `Malic`. However, the techniques discussed hereafter can be applied to any number of predictors. Figure 28 shows the three classes on a scatterplot of `Alcohol` vs. `Malic`.

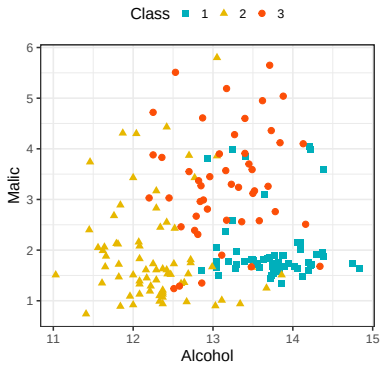


Figure 28: Scatterplot of Alcohol vs. Malic in the wine data.

Let us start with a KNN classifier with $K = 30$. We can use the `knn()` function in the `class` package, or use `caret` with `method = "knn"`. Note that `caret` does both regression and classification. It automatically determines the problem depending on whether the response is numeric or categorical (factor). We have already converted the `Class` variable in the `wines` data to a factor.

```
## 30-NN Classifier / no tuning needed
fit <- train(Class ~ Alcohol + Malic,
             data = wines,
             method = "knn",
             tuneGrid = expand.grid(k = 30),
             trControl = trainControl(method = "none"))
fit
```

```
## k-Nearest Neighbors
##
## 178 samples
##   2 predictor
##   3 classes: '1', '2', '3'
##
## No pre-processing
## Resampling: None
```

Figure 29 shows the decision boundaries of this classifier.

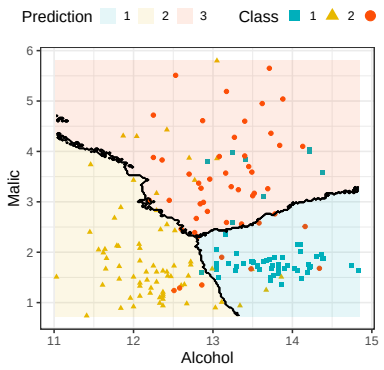


Figure 29: Decision boundary of 20-NN classifier of the wines data.

We can tune K as we did in the regression setting. The code below searches odd values of K (to avoid ties) for the optimal value with largest test accuracy. We use 50 times repeated 5-fold CV for tuning – Figure 30 shows the results.

```
set.seed(1001)
## K values for tuning
kgrid <- expand.grid(k = seq(1,51, by=2))
## LOOCV tuning
tr <- trainControl(method = "repeatedcv",
                   number = 5,
                   repeats = 50)
## Train the classifier
fit <- train(Class ~ Alcohol + Malic,
             data = wines,
             method = "knn",
             tuneGrid = kgrid,
             trControl = tr)
plot(fit)
```

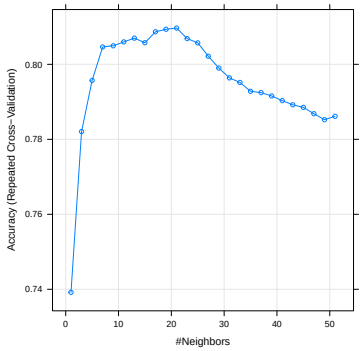


Figure 30: Results for repeated 5-fild CV tuning.

```
fit$bestTune$k
```

```
## [1] 21
```

```
## Refit the model with best K
tuned_knn_class <- train(Class ~ Alcohol + Malic,
             data = wines,
             method = "knn",
             tuneGrid = expand.grid(k = fit$bestTune$k),
             trControl = trainControl(method = "none"))
```

To estimate the prediction error of the tuned model, we can use any of the methods discussed previously. For example, you can use bootstrap as the outer loop while the inner loop uses LOOCV for tuning K.

```
new_dat <- data.frame(Alcohol = c(13, 12.78),
                      Malic = c(3,2))
new_dat
```

```
##   Alcohol Malic
## 1   13.00     3
## 2   12.78     2
```

We can predict classes of new unlabeled data. Let us consider two new data points – Figure 31 shows the original data, along with the two new points (black). We can predict their classes as follows.

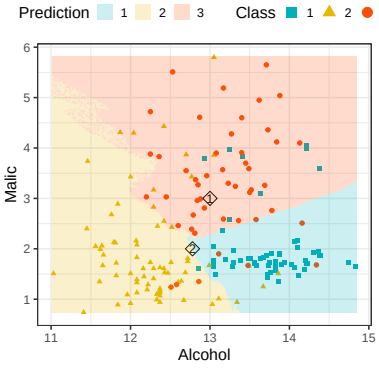


Figure 31: Decision boundary of 21-NN classifier of the wines data with two new unlabeled points.

```
pred_class <- predict(tuned_knn_class,
                      newdata = new_dat)
pred_class
```

```
## [1] 3 2
## Levels: 1 2 3
```

The first point (first row of `new_dat`) was classified into Class `3`, and second one to class `2`. The issue with just obtaining the final predicted class is that we do not know how sure we are about these predictions. In addition, we often look at the class probabilities for each new data. We can specify `type = "prob"` to do so.

```
pred_prob <- predict(tuned_knn_class,
                     newdata = new_dat,
                     type = "prob")
pred_prob
```

```
##           1          2         3
## 1 0.1428571 0.04761905 0.8095238
## 2 0.3809524 0.42857143 0.1904762
```

Note that for the first point, has as 80% probability associated with class `3`, and hence we are quite confident about out final class prediction of `3`. However, for the second point, probabilities associated with classes `1` and `2` are quite similar (38% vs 43%). So while we are quite confident about the predicted class of the first data point, there is some uncertainlty about the second prediction.

Evaluating Predicted Classes

There are many other metrics to evaluate a classification technique other than error rate and accuracy. The main criticism of these two criteria are that they provide a global measure, but do not provide much insight into how individual classes are correctly identified. For example,80% accuracy of a classifier does *not* guaranty that it will correctly classify items form *both* the classes correctly 80 of times. Such a criticism is even more relevant when there is class imbalance in the data: say we have a situation where 80 of observations belong to class A, and rest in class B. If we employ a classifier that classifies *every point into class A* regardless of their predictor values. This classifier will have 80% accuracy! This is called the *no information rate (NIR)* of the classification problem.

No information rate (NIR)

The percentage of the largest class in the training set.

The NIR represents the accuracy that can be obtained without using any model. Thus for any classifier, no information rate should be the minimum accuracy it should have. Any classifier having accuracy better than NIR might be considered viable.

For a problems with two classes (say "positive" = 1 and "negative" = 2), most of the measures to evaluate a classifier can be obtained by cross-tabulating the true and predicted classes of a test set. Such a table is called *confusion matrix*. An example is shown in Table 1.

Table 1: Example of a confusion matrix

	True		
Predicted	1	2	-SUM-
1	57	7	7
2	2	64	2
-SUM-	2	7	9

In general, for a two class problem, the confusion table looks like 2 (I have deleted the `-SUM-` column/row) .

Table 2: General confusion matrix for a two class problem.

	True	
Predicted	Positive	Negative
Positive	True positive (TP)	False positive (FP)
Negative	False negative (FN)	True negative (TN)

Some measures we might look at are as follows:

- *sensitivity*[33] $= \frac{\text{number of positive cases classified as positive}}{\text{Total number of positive samples}} = \frac{TP}{TP+FN}$
- *specificity*[34] $= \frac{\text{number of negative cases classified as negative}}{\text{Total number of negative samples}} = \frac{TN}{TN+FP}$
- *Precision* $= \frac{\text{number of positive cases classified as positive}}{\text{Total number of predicted positive cases}} = \frac{TP}{TP+FP}$

[33] Also called "true positive rate" or "recall"

[34] Also called "true negative rate"

We can also examine:

- *Cohen's kappa*:[35] measures the agreement of the classifier to the sample data taking into account any class imbalances, and how much agreement is by chance. Values close to 1 are considered good. The R function to do so is `cohen.kappa()` in `psych` library.
- *McNemar's test*:[36] hypothesis test for agreement between the predictions from an classifier to the observed data using a Chi-squared test. The R function to do so is `mcnemar.test()`.

[35] Cohen, Jacob (1960). "A coefficient of agreement for nominal scales". Educational and Psychological Measurement. 20 (1): 37–46.

[36] Alan Agresti (1990). Categorical data analysis. New York: Wiley. Pages 350–354.

For a multi-class problem, we can create these measures using a "one-vs-all" approach, that is, by comparing each class vs the remaining combined (class 1 vs not class 1, and so on).

Just for demonstration, let us calculate the above mentioned measures using the wines data set, that is, the training set.

```
## Confusion matrix and other measures
pred_class <- predict(tuned_knn_class,
                      newdata = wines)
confusionMatrix(data = wines$Class,
                reference = pred_class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 50  1  8
##          2  5 56 10
##          3  5  3 40
##
## Overall Statistics
##
##                Accuracy : 0.8202
##                  95% CI : (0.7558, 0.8737)
##     No Information Rate : 0.3371
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.73
##
##  Mcnemar's Test P-Value : 0.06792
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            0.8333   0.9333   0.6897
## Specificity            0.9237   0.8729   0.9333
## Pos Pred Value         0.8475   0.7887   0.8333
## Neg Pred Value         0.9160   0.9626   0.8615
## Prevalence             0.3371   0.3371   0.3258
## Detection Rate         0.2809   0.3146   0.2247
## Detection Prevalence   0.3315   0.3989   0.2697
## Balanced Accuracy      0.8785   0.9031   0.8115
```

We should keep in mind that the results might have large bias since we are using the same data to build our model as well as test it. We should not put to much emphasis on high accuracy we see here. Ideally, we would create a test set, or perform re-sampling to properly measure performance.

Often, we want a single measure of performance of the classifier rather than the multitude of measures shown above. There are many such options, such as *Youden's J Index*,

$$J = \text{Sensitivity} + \text{Specificity} - 1,$$

which measures the proportions of correct predictions for both the positive and negative events.

Evaluating predicted probabilities via ROC curves

Another approach to combine sensitivity and specificity is to investigate the Receiver Operating Characteristic (ROC) curves. Consider a two-class classification problem. According to the Bayes rule, we assign a observation, x, to class 1 if $P(Y = 1|X = x) > 0.5$, assign to class 2 otherwise. However, is the cutoff 0.5 reasonable all the time? Sometimes using the default cutoff of 0.5 results in loss of sensitivity/specificity, and changing the cutoff might increase the class-specific performance of the classifier. The ROC curve can be used to determine other cutoff values for class probabilities.

We calculate the ROC by using a set of cutoff values in a continuum. For each of the cutoff values, we calculate the sensitivity (the true-positive rate) and 1− specificity (the false-positive rate). These quantities are then plotted against each other. The resulting curve is the ROC curve. Keep in mind that changing the cutoff values gives us either more positive or negative classifications – it can not reduce false positive and false negative simultaneously. Thus for a cutoff, if sensitivity increases the specificity most likely decreases.

We can use libraries such as `pROC`, `ROCR`, `caTools`, `PresenceAbsence`, and many others to produce ROC curve and statistics. Let us consider a two class problem (wines data with only classes 1 and 2). Figure 32 shows the predicted probabilities of an item belonging to class 1, that is, $P(Y = 1|X)$ for items of each of the true classes. In this case, the two distributions do not have much overlap.

```
# Create a two-class problem
wines <- read.table("data/Wines.txt", header = TRUE)
new_wine <- wines %>% filter(Class == 1 | Class == 2)
new_wine$Class <- as.factor(new_wine$Class)
# 20-NN for demonstration
knn_k20 <- train(Class ~ Alcohol + Malic,
                 data = new_wine,
                 method = "knn",
                 trControl = trainControl(method = "none"),
                 tuneGrid = expand.grid(k = 20))
```

```
# Prediction on the training data
pred_wine <- predict(knn_k20,
                     newdata = new_wine,
                     type = "prob")
```

```
# ROC
library(pROC)
library(ggplot2)

roccurve <- roc(response = new_wine$Class,
                predictor = pred_wine[,2])

ggroc(roccurve, legacy.axes = TRUE, lwd=2) +
  theme_bw(base_size = 18)
```

```
# Can also use:
# plot(roccurve, legacy.axes = TRUE)
```

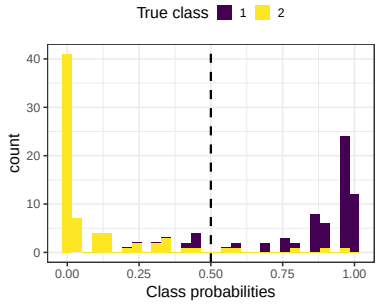


Figure 32: Predited class probabilities of the two classes.

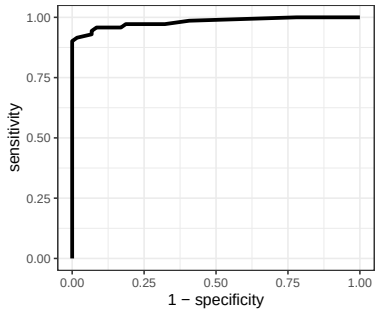


Figure 33: ROC curve of the wine data with two classes.

A perfect classifier will have both sensitivity and specificity value 1, that is, there will be no misclassification error (perfect separation between the classes). In contrast, a "random guess" classifier will distribute the observations into two classes randomly, leading to a diagonal ROC curve. The corresponding ROC curves are shown in Figure 34. Thus a classifier can be evaluated based on how close its ROC curve is to the perfect ROC curve. A single measure is the area under the ROC curve (AUC). Large AUC values are associated with better classifier (since the ROC curves are closer to the perfect ROC curve, which has AUC 1.)

```
auc(roccurve)
```

```
## Area under the curve: 0.9813
```

A disadvantage of AUC is that we lose information about the ROC curve if we just use AUC. Often, given multiple classifiers, a single ROC curve might not be uniformly better than all others, and the curves can cross. Such patterns are suppressed if we only look at AUC.

We can use ROC curves to visually compare different models. Such comparisons include investigating different set of covariate in the same model, choice of different hyperparameters (i.e., comparing different K values in KNN), or between different classifiers.

Keep in mind, ROC curve and AUC are still statistics, i.e., summaries of the data. As such, we should compute them on the test

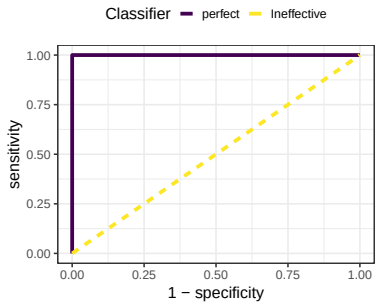


Figure 34: ROC curve of a perfect and a completely ineffective classifier.

set(s) to avoid getting overly optimistic performance due to overfitting.

Summary

In this chapter we discussed the following main concepts.

- K-nearest neighbors methods: regression and classification.
- Evaluation metrics: MSE/RMSE for regression, Accuracy/misclassification error for classification.
- Bias-variance trade-off in relation to model flexibility.
- Irreducible error (regression) and Bayes error rate (classification), training and test MSE/error.
- Data splitting methods: Holdout, V-fold CV, Leave-One-Out CV, Bootstrap.
- Hyperparameter tuning methods.
- Test error estimation methods.

We have used R packages `caret` and `rsample` for the most parts. Many of the plots used in this chapter were created using `ggplot2` package (code not shown).