Pure Prediction: KNN and SVM

36-600

Fall 2022

KNN: K Nearest Neighbors

In words: KNN examines the k data points closest to a location x and uses just those data to generate predictions.

KNN straddles the boundary between fully parameterized models like linear regression and fully data-driven models like random forest. A KNN model is data-driven, but one *can* actually write down a compact parametric form for it *a priori*:

- For regression:

  $$\hat{Y}|\mathbf{x} = \frac{1}{k}\sum_{i=1}^k Y_i \,,$$

  where the summation is over the points $\mathbf{x}_1,\ldots,\mathbf{x}_k$ that are "closest" to $\mathbf{x}$ (usually in a Euclidean sense).

- Classification involves the same summation, with a twist:

  $$P[Y = j|\mathbf{x}] = \frac{1}{k}\sum_{i=1}^k \mathbb{I}(Y_i = j) \,,$$

  where $\mathbb{I}(\cdot)$ is the indicator function. It returns 0 if the argument is false, and 1 otherwise. The summation yields the proportion of neighbors that are of class j.

Finding the Optimal Number of Neighbors

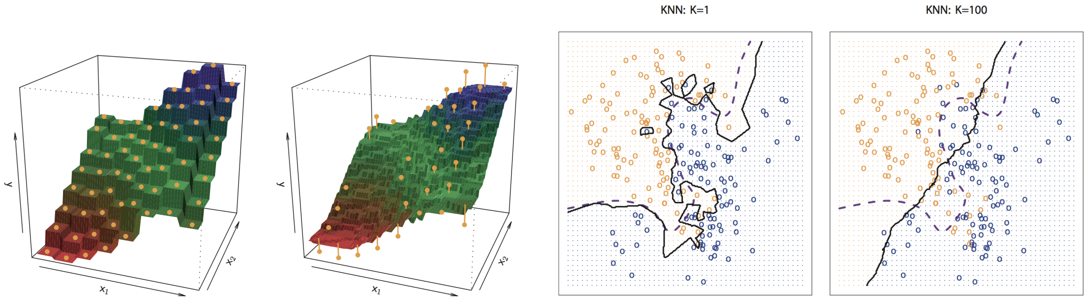
For KNN, the number of neighbors k is a *tuning parameter*. This means that in theory, in addition to splitting data into training and test datasets, *the training data themselves have to be split into a smaller training set and a validation set*. For each value of k, you train on the smaller training set, and compute MSE or MCR using the validation set. Once the optimum value of k is determined, you re-run the model using the *full* (unsplit) training set, and assess the model using the test data. (Again: the test data are never used when learning/tuning the model!)

Luckily for you, however, the `FNN` package in `R` does cross-validation on the training set for you, under the hood…you do not need to code this yourself!

Finding the Optimal Number of Neighbors

Note that if…

- k is too small, the resulting model is *too flexible*, which means it has low bias (it is right on average) but high variance (the predictions are more uncertain). See the panels to the left below.
- k is too large, the resulting model is *not flexible enough*, resulting in high bias (wrong on average) and low variance (nearly the same predictions, every time). See the panels to the right below.



(Figures 3.16 and 2.16, *Introduction to Statistical Learning* by James et al.)

KNN in Context

Here are two quotes from the *ISLR* textbook to keep in mind when thinking about KNN:

- "As a general rule, parametric methods [like linear regression] will tend to outperform non-parametric approaches [like KNN] when there is a small number of observations per predictor." This is the *curse of dimensionality*: for data-driven models, the amount of data you need to get similar model performance goes up exponentially with p.

$\Rightarrow$ KNN might not be a good model to learn when the number of predictor variables is large.

- "Even in problems in which the dimension is small, we might prefer linear regression to KNN from an interpretability standpoint. If the test MSE of KNN is only slightly lower than that of linear regression, we might be willing to forego a little bit of prediction accuracy for the sake of a simple model…"

$\Rightarrow$ KNN is not the best model to learn if inference is the goal of an analysis.

KNN: Three Critical Points to Remember

- To determine which neighbors are the nearest neighbors, pairwise (Euclidean) distances are computed…so we need to scale (or standardize) the individual predictor variables.
- Since it utilizes distances between data, *KNN only works with quantitative predictor variables*.
- Don't blindly compute a pairwise distance matrix! For instance, if n = 100,000, then your pairwise distance matrix will have 10^{10} elements, each of which uses 8 bytes in memory…resulting in a memory usage of 80 GB! Your laptop cannot handle this. It can barely handle a few GB at this point. If n is large, you have four options:
    - subsample your data, limiting n to be $\lesssim$ 15,000-20,000;
    - use a variant of KNN that works with sparse matrices (matrices that can be compressed since most values are zero);
    - make use of a "kd tree" to more effectively (but only approximately) identify nearest neighbors; or
    - find a computer system with 32 GB or 64 GB (or more) of memory.

The `FNN` package in `R` has an option to search for neighbors via the use of a kd tree. Always apply this option if you have more than, e.g., 10,000 rows in your data frame. Otherwise you should be safe to go with the default `brute` algorithm.

KNN: Regression Example

We input and split a 10,000-galaxy dataset that we will use to predict distances of galaxies from the Earth. Note that we scale the predictor variables (but not the response), using the `scale()` function.

```r
library(FNN)

k.max = 50
mse.k = rep(NA,k.max)
for ( kk in 1:k.max ) {
  knn.out = knn.reg(train=pred.train,y=resp.train,k=kk,algorithm="brute") # 7000 galaxies - brute OK
  mse.k[kk] = mean((knn.out$pred-resp.train)^2)
}
k.min = which.min(mse.k)
cat("The optimal number of nearest neighbors is ",k.min,"\n")
```

```
## The optimal number of nearest neighbors is  11
```

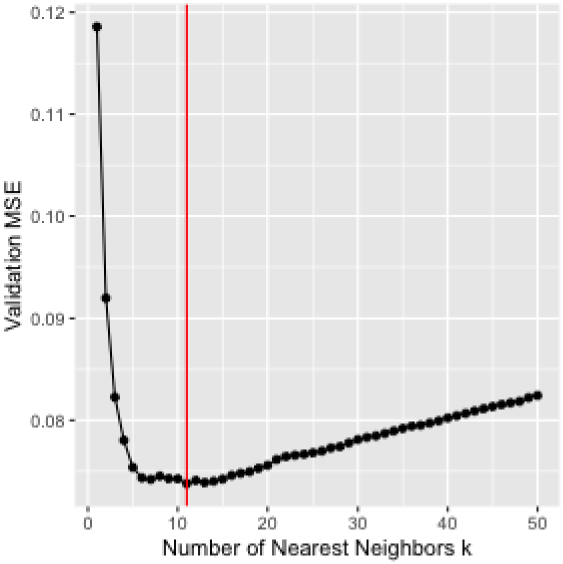
Note: if the optimal number of nearest neighbors is $k_{\rm max}$, then your value of $k_{\rm max}$ is too small. Increase it and start over!

Also note how we use a `for`-loop, and compute and save the MSE for each value of k. See the next slide.

KNN: Regression Example

```
suppressMessages(library(tidyverse))

ggplot(data=data.frame("k"=1:k.max,"mse"=mse.k),mapping=aes(x=k,y=mse)) +
  geom_point() + geom_line() +
  xlab("Number of Nearest Neighbors k") + ylab("Validation MSE") +
  geom_vline(xintercept=k.min,color="red")
```



To be clear: we are still at the training stage here, attempting to determine the optimal value of k, which we dub `k.min`. We are not recording the minimum value of the validation-set MSE…we simply want to determine the value of k for which this quantity is minimzed.

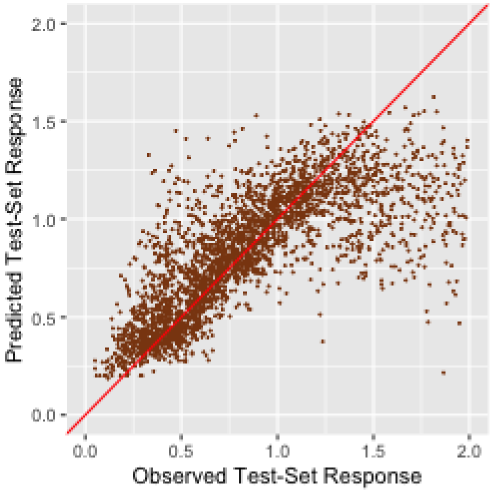
KNN: Regression Example

Now that we have `k.min`, we generate test-set predictions and generate the test-set MSE.

```
knn.out = knn.reg(train=pred.train,test=pred.test,y=resp.train,k=k.min,algorithm="brute")
(knn.mse = mean((knn.out$pred-resp.test)^2))
```

```
## [1] 0.07574588
```

```
ggplot(data=data.frame("x"=resp.test,"y"=knn.out$pred),mapping=aes(x=x,y=y)) +
  geom_point(size=0.1,color="saddlebrown") +
  xlim(0,2) + ylim(0,2) +
  xlab("Observed Test-Set Response") + ylab("Predicted Test-Set Response") +
  geom_abline(intercept=0,slope=1,color="red")
```



KNN: Prediction in Classification Models

Note that for classification:

- instead of using `knn.reg()`, you would use `knn.cv()` to determine the optimum value of k; and
- you would use `knn()` to generate predictions on the test set.

Note the `cl` argument for both: this is where `resp.train` goes (since `cl` means "class"). So change `y` to `cl`.

`knn()` will output class predictions (based on majority vote!), unless you add the argument `prob=TRUE` to the call. If you want to compute, e.g., a ROC curve for a KNN classifier, use the (pseudo-)code below to extract the Class 1 probabilities. Once you have those, you can create the ROC curve, or a confusion matrix for a specific class-separation threshold, etc., etc.

```
knn.out = knn(...,prob=TRUE)
knn.prob = attributes(knn.out)$prob
w = which(knn.out=="<class 0>") # insert name of Class 0 here
knn.prob[w] = 1 - knn.prob[w]   # knn.prob is now the Class 1 probability!
```

SVM: Support-Vector Machine

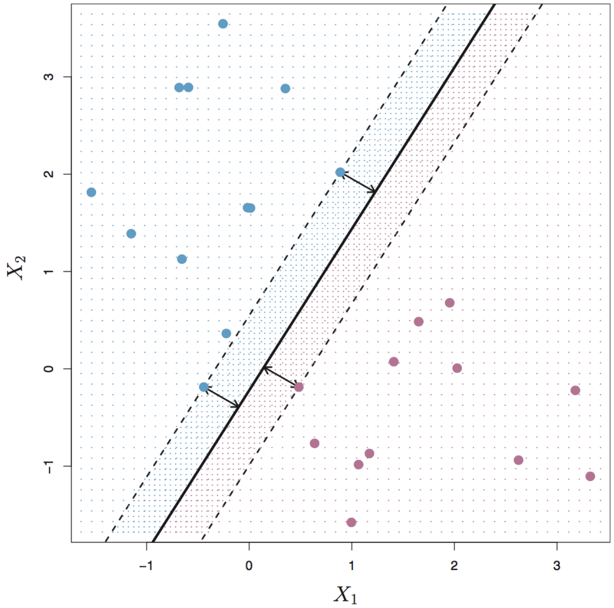
A *support vector machine* is an enigmatically named machine learning algorithm for most often used for classification.

In theory (but not in practice…because of algorithmic cleverness), SVM transforms predictor data into a *higher-dimensional space* and in that space constructs a linear boundary that optimally separates instances of two classes. (Note that SVM is not designed to tackle analyses in which the response has more than two classes!)

- Like a KNN model, an SVM model is a purely predictive model, useless for inference…
- …and like a KNN model, an SVM model utilizes distances between data, so scaling the predictor data is required…
- …and like a KNN model, an SVM model works only with quantitative predictor variables.

The *ISLR* textbook gives a somewhat mathematical description of SVM, and you should read the SVM chapter (chapter 9 of the first edition) if you are interested in such details. Here, we will simply give examples of SVM usage.

But First: What *is* a Support Vector, Anyway?

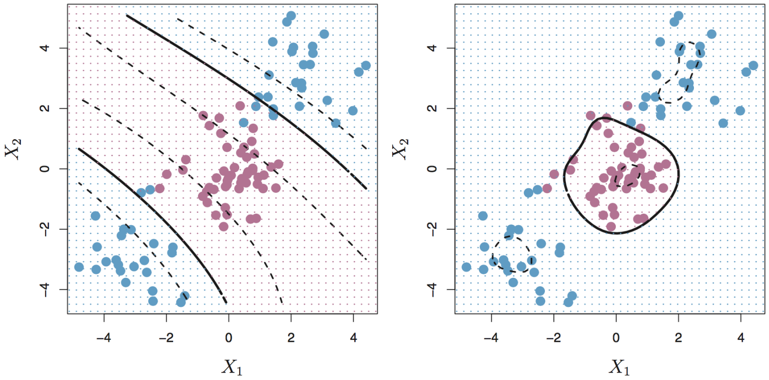


(Figure 9.3, *Introduction to Statistical Learning* by James et al.)

This figure shows an example of a *maximum margin classifier* that determines the linear boundary in the native space of the predictor data that has the largest *minimum* distance to a training datum. MMC requires a complete separation between instances of the two classes, so it is really never used in practice. However:

Look at the three short line segments that are perpendicular to the boundary. These are *support vectors*: they "hold up," or support, the linear boundary.

Support Vector Machine



(Figure 9.9, *Introduction to Statistical Learning* by James et al.)

SVM, as stated above, defines a linear boundary in a higher-dimensional space than the native space of the predictor variables.

Example: SVM with a polynomial kernel of degree 2 transforms a space where $p = 2$ to one where $p = 5$:

$$\{X_1, X_2\} \rightarrow \{X_1, X_2, X_1^2, X_2^2, X_1X_2\}\,.$$

(But it doesn't, really. It utilizes a bit of cleverness dubbed the "kernel trick" to keep the calculations in the native space of the data. However, despite this, you need to know that SVM is **really slow**. It is an order n-cubed algorithm, meaning that running SVM on a 10,000-row dataset can take up to 1000 times longer than running it on a 1000-row dataset. It is a nice algorithm that often yields good models, but really only useful for $n \sim 1000$...unless you are willing to wait.)

SVM: Example

We'll begin by importing data on 500 stars and 500 quasars.

```
##      col.ug            col.gr             col.ri
##  Min.   :-4.2274   Min.   :-2.98092   Min.   :-0.40610
##  1st Qu.: 0.6613   1st Qu.: 0.09591   1st Qu.: 0.02866
##  Median : 1.1102   Median : 0.26471   Median : 0.12162
##  Mean   : 1.3196   Mean   : 0.37682   Mean   : 0.21581
##  3rd Qu.: 1.7465   3rd Qu.: 0.51801   3rd Qu.: 0.25169
##  Max.   : 6.2807   Max.   : 2.68311   Max.   : 3.39274
##      col.iz             mag.r        class
##  Min.   :-3.69967   Min.   :14.43   QSO :500
##  1st Qu.: 0.02976   1st Qu.:17.95   STAR:500
##  Median : 0.14411   Median :18.75
##  Mean   : 0.18544   Mean   :18.66
##  3rd Qu.: 0.29248   3rd Qu.:19.47
##  Max.   : 4.04392   Max.   :24.82
```

We will use the functions of the `e1071` package below, after scaling the predictors and performing a 70-30 data split.

SVM: Example With Linear Kernel

```
library(e1071)

set.seed(202) # reproducible cross-validation
tune.out = tune(svm,class~.,data=df.train,kernel="linear",ranges=list(cost=10^seq(-2,2,by=0.2)))
cat("The estimated optimal value for C is ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal value for C is  1.584893
```

```
resp.pred = predict(tune.out$best.model,newdata=df.test)
mean(resp.pred!=df.test$class) ; table(resp.pred,df.test$class)
```

```
## [1] 0.1433333

##
## resp.pred QSO STAR
##      QSO  138   29
##      STAR  14  119
```

Compare with...

```
log.out = glm(class~.,data=df.train,family=binomial)
log.prob = predict(log.out,newdata=df.test,type="response")
log.pred = ifelse(log.prob>0.5,"STAR","QSO")
mean(log.pred!=df.test$class) ; table(log.pred,df.test$class)
```

```
## [1] 0.1766667

##
## log.pred QSO STAR
##     QSO  132   33
##     STAR  20  115
```

SVM: Example With Polynomial Kernel

```r
set.seed(202)
tune.out = tune(svm,class~.,data=df.train,kernel="polynomial",
                ranges=list(cost=10^seq(2,4,by=0.5),degree=2:4))
cat("The estimated optimal values for C and degree are ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal values for C and degree are  1000 3
```

```r
resp.pred = predict(tune.out$best.model,newdata=df.test)
(svm.poly.mcr = mean(resp.pred!=df.test$class))
```

```
## [1] 0.1033333
```

```r
table(resp.pred,df.test$class)
```

```
##
## resp.pred QSO STAR
##      QSO  130    9
##      STAR  22  139
```

Note: if the optimal value is a boundary value (for instance, the optimal value of cost is 1 when the minimum value considered is 1), change the `ranges` and rerun the code. Also, note the very sparse nature of the `ranges` grid: this is to ensure a relatively small computation time. You can iterate if you want to hone in more precisely on the tuning parameter optimal values.

SVM: Example With Radial Kernel

```
set.seed(202)
tune.out = tune(svm,class~.,data=df.train,kernel="radial",
                ranges=list(cost=10^seq(-1,1,by=0.5),gamma=10^seq(-1,1,by=0.4)))
cat("The estimated optimal values for C and gamma are ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal values for C and gamma are  1 0.6309573
```

```
resp.pred = predict(tune.out$best.model,newdata=df.test)
(svm.poly.mcr = mean(resp.pred!=df.test$class))
```

```
## [1] 0.05
```

```
table(resp.pred,df.test$class)
```

```
##
## resp.pred QSO STAR
##      QSO  142    5
##      STAR  10  143
```

If you go back a few slides, you will see that a radial-kernel SVM model does **much** better than a logistic regression model, in terms of misclassification rate.

Note that in our examples, we assumed that the class-separation threshold is 0.5 (which is OK because we had balanced classes). If you wish to create a ROC curve, you'd pass the argument `probability=TRUE` to *both* `tune()` and `predict()`; the output of the latter will be a matrix, and you'd extract the second column. You'd then pass that vector of numbers into `roc()`.