Cross Validation

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

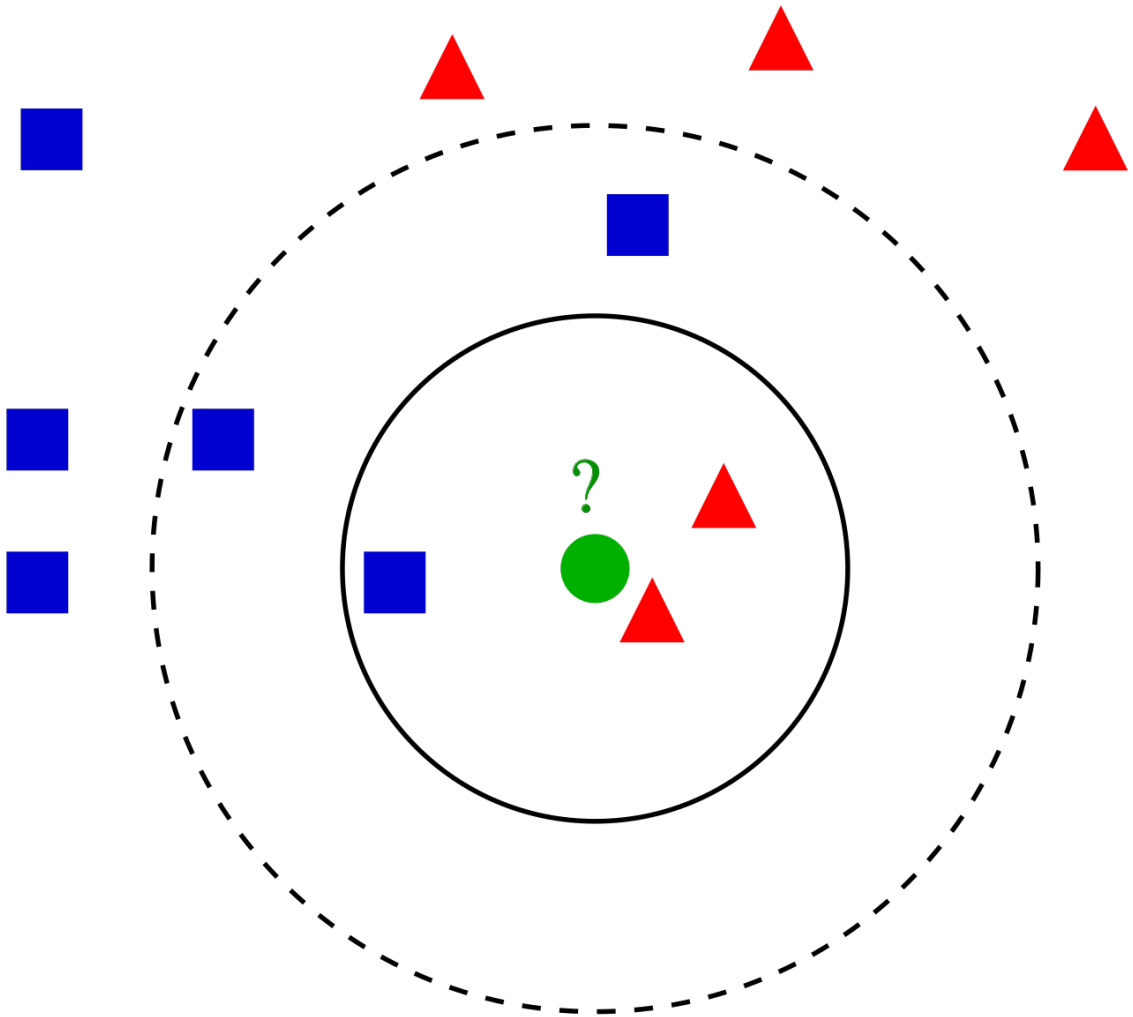
8/21/2023

Cross-validation

Cross-validation is one of the most important ideas in machine learning. Here we focus on the conceptual and mathematical aspects. We will describe how to implement cross validation in practice in later examples.

Cross-validation example

To motivate the concept, we will introduce an actual machine learning algorithm: **k-nearest neighbors (kNN)**.



Cross-validation example: k-nearest neighbors (kNN)

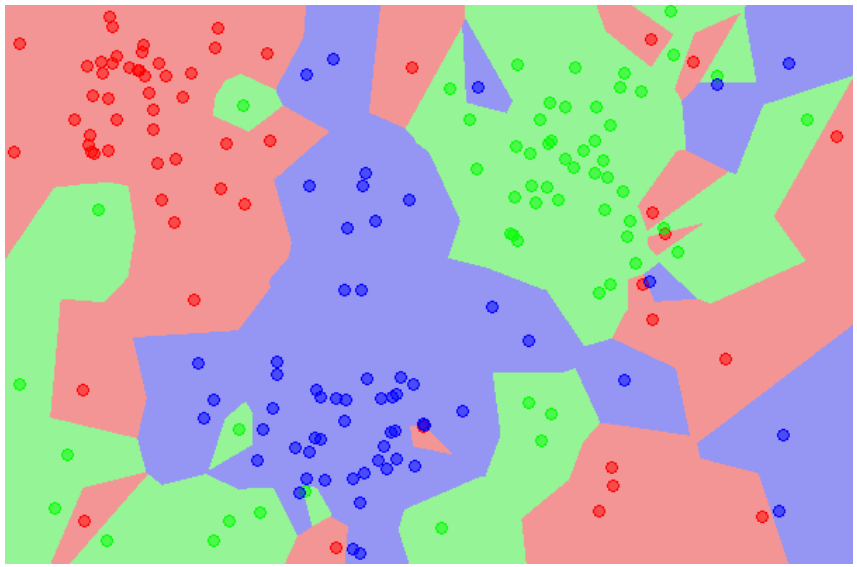
K-nearest neighbor algorithm pseudocode

Programming languages like Python and R are used to implement the KNN algorithm. The following is the pseudocode for KNN:

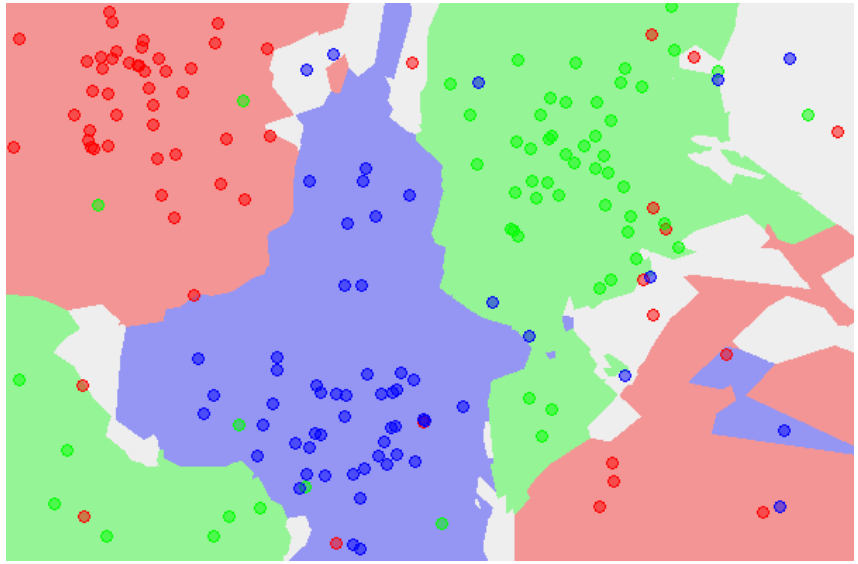
1. Load the data
2. Choose K value
3. For each data point in the data:
    - Find the Euclidean distance to all training data samples
    - Store the distances on an ordered list and sort it
    - Choose the top K entries from the sorted list
    - Label the test point based on the majority of classes present in the selected points
4. End

Cross-validation example: k-nearest neighbors (kNN)

Using k=*1*:



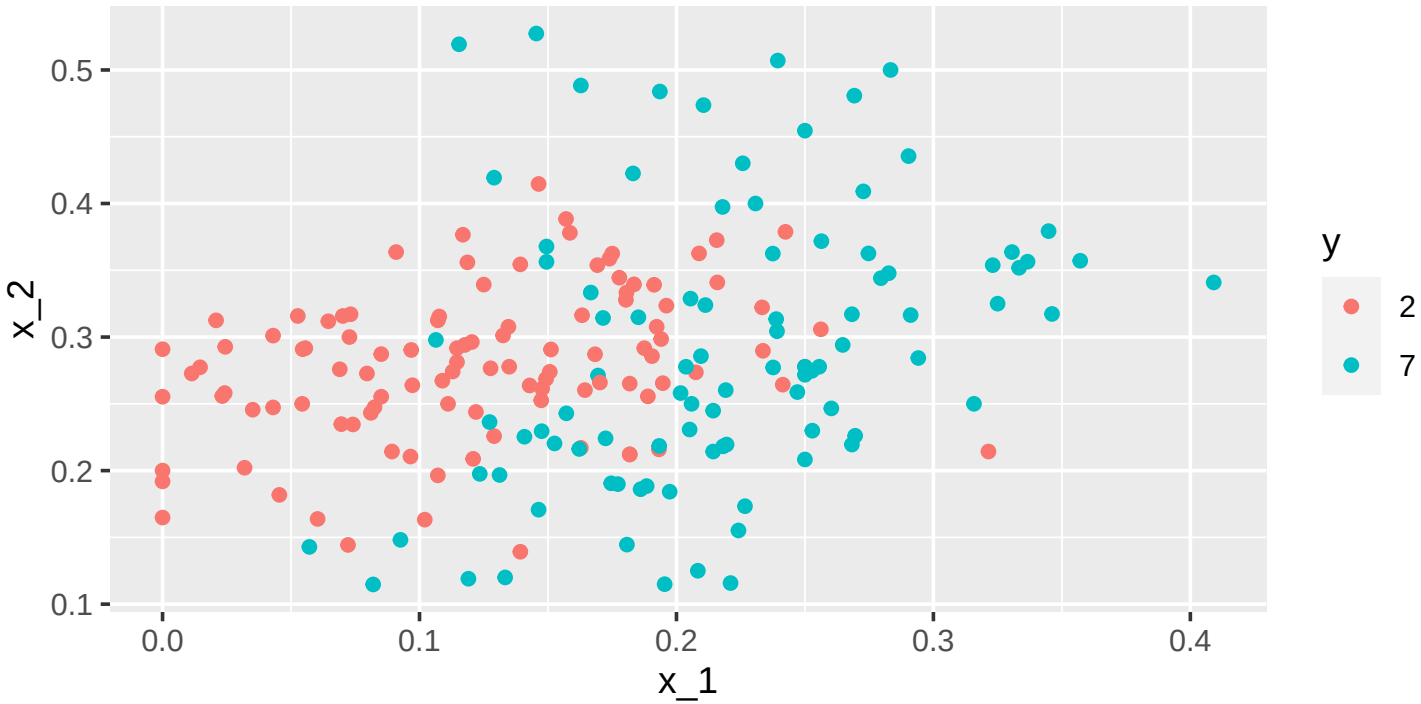
and k=5:



Cross-validation Dataset

Using the following dataset:

```
library(dslabs); data("mnist_27")
mnist_27$test %>% ggplot(aes(x_1, x_2, color = y)) + geom_point()
```



Cross-validation example: k-nearest neighbors (kNN)

We can use the `knn3` function from the `caret` package as follows, with the number of neighbors equal to $k = 5$.:

```
library(caret)
knn_fit <- knn3(y ~ ., data = mnist_27$train, k = 5)
```

Cross-validation example

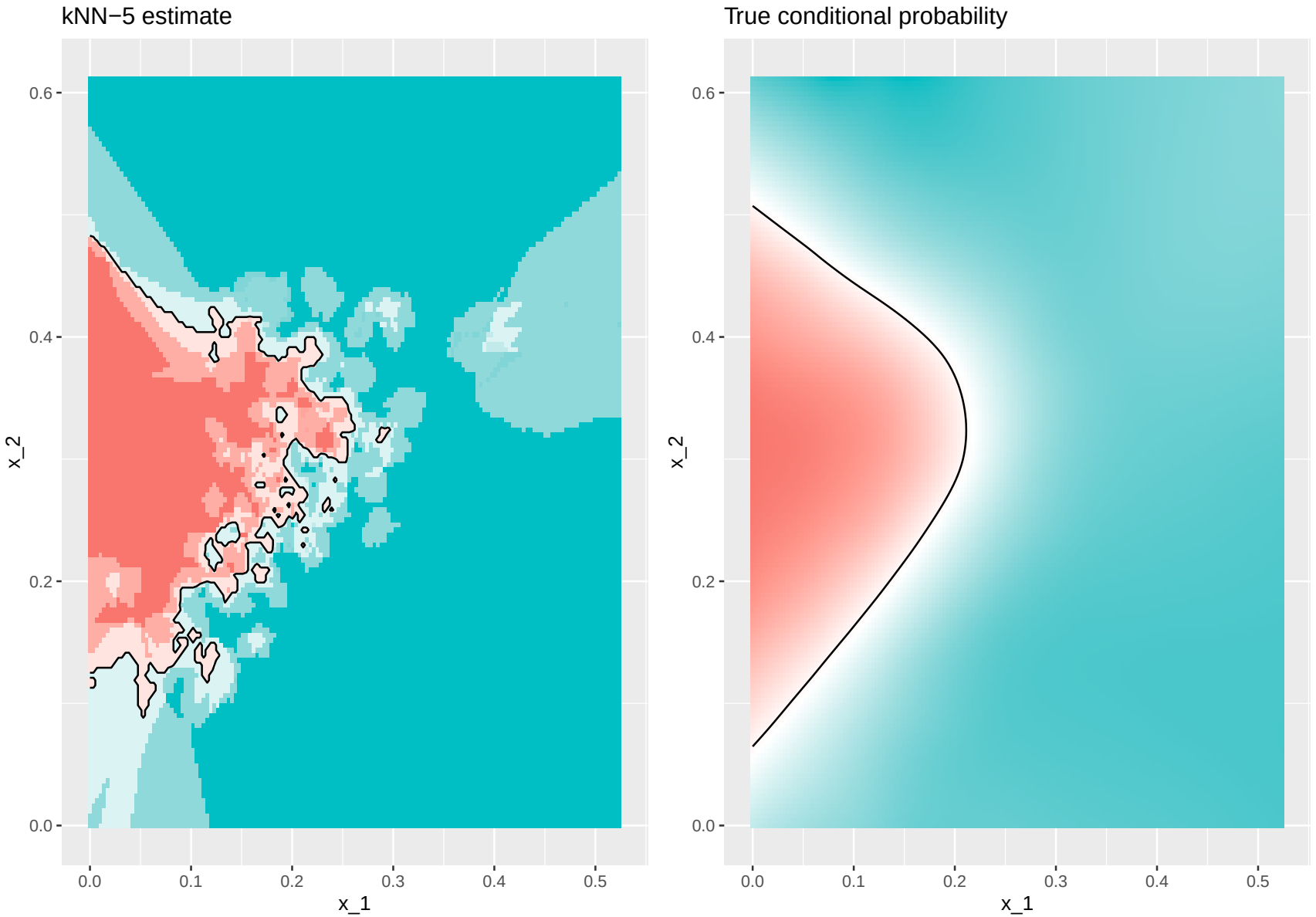
Since our dataset is balanced and we care just as much about sensitivity as we do about specificity, we will use accuracy to quantify performance.

The `predict` function for `knn` produces a probability for each class.

```
y_hat_knn <- predict(knn_fit, mnist_27$test, type = "class")
confusionMatrix(y_hat_knn, mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##    0.815
```

Cross-validation example



Over-training

Over-training or **over-fitting** results in having higher accuracy in the train set compared to the test set:

```
y_hat_knn <- predict(knn_fit, mnist_27$train, type = "class")
confusionMatrix(y_hat_knn, mnist_27$train$y)$overall["Accuracy"]
```

```
## Accuracy
##   0.8825
```

```
y_hat_knn <- predict(knn_fit, mnist_27$test, type = "class")
confusionMatrix(y_hat_knn, mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##    0.815
```

Over-training

Over-training is at its worst when we set $k = 1$:

```
knn_fit_1 <- knn3(y ~ ., data = mnist_27$train, k = 1)
y_hat_knn_1 <- predict(knn_fit_1, mnist_27$train, type = "class")
confusionMatrix(y_hat_knn_1, mnist_27$train$y)$overall[["Accuracy"]]
```

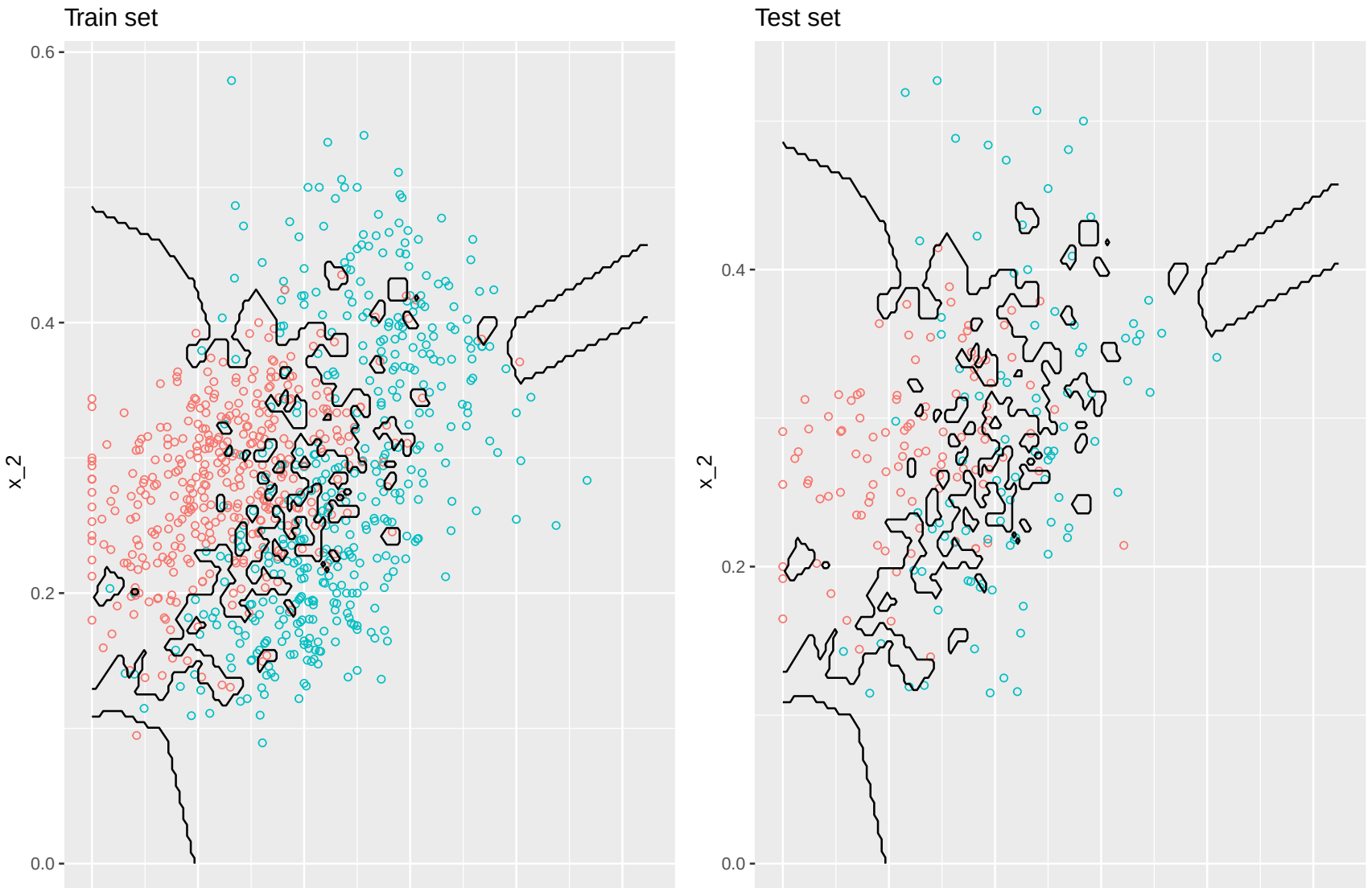
```
## [1] 0.99625
```

```
y_hat_knn_1 <- predict(knn_fit_1, mnist_27$test, type = "class")
confusionMatrix(y_hat_knn_1, mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.74
```

Over-training

We can see the over-fitting problem in this figure.



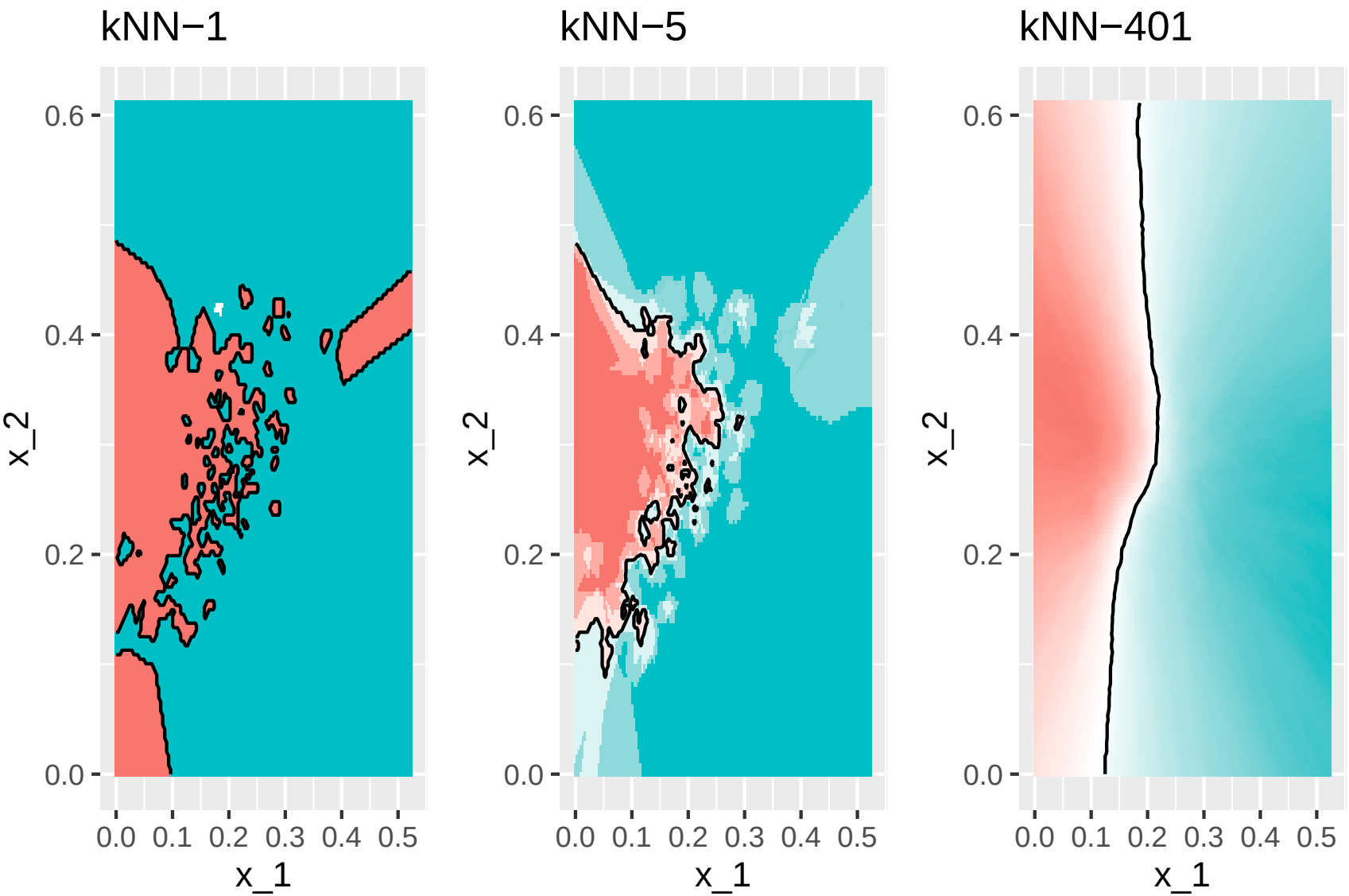
Over-smoothing

Although not as badly as with $k = 1$, we saw that $k = 5$ is also over-trained. Hence, we should consider a larger k. Let's try, as an example, a much larger number: $k = 401$.

```
knn_fit_401 <- knn3(y ~ ., data = mnist_27$train, k = 401)
y_hat_knn_401 <- predict(knn_fit_401, mnist_27$test,
                         type = "class")
confusionMatrix(y_hat_knn_401, mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.79
```

Over-smoothing



Picking the k in kNN

So how do we pick k? In principle, we want to pick the k that maximizes accuracy, or minimizes the expected MSE (defined later).

The goal of cross validation is to estimate these quantities for any given algorithm and set of tuning parameters such as k. To understand why we need a special method to do this let's repeat what we did above but for different values of k:

```
ks <- seq(3, 251, 2)
```

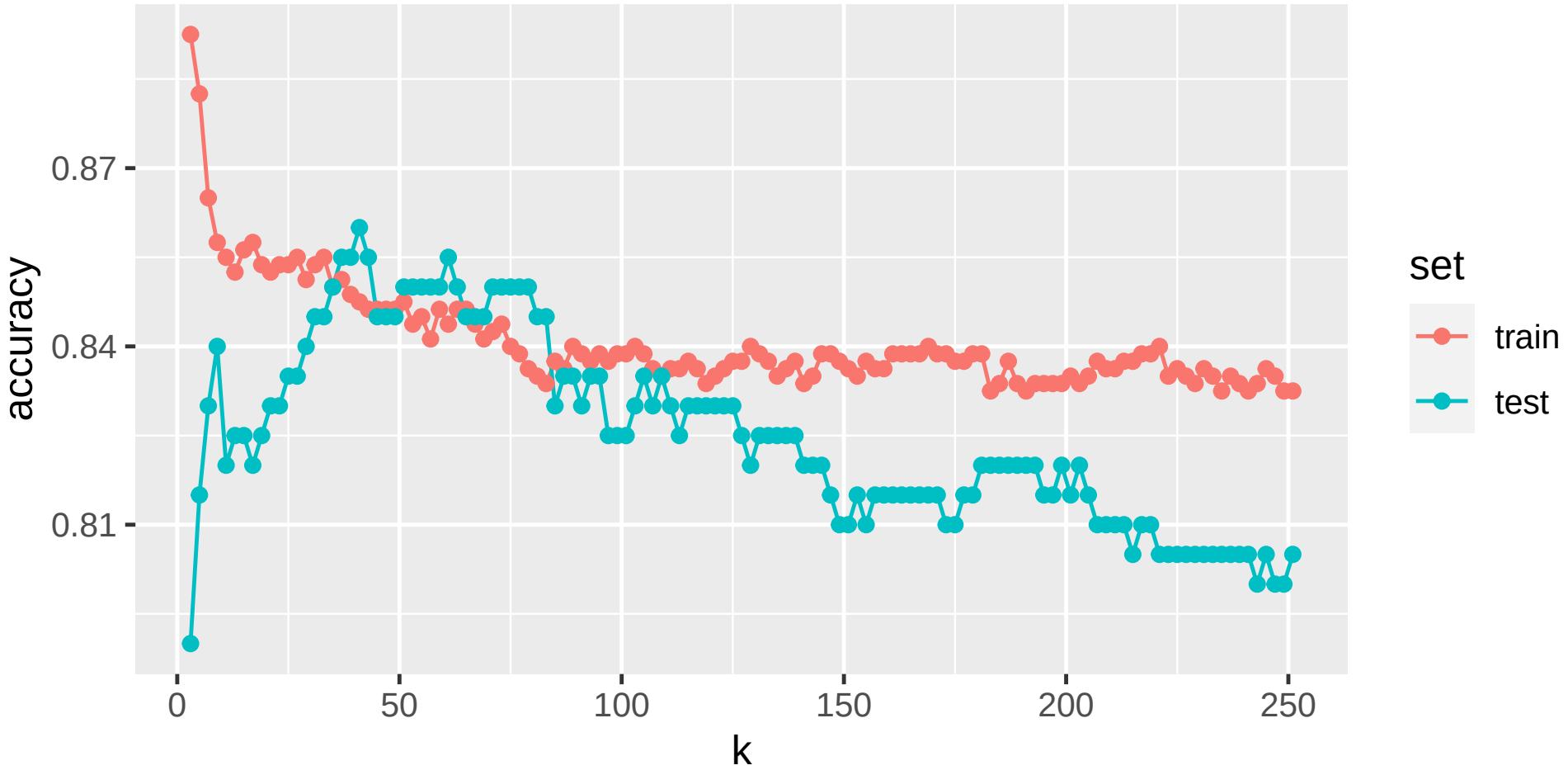
Picking the k in kNN

We do this using `map_df` function to repeat the above for each one.

```
library(purrr)
accuracy <- map_df(ks, function(k){
  fit <- knn3(y ~ ., data = mnist_27$train, k = k)

  y_hat <- predict(fit, mnist_27$train, type = "class")
  cm_train <- confusionMatrix(y_hat, mnist_27$train$y)
  train_error <- cm_train$overall["Accuracy"]

  y_hat <- predict(fit, mnist_27$test, type = "class")
  cm_test <- confusionMatrix(y_hat, mnist_27$test$y)
  test_error <- cm_test$overall["Accuracy"]

  tibble(train = train_error, test = test_error)
})
```

Picking the k in kNN

Note that we estimate accuracy by using both the training set and the test set. We can now plot the accuracy estimates for each value of k:



Picking the k in kNN

If we were to use these estimates to pick the k that maximizes accuracy, we would use the estimates built on the test data:

```
ks[which.max(accuracy$test)]
```

```
## [1] 41
```

```
max(accuracy$test)
```

```
## [1] 0.86
```

Picking the k in kNN

First, note that the estimate obtained on the training set is generally lower than the estimate obtained with the test set, with the difference larger for smaller values of k. This is due to over-training.

Also note that the accuracy versus k plot is quite jagged. We do not expect this because small changes in k should not affect the algorithm's performance too much. The jaggedness is explained by the fact that the accuracy is computed on a sample and therefore is a random variable. This demonstrates why we prefer to minimize the expected loss (defined later) rather than the loss we observe with one dataset.

Picking the k in kNN

Another reason we need a better estimate of accuracy is that if we use the test set to pick this k, we should not expect the accompanying accuracy estimate to extrapolate to the real world. This is because even here we broke a golden rule of machine learning: we selected the k using the test set. Cross validation also provides an estimate that takes this into account.

Mathematical description of cross validation

A common goal of machine learning is to find an algorithm that produces predictors $\hat{Y}$ for an outcome Y that minimizes the MSE:

$$\text{MSE} = \text{E}\left\{\frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2\right\}$$

When all we have at our disposal is one dataset, we can estimate the MSE with the observed MSE like this:

$$\hat{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

These two are often referred to as the *true error* and *apparent error*, respectively.

Mathematical description of cross validation

There are two important characteristics of the apparent error we should always keep in mind:

1. Because our data is random, the apparent error is a random variable. For example, the dataset we have may be a random sample from a larger population. An algorithm may have a lower apparent error than another algorithm due to luck.
2. If we train an algorithm on the same dataset that we use to compute the apparent error, we might be overtraining. In general, when we do this, the apparent error will be an underestimate of the true error. We will see an extreme example of this with k-nearest neighbors.

Mathematical description of cross validation

Cross validation is a technique that permits us to alleviate both these problems. To understand cross validation, it helps to think of the true error, a theoretical quantity, as the average of many apparent errors obtained by applying the algorithm to B new random samples of the data, none of them used to train the algorithm:

$$\frac{1}{B}\sum_{b=1}^{B}\frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}_i^b - y_i^b\right)^2$$

with B a large number that can be thought of as practically infinite.

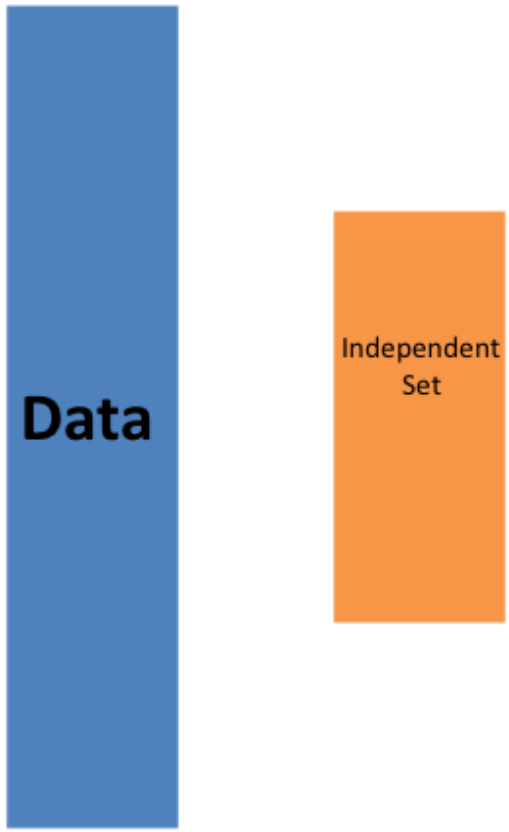
Mathematical description of cross validation

As already mentioned, this is a theoretical quantity because we only have available one set of outcomes: $y_1, \ldots, y_n$. Cross validation is based on the idea of imitating the theoretical setup above as best we can with the data we have.

To do this, we have to generate a series of different random samples. There are several approaches we can use, but the general idea for all of them is to randomly generate smaller datasets that are not used for training, and instead used to estimate the true error.

K-fold cross validation

The first one we describe is **K-fold cross validation**. A machine learning challenge starts with a dataset (blue). We need to use this to build an algorithm that will be used in an independent validation dataset (yellow).



K-fold cross validation

But we don't get to see these independent datasets.

K-fold cross validation

So to imitate this situation, we carve out a piece of our dataset and pretend it is an independent dataset: we divide the dataset into a **training set** (blue) and a **test set** (red). We will train our algorithm exclusively on the training set and use the test set only for evaluation purposes.

K-fold cross validation

Now this presents a new problem because for most machine learning algorithms we need to select parameters, for example the number of neighbors k in k-nearest neighbors. Here, we will refer to the set of parameters as λ.

We need to optimize algorithm parameters without using our test set and we know that if we optimize and evaluate on the same dataset, we will overtrain.

For each set of algorithm parameters being considered, we want an estimate of the MSE and then we will choose the parameters with the smallest MSE. Cross validation provides this estimate.

K-fold cross validation

First, before we start the cross validation procedure, it is important to fix all the algorithm parameters. We will use $\hat{y}_i(\lambda)$ to denote the predictors obtained when we use parameters λ.

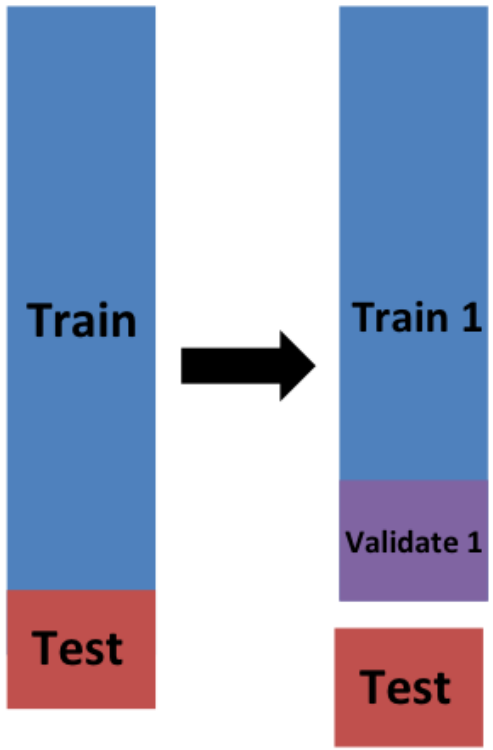
So, if we are going to imitate this definition:

$$\text{MSE}(\lambda) = \frac{1}{B}\sum_{b=1}^{B}\frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}_i^b(\lambda) - y_i^b\right)^2$$

We want to consider datasets that can be thought of as an independent random sample and we want to do this several times. With K-fold cross validation, we do it K times. We are showing an example that uses $K = 5$.

K-fold cross validation

We will eventually end up with K samples, but let's start by describing how to construct the first: we simply pick $M = N/K$ observations at random (we round if M is not a round number) and think of these as a random sample $y_1^b, \ldots, y_M^b$, with $b = 1$. We call this the validation set:



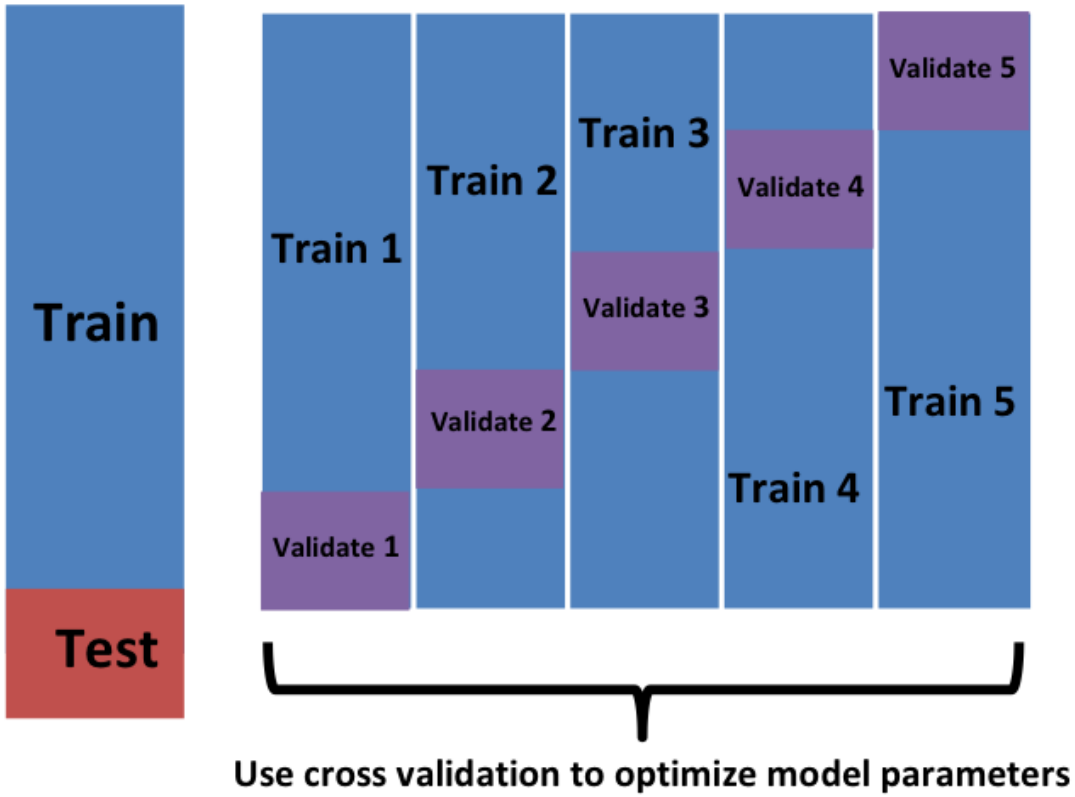
K-fold cross validation

Now we can fit the model in the training set, then compute the apparent error on the independent set:

$$\hat{\text{MSE}}_b(\lambda) = \frac{1}{M}\sum_{i=1}^{M}\left(\hat{y}_i^b(\lambda) - y_i^b\right)^2$$

K-fold cross validation

Note that this is just one sample and will therefore return a noisy estimate of the true error. This is why we take K samples, not just one. In K-cross validation, we randomly split the observations into K non-overlapping sets:



K-fold cross validation

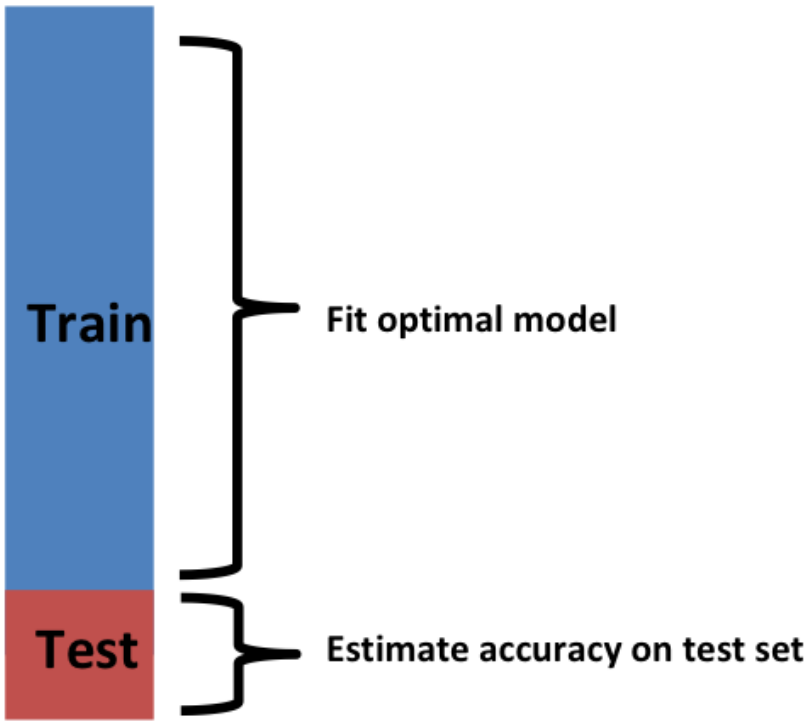
Now we repeat the calculation above for each of these sets $b = 1, \ldots, K$ and obtain $\hat{\text{MSE}}_1(\lambda), \ldots, \hat{\text{MSE}}_K(\lambda)$. Then, for our final estimate, we compute the average:

$$\hat{\text{MSE}}(\lambda) = \frac{1}{B} \sum_{b=1}^{K} \hat{\text{MSE}}_b(\lambda)$$

and obtain an estimate of our loss. A final step would be to select the λ that minimizes the MSE.

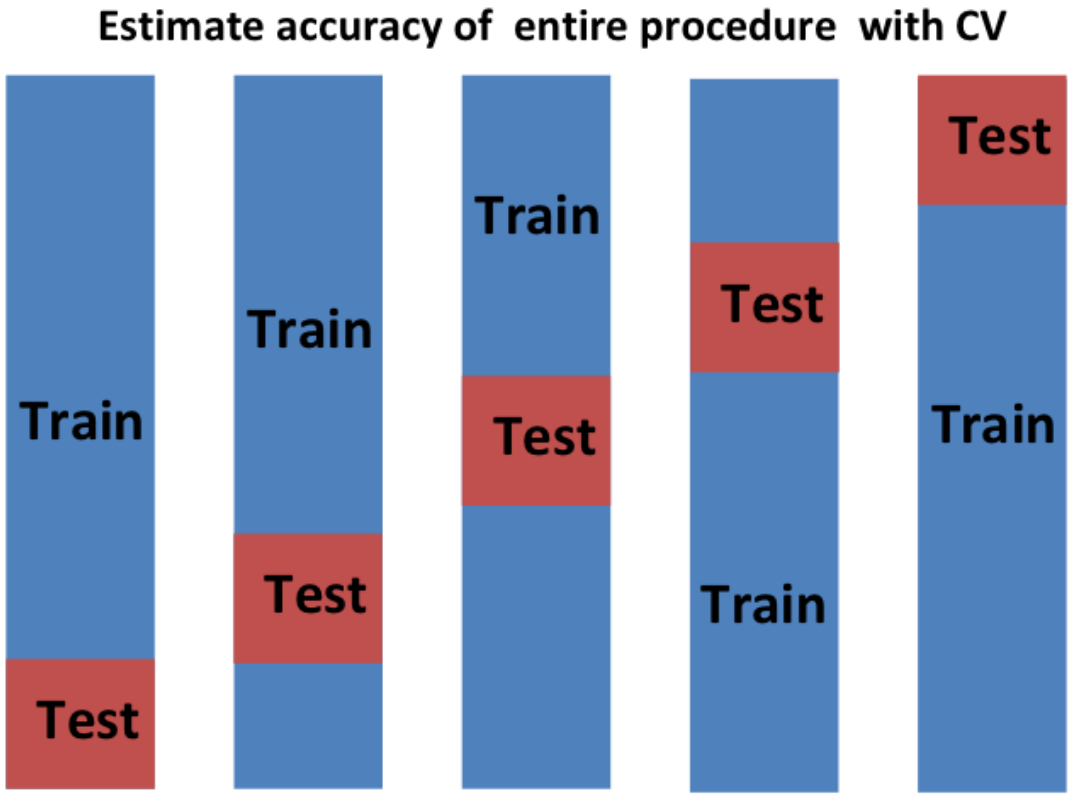
K-fold cross validation

We have described how to use cross validation to optimize parameters. However, we now have to take into account the fact that the optimization occurred on the training data. Here is where we use the test set we separated early on:



K-fold cross validation

We can do cross validation again:

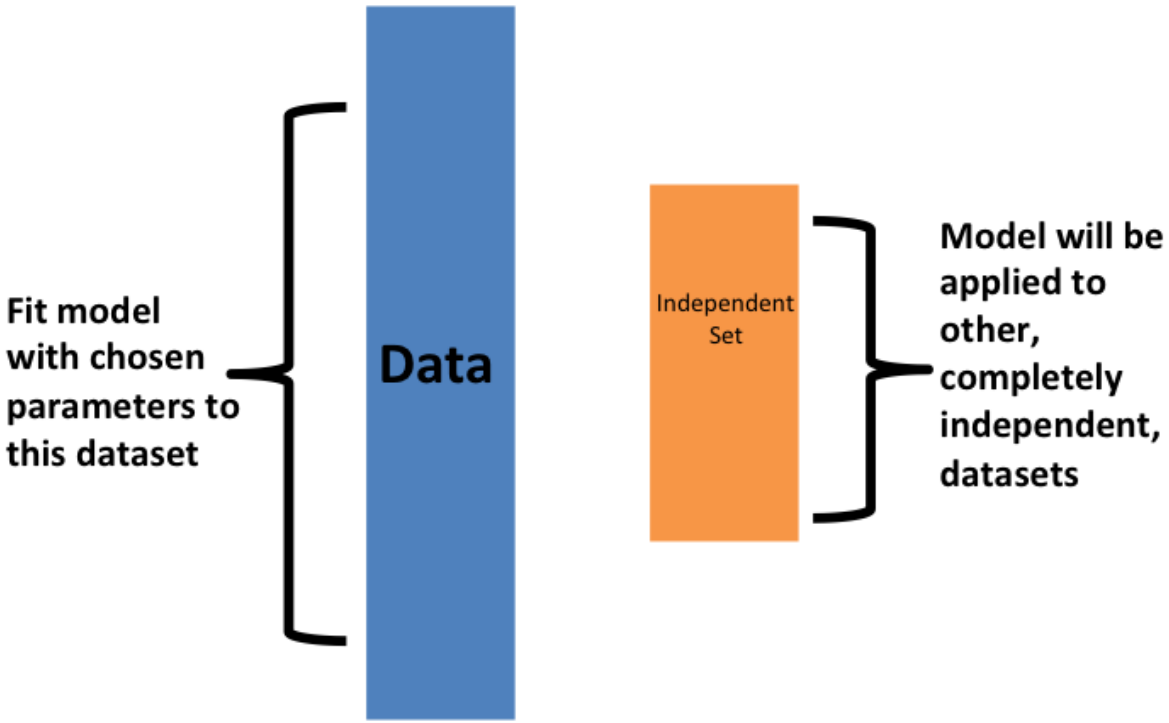


K-fold cross validation

and obtain a final estimate of our expected loss. However, note that this means that our entire compute time gets multiplied by K. You will soon learn that performing this task takes time because we are performing many complex computations. As a result, we are always looking for ways to reduce this time. For the final evaluation, we often just use the one test set.

Once we are satisfied with this model and want to make it available to others, we could refit the model on the entire dataset, without changing the optimized parameters.

K-fold cross validation



K-fold cross validation

Now how do we pick the cross validation K? Large values of K are preferable because the training data better imitates the original dataset. However, larger values of K will have much slower computation time: for example, 100-fold cross validation will be 10 times slower than 10-fold cross validation. For this reason, the choices of $K = 5$ and $K = 10$ are popular.

K-fold cross validation

One way we can improve the variance of our final estimate is to take more samples. To do this, we would no longer require the training set to be partitioned into non-overlapping sets. Instead, we would just pick K sets of some size at random.

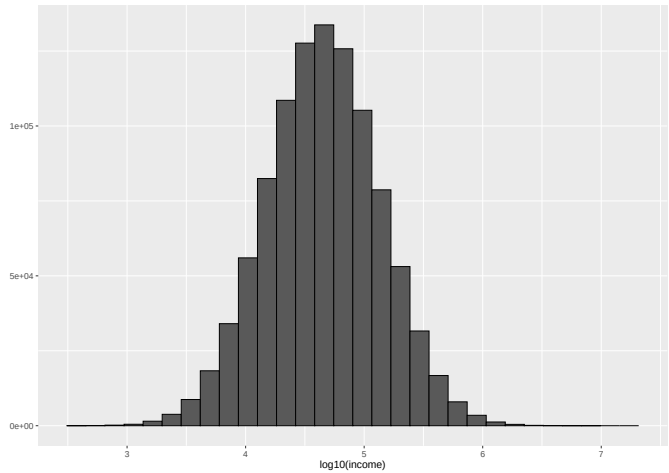
K-fold cross validation

One popular version of this technique, at each fold, picks observations at random with replacement (which means the same observation can appear twice). This approach has some advantages (not discussed here) and is generally referred to as the *bootstrap*. In fact, this is the default approach in the **caret** package. We describe how to implement cross validation with the **caret** package in the next chapter. In the next section, we include an explanation of how the bootstrap works in general.

Bootstrap

Suppose the income distribution of your population is as follows:

```
set.seed(1995)
n <- 10^6
income <- 10^(rnorm(n, log10(45000), log10(3)))
suppressWarnings(qplot(log10(income), bins = 30, color = I("bl
```



Bootstrap

The population median is:

```
m <- median(income)
m
```

```
## [1] 44938.54
```

Bootstrap

Suppose we don't have access to the entire population, but want to estimate the median m. We take a sample of 100 and estimate the population median m with the sample median M:

```
N <- 100
X <- sample(income, N)
median(X)
```

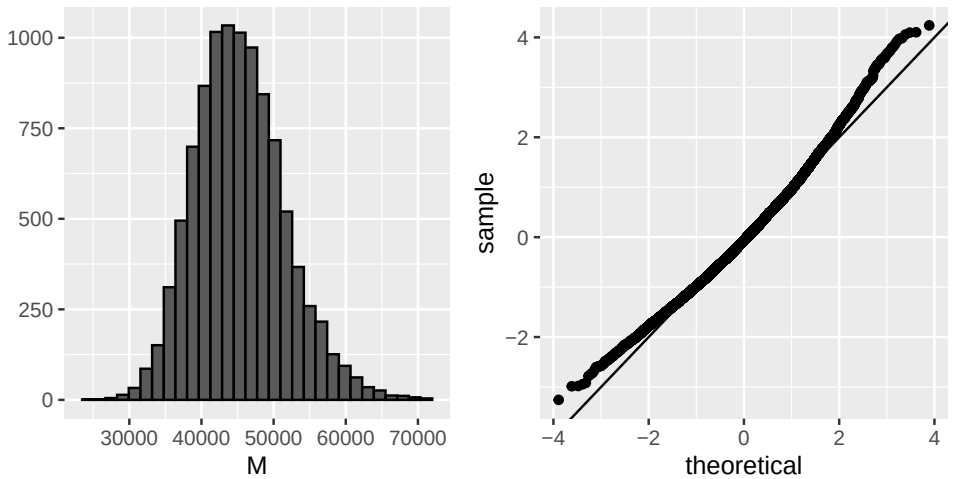
```
## [1] 38461.33
```

Can we construct a confidence interval? What is the distribution of M?

Bootstrap

We can use a Monte Carlo simulation to learn the distribution of M.

```
library(gridExtra)
B <- 10^4
M <- replicate(B, {X <- sample(income, N); median(X)})
p1 <- qplot(M, bins = 30, color = I("black"))
p2 <- qplot(sample = scale(M), xlab = "theoretical",
            ylab = "sample") + geom_abline()
grid.arrange(p1, p2, ncol = 2)
```



Bootstrap

If we know this distribution, we can construct a confidence interval. The problem here is that, as we have already described, in practice we do not have access to the distribution. We can see that the 95% confidence interval based on CLT

```
median(X) + 1.96 * sd(X) / sqrt(N) * c(-1, 1)
```

```
## [1] 21017.93 55904.72
```

is quite different from the confidence interval we would generate if we know the actual distribution of M:

```
quantile(M, c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 34437.72 59049.59
```

Bootstrap

The bootstrap permits us to approximate a Monte Carlo simulation without access to the entire distribution. The general idea is relatively simple. We act as if the observed sample is the population. We then sample (with replacement) datasets, of the same sample size as the original dataset. Then we compute the summary statistic, in this case the median, on these *bootstrap samples*.

Theory tells us that, in many situations, the distribution of the statistics obtained with bootstrap samples approximate the distribution of our actual statistic.

Bootstrap

This is how we construct bootstrap samples and an approximate distribution:

```
B <- 10^4
M_star <- replicate(B, {
  X_star <- sample(X, N, replace = TRUE)
  median(X_star)
})
```

Note a confidence interval constructed with the bootstrap is much closer to one constructed with the theoretical distribution:

```
quantile(M_star, c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 30252.82 56908.62
```

Bootstrap

For more on the Bootstrap, including corrections one can apply to improve these confidence intervals, please consult the book *An introduction to the bootstrap* by Efron, B., & Tibshirani, R. J.

Note that we can use ideas similar to those used in the bootstrap in cross validation: instead of dividing the data into equal partitions, we simply bootstrap many times.

The caret package

We have already learned about the kNN machine learning algorithms. This is just one of many algorithms out there. Many of these algorithms are implemented in R. However, they are distributed via different packages, developed by different authors, and often use different syntax.

The caret package

The **caret** package tries to consolidate these differences and provide consistency. It currently includes 237 different methods which are summarized in the **caret** package manual[1].

Keep in mind that **caret** does not include the needed packages and, to implement a package through **caret**, you still need to install the library. The required packages for each method are described in the package manual.

[1] https://topepo.github.io/caret/available-models.html

The caret `train` functon

The **caret** package also provides a function that performs cross validation for us. Here we provide some examples showing how we use this incredibly helpful package. We will use the 2 or 7 example to illustrate:

```
library(tidyverse)
library(dslabs)
data("mnist_27")
```

The **caret** `train` function lets us train different algorithms using similar syntax. So, for example, we can type:

```
library(caret)
train_glm <- train(y ~ ., method = "glm", data = mnist_27$train)
train_knn <- train(y ~ ., method = "knn", data = mnist_27$train)
```

The caret `train` functon

To make predictions, we can use the output of this function directly without needing to look at the specifics of `predict.glm` and `predict.knn`. Instead, we can learn how to obtain predictions from `predict.train`.

The code looks the same for both methods:

```
y_hat_glm <- predict(train_glm, mnist_27$test, type = "raw")
y_hat_knn <- predict(train_knn, mnist_27$test, type = "raw")
```

The caret `train` functon

This permits us to quickly compare the algorithms. For example, we can compare the accuracy like this:

```
confusionMatrix(y_hat_glm, mnist_27$test$y)$overall[["Accuracy"]]
```

```
## [1] 0.75
```

```
confusionMatrix(y_hat_knn, mnist_27$test$y)$overall[["Accuracy"]]
```

```
## [1] 0.84
```

Cross validation

When an algorithm includes a tuning parameter, `train` automatically uses cross validation to decide among a few default values. To find out what parameter or parameters are optimized, you can read the manual [2] or study the output of:

```
getModelInfo("knn")
```

We can also use a quick lookup like this:

```
modelLookup("knn")
```

If we run it with default values:

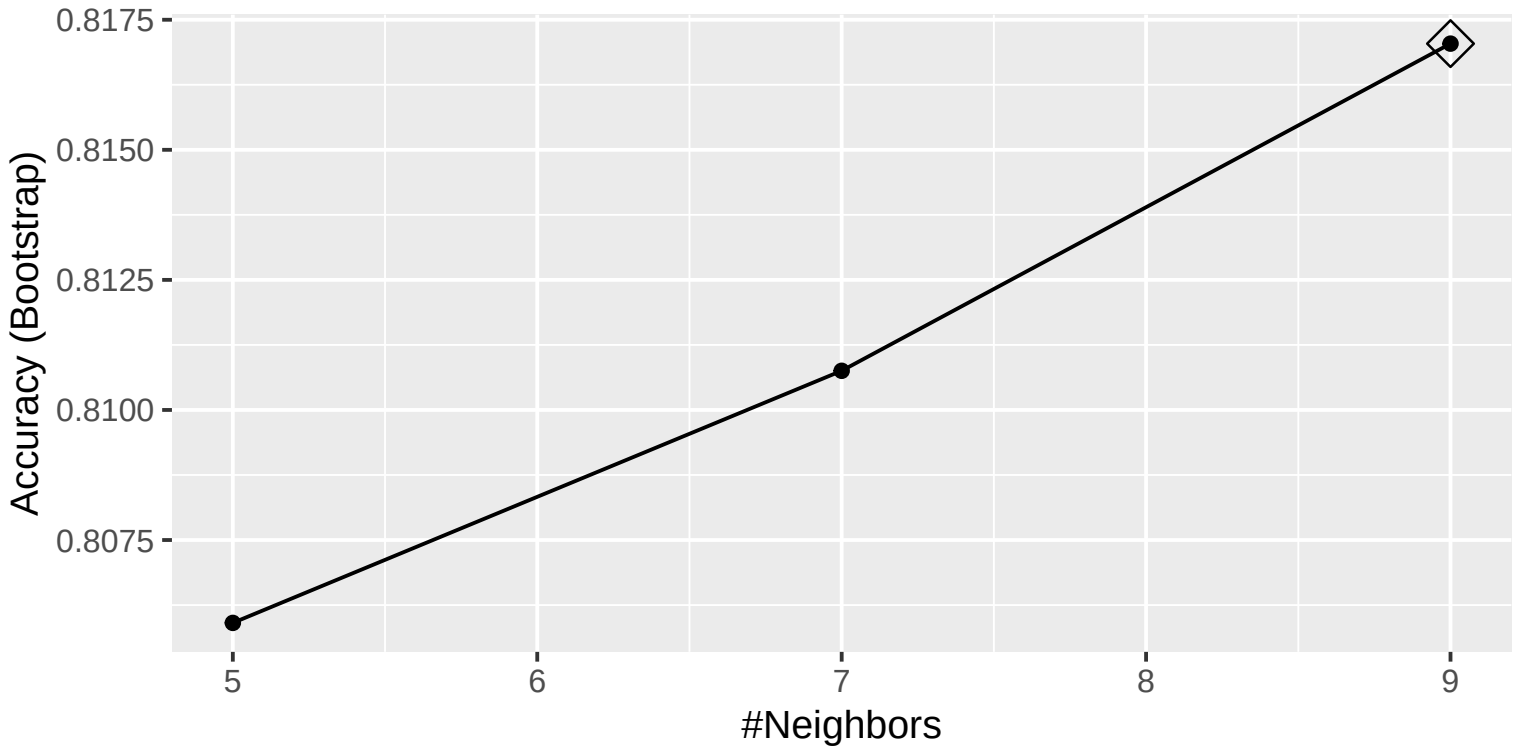
```
train_knn <- train(y ~ ., method = "knn", data = mnist_27$train)
```

[2] http://topepo.github.io/caret/available-models.html

Cross validation

You can quickly see the results of the cross validation using the `ggplot` function. The argument `highlight` highlights the max:

```
ggplot(train_knn, highlight = TRUE)
```



Cross validation

By default, the cross validation is performed by taking 25 bootstrap samples comprised of 25% of the observations. For the `kNN` method, the default is to try $k = 5, 7, 9$. We change this using the `tuneGrid` parameter. The grid of values must be supplied by a data frame with the parameter names as specified in the `modelLookup` output.

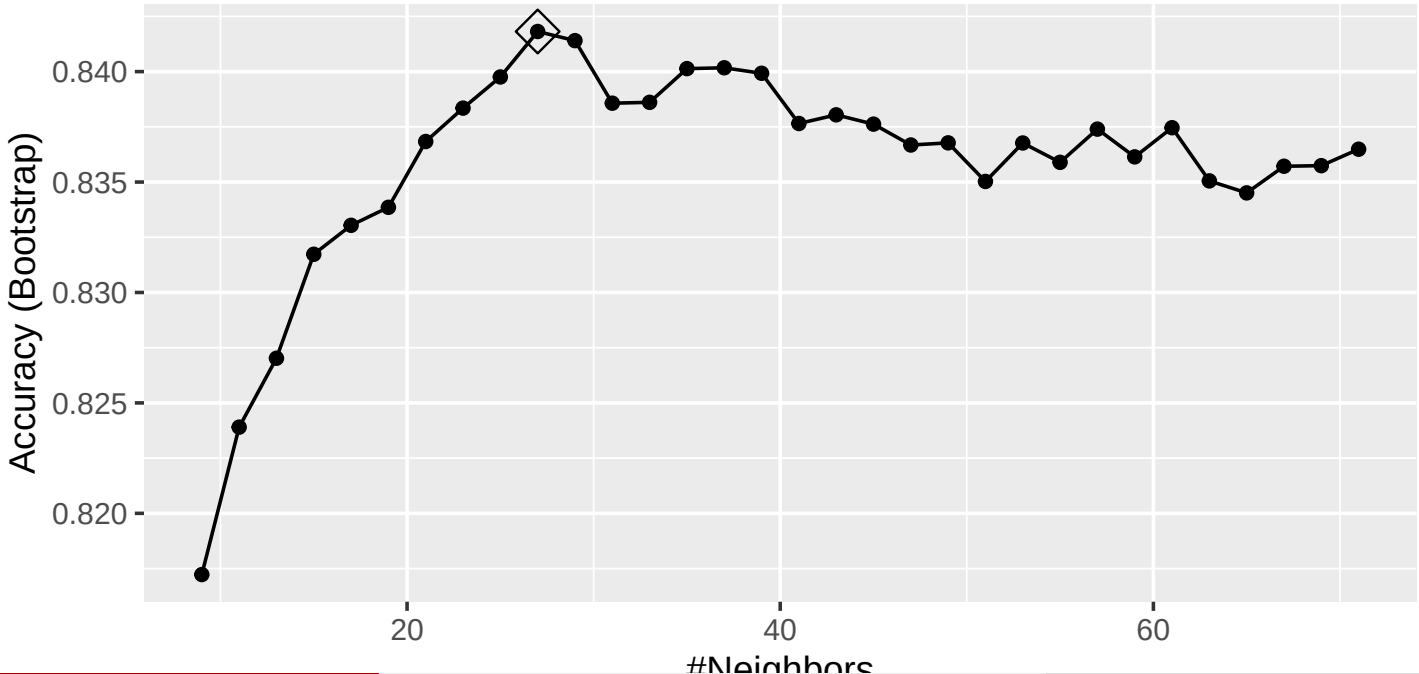
Here, we present an example where we try out 30 values between 9 and 67. To do this with **caret**, we need to define a column named `k`, so we use this: `data.frame(k = seq(9, 67, 2))`.

Note that when running this code, we are fitting 30 versions of kNN to 25 bootstrapped samples. Since we are fitting $30 \times 25 = 750$ kNN models.

Cross validation

We will set the seed because cross validation is a random procedure and we want to make sure the result here is reproducible.

```
set.seed(2008)
train_knn <- train(y ~ ., method = "knn",
                   data = mnist_27$train,
                   tuneGrid = data.frame(k = seq(9, 71, 2)))
ggplot(train_knn, highlight = TRUE)
```



Cross validation

To access the parameter that maximized the accuracy, you can use this:

```
train_knn$bestTune
```

```
##     k
## 10 27
```

and the best performing model like this:

```
train_knn$finalModel
```

```
## 27-nearest neighbor model
## Training set outcome distribution:
##
##   2   7
## 379 421
```

Cross validation

The function `predict` will use this best performing model. Here is the accuracy of the best model when applied to the test set, which we have not used at all yet because the cross validation was done on the training set:

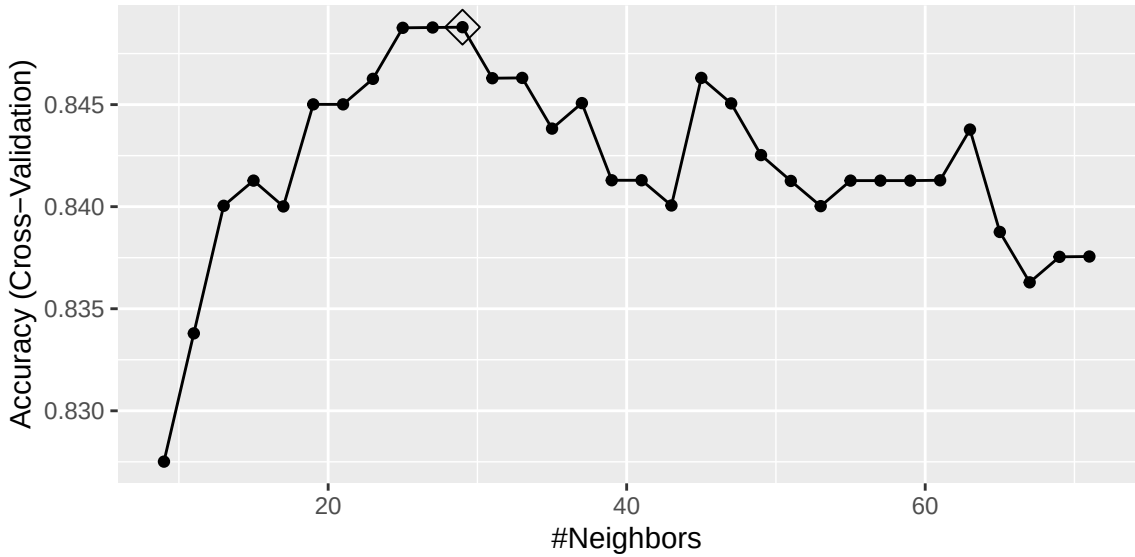
```
confusionMatrix(predict(train_knn, mnist_27$test, type = "raw"),
                mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##    0.835
```

Cross validation

If we want to change how we perform cross validation, we can use the `trainControl` function. We can make the code above go a bit faster by using, for example, 10-fold cross validation:

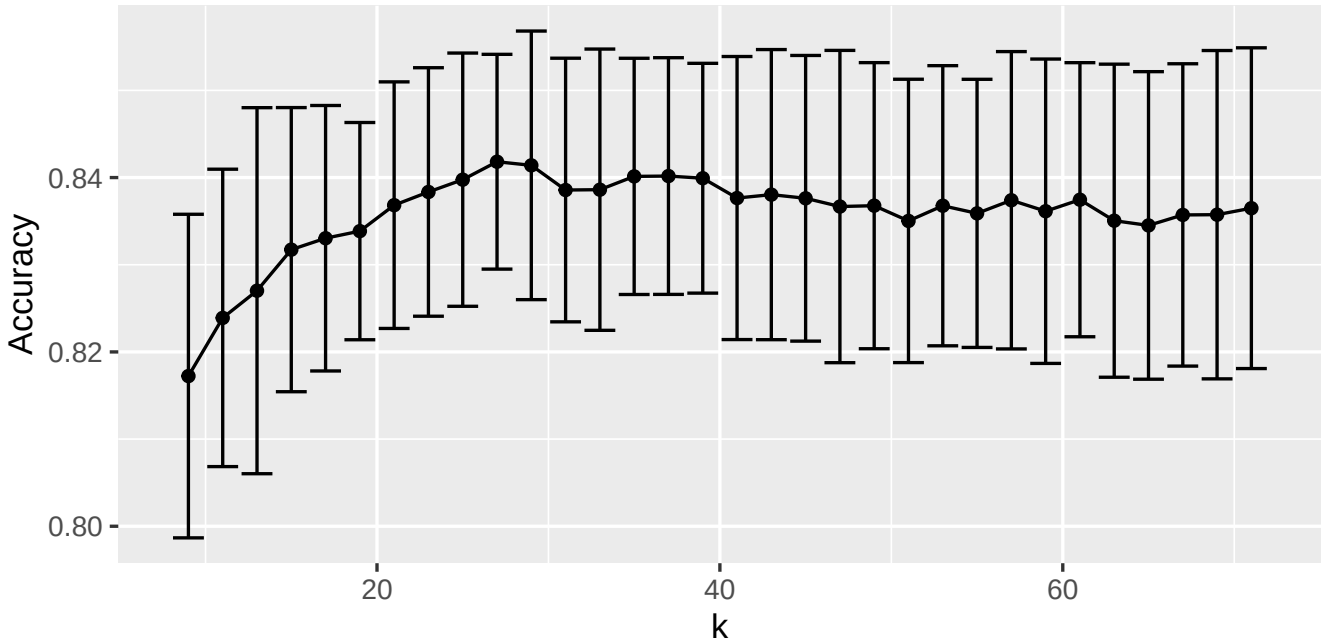
```
control <- trainControl(method = "cv", number = 10, p = .9)
train_knn_cv <- train(y ~ ., method = "knn",
                      data = mnist_27$train,
                      tuneGrid = data.frame(k = seq(9, 71, 2)),
                      trControl = control)
ggplot(train_knn_cv, highlight = TRUE)
```



Cross validation

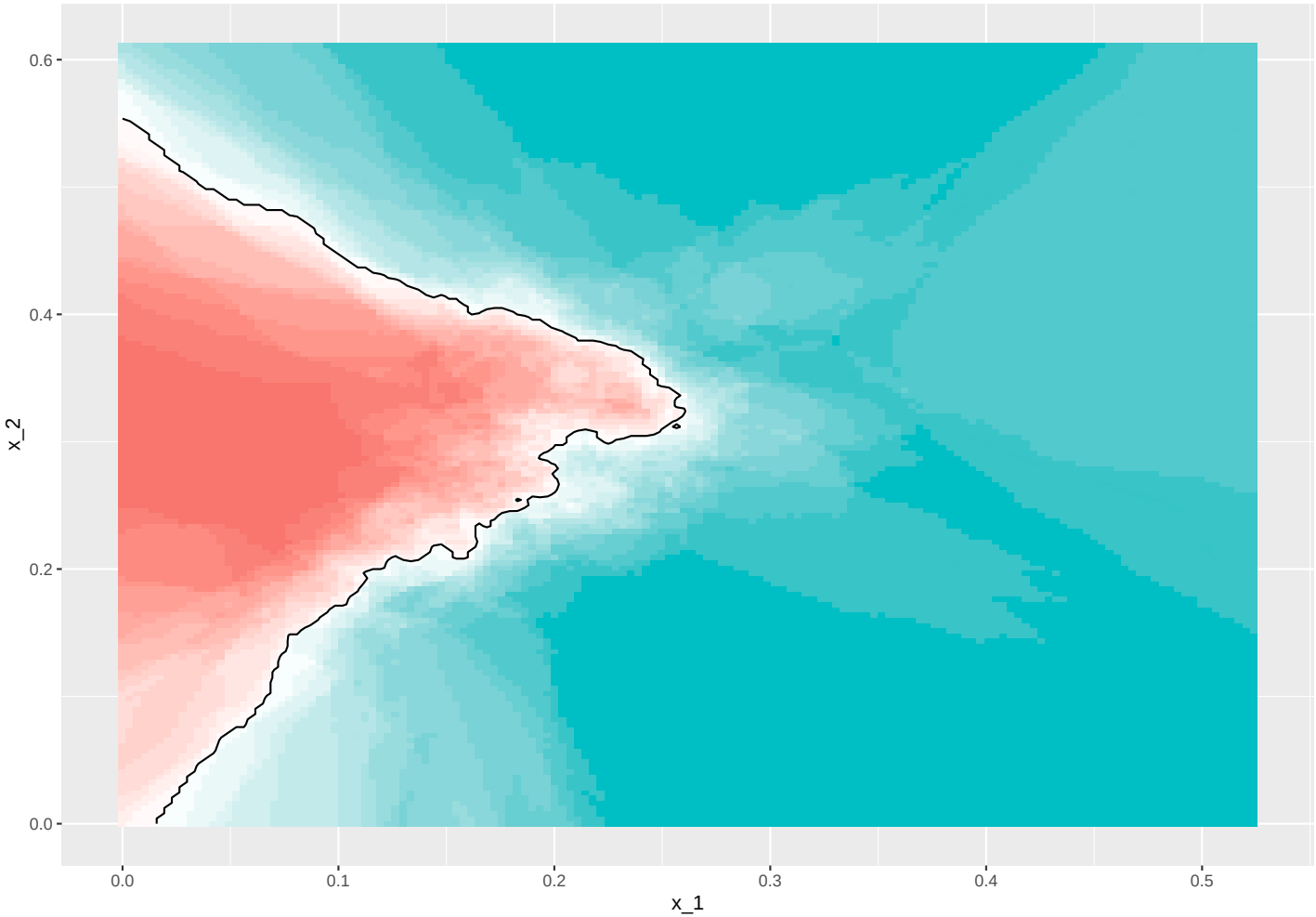
We can also see the standard deviation bars obtained from the cross validation samples:

```
train_knn$results %>%
  ggplot(aes(x = k, y = Accuracy)) +
  geom_line() + geom_point() +
  geom_errorbar(aes(x = k,
                    ymin = Accuracy - AccuracySD,
                    ymax = Accuracy + AccuracySD))
```



Example: fitting with loess

The best fitting kNN model approximates the true conditional probability:



Example: fitting with loess

However, we do see that the boundary is somewhat wiggly. This is because kNN, like the basic bin smoother, does not use a kernel. To improve this we could try loess. By reading through the available models part of the manual[3] we see that we can use the `gamLoess` method. In the manual[4] we also see that we need to install the **gam** package if we have not done so already:

```
install.packages("gam")
```

[3] https://topepo.github.io/caret/available-models.html
[4] https://topepo.github.io/caret/train-models-by-tag.html

Example: fitting with loess

Then we see that we have two parameters to optimize:

```
modelLookup("gamLoess")
```

```
##       model parameter  label forReg forClass probModel
## 1 gamLoess      span   Span   TRUE     TRUE      TRUE
## 2 gamLoess    degree Degree   TRUE     TRUE      TRUE
```

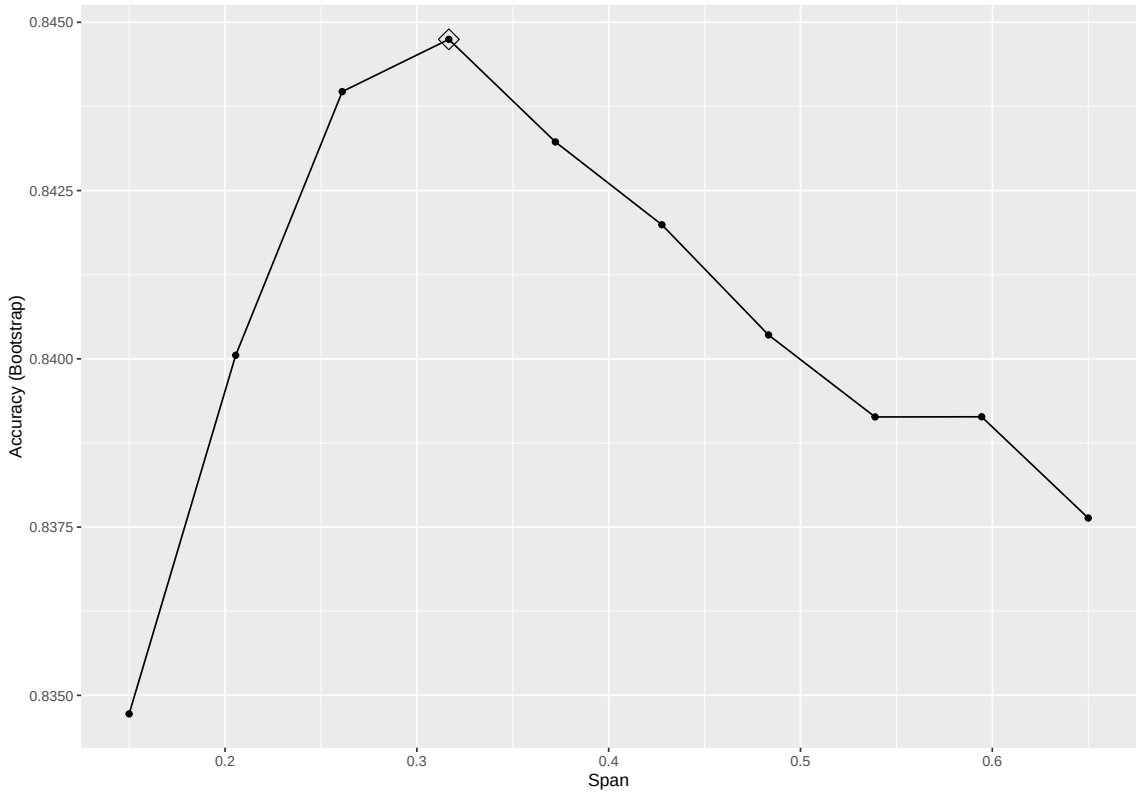
We will stick to a degree of 1. But to try out different values for the span, we still have to include a column in the table with the name degree so we can do this:

```
grid <- expand.grid(span = seq(0.15, 0.65, len = 10)
                    , degree = 1)
```

Example: fitting with loess

We will use the default cross validation control parameters.

```
train_loess <- train(y ~ ., method = "gamLoess",
                     tuneGrid=grid,
                     data = mnist_27$train)
ggplot(train_loess, highlight = TRUE)
```



Example: fitting with loess

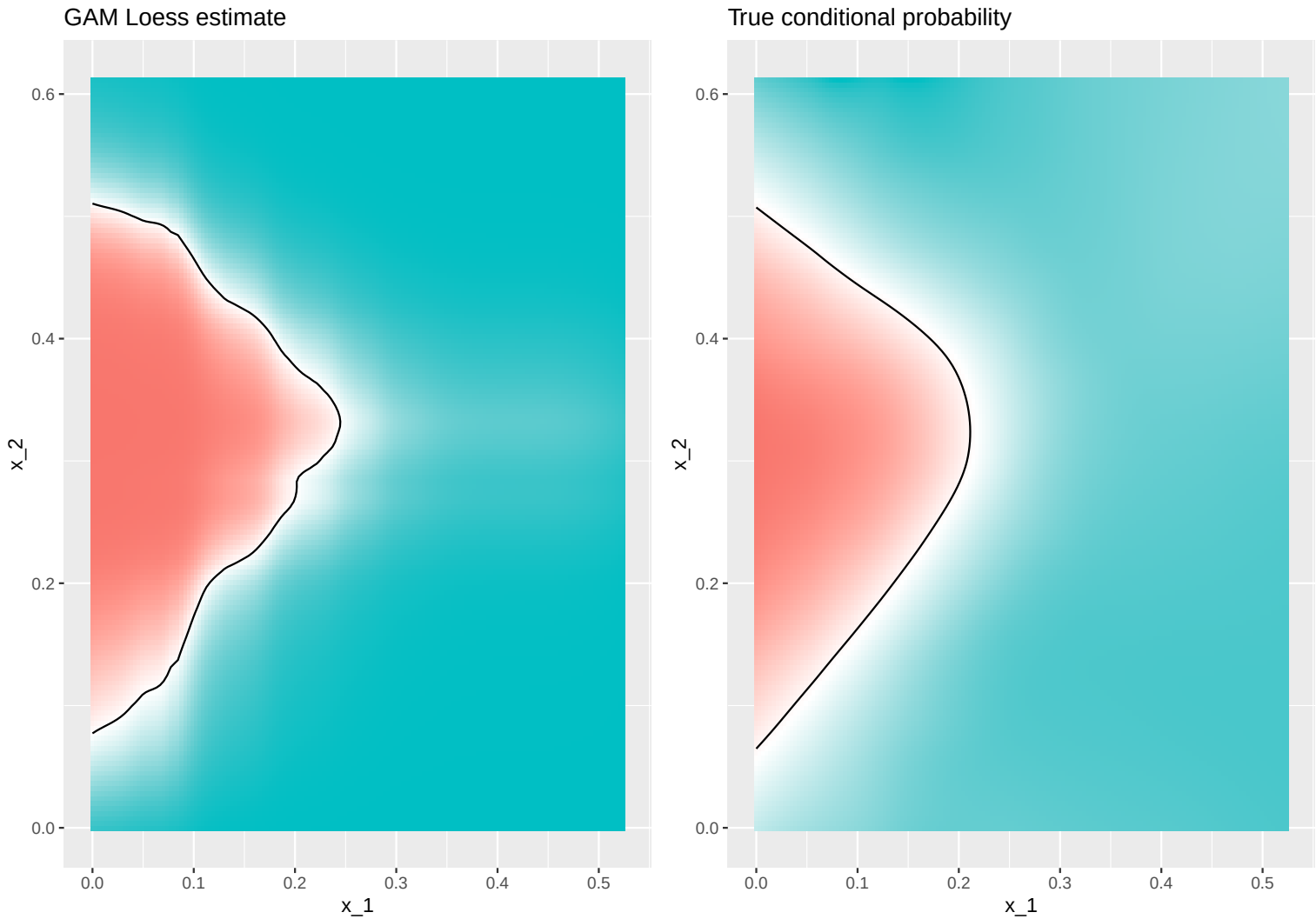
We can see that the method performs similar to kNN:

```
confusionMatrix(data = predict(train_loess, mnist_27$test),
                reference = mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.85
```

Example: fitting with loess

It produces a smoother estimate of the conditional probability:



Session Info

```
sessionInfo()
```

```
## R version 4.2.3 (2023-03-15)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Ventura 13.4.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] splines   stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] gridExtra_2.3   dslabs_0.7.6    gam_1.22-2      foreach_1.5.2
##  [5] lubridate_1.9.2 forcats_1.0.0   stringr_1.5.0   dplyr_1.1.2
##  [9] purrr_1.0.2     readr_2.1.4     tidyr_1.3.0     tibble_3.2.1
## [13] tidyverse_2.0.0 caret_6.0-94    lattice_0.21-8  ggplot2_3.4.3
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.11          listenv_0.9.0        class_7.3-22
##  [4] digest_0.6.33        ipred_0.9-14         utf8_1.2.3
##  [7] parallelly_1.36.0    R6_2.5.1             plyr_1.8.8
## [10] hardhat_1.3.0        stats4_4.2.3         e1071_1.7-13
## [13] evaluate_0.21        pillar_1.9.0         rlang_1.1.1
## [16] rstudioapi_0.15.0    data.table_1.14.8    rpart_4.1.19
## [19] Matrix_1.5-4.1       rmarkdown_2.24       labeling_0.4.2
## [22] gower_1.0.1          munsell_0.5.0        proxy_0.4-27
```