Day 6 - Won't You be my Neighbor?

K-Nearest Neighbors (kNN) for Classification

Today, we will be looking at the k-nearest neighbor(kNN) algorithm for classification. The basic idea behind is:

> In order to choose a class for new data, ask the k nearest points in the training data (majority vote wins).

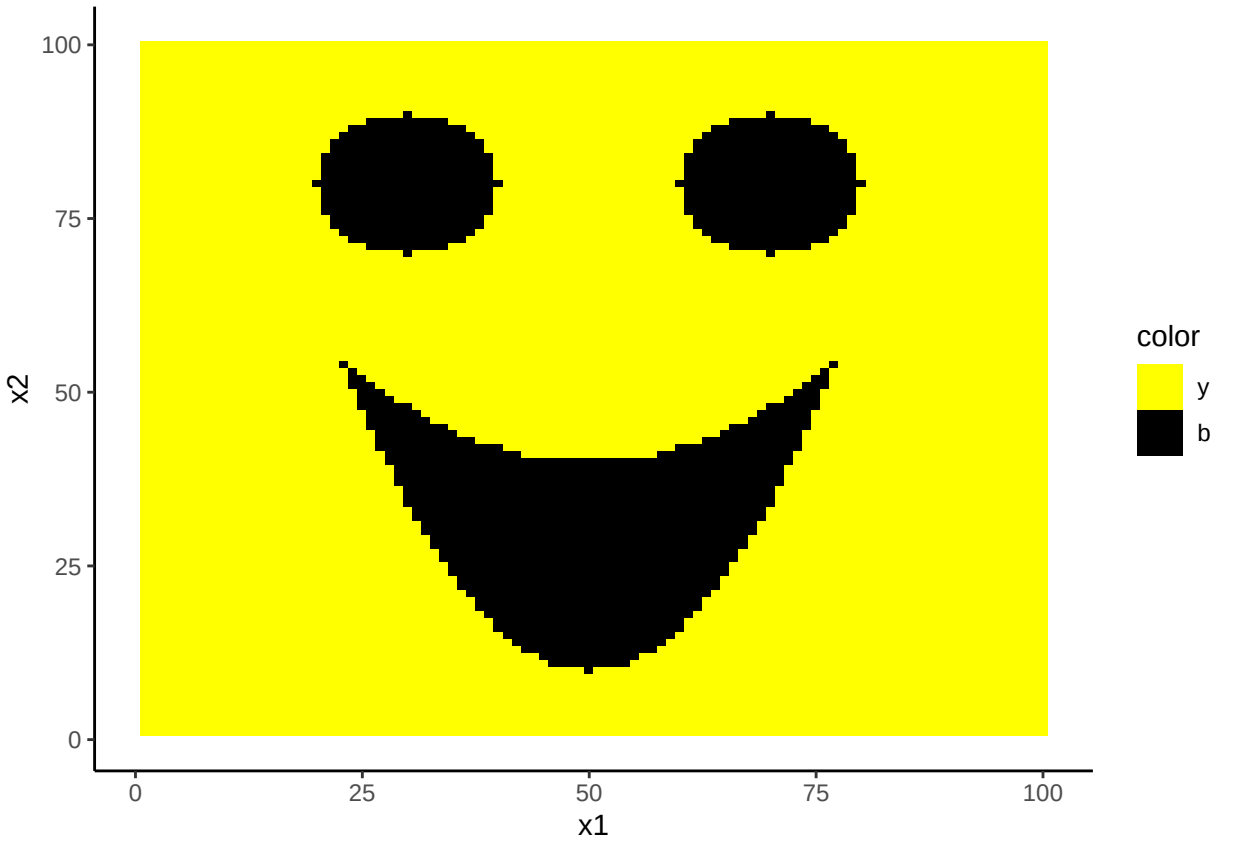
In order use kNN, we need to make two choices:

- How are we measuring near? In math-speak, what is our distance metric?
  - Often, this is euclidean distance (the square root of the sum of squares of the differences)
  - Other metrics (such as Manhattan distance) can also be used.
  - Some application domains have specialized ways of measuring nearness.
- How many neighbors (k) should we ask?
  - For k=1, we ask only the closest point in the data.
  - For k>1, we take a vote of the k closest points. Whichever class has the majority wins. Often, we use odd k for two-class problems so that we don't have ties.
  - Smaller k gives a more flexible model, larger k gives a less flexible model.
  - What might be a good way of choosing k?

Smiley

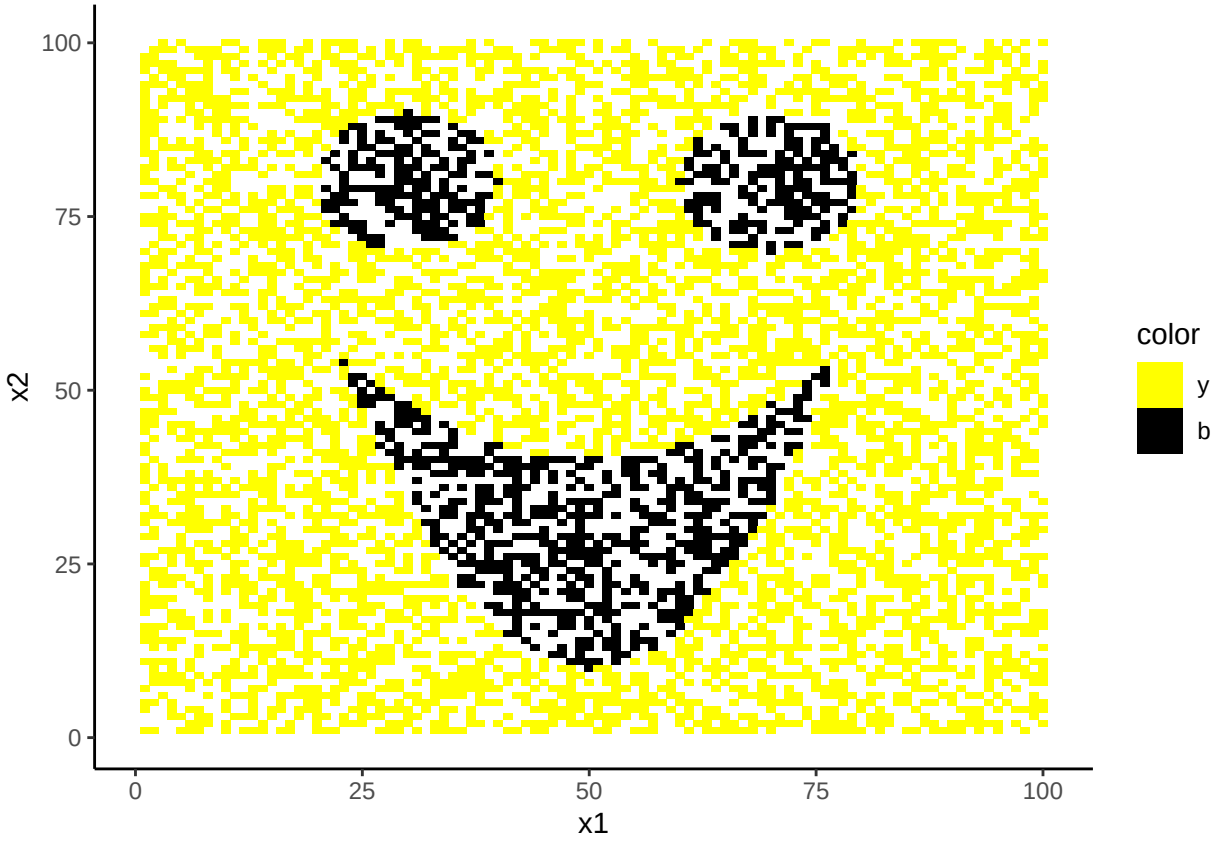
Let's start out with a somewhat silly example inspired by http://blog.yhat.com/posts/classification-using-knn-and-python.html. We are going to make a smiley face:

```
smiley <- expand.grid(x1 = seq(1,100), x2 = seq(1,100)) # creates all combinations of x1,x2
smiley$color <- "y" # creates a new variable and sets all of the values to "yellow"
left_eye <- (smiley$x1 - 30)^2 + (smiley$x2 - 80)^2 <= 100
right_eye <- (smiley$x1 - 70)^2 + (smiley$x2 - 80)^2 <= 100
mouth <- (smiley$x2 - 0.06*(smiley$x1-50)^2 >= 10) & (smiley$x2 - 0.02*(smiley$x1-50)^2 <= 40)
smiley$color[which(left_eye|right_eye|mouth)] <- "b"

ggplot(smiley, mapping = aes(x = x1, y = x2, fill = color)) +
  geom_tile() +
  scale_fill_manual(values = c("yellow","black"), limits = c("y","b")) +
  theme_classic()
```



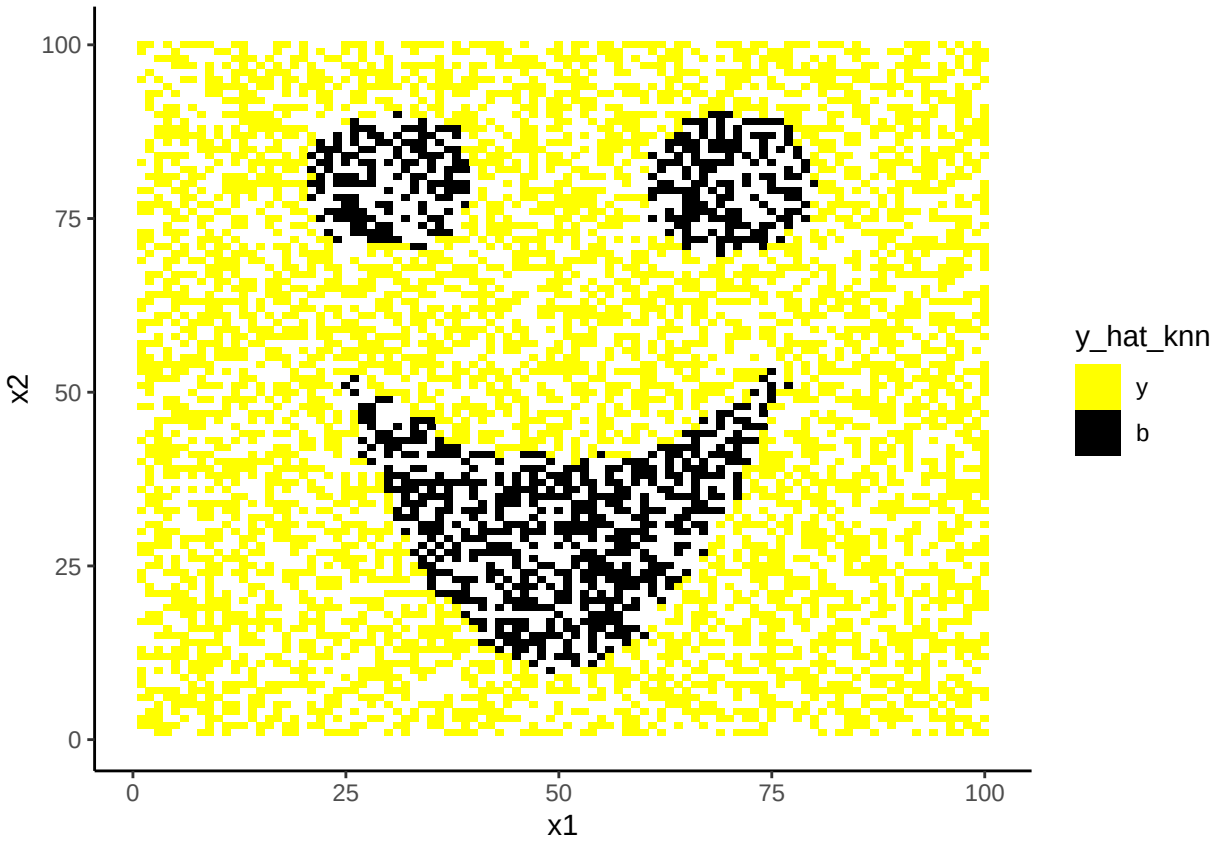
Let's remove 50% of our data (actually we will put it into a test set) as if our smiely face image is corrupted. Can we reconstruct the smiley face using kNN?

```
set.seed(42) # This is for reproducibility, so that everyone gets the same answers.
test_index <- createDataPartition(smiley$color, p = 0.50, list = FALSE)
test_set <- smiley[test_index,]
train_set <- smiley[-test_index,]

ggplot(train_set, mapping = aes(x = x1, y = x2, fill = color)) +
  geom_tile() +
  scale_fill_manual(values = c("yellow","black"), limits = c("y","b")) +
  theme_classic()
```



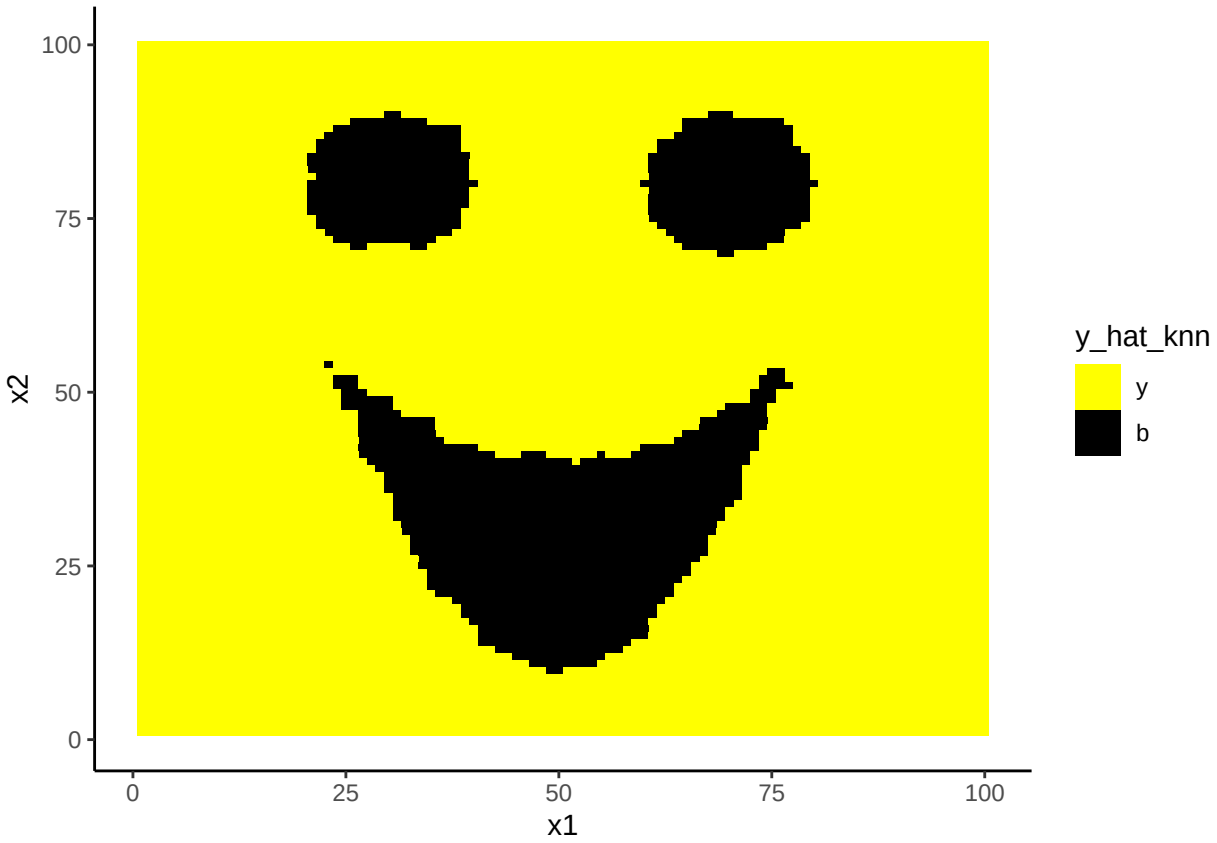
Let's use kNN to reconstruct our missing pixels:

```
model_knn <- train(color ~ .,
                   data = train_set,
                   method = "knn",
                   tuneGrid = data.frame(k=3))
y_hat_knn <- predict(model_knn, newdata = test_set[,1:2])

ggplot(test_set, mapping = aes(x = x1, y = x2, fill = y_hat_knn)) +
  geom_tile() +
  scale_fill_manual(values = c("yellow","black"), limits = c("y","b")) +
  theme_classic()
```



Let's combine the training set with our predictions for the full reconstructed image.

```
ggplot(test_set, mapping = aes(x = x1, y = x2, fill = y_hat_knn)) +
  geom_tile() +
  geom_tile(data = train_set, mapping = aes(x = x1, y = x2, fill = color)) +
  scale_fill_manual(values = c("yellow","black"), limits = c("y","b")) +
  theme_classic()
```



Let's look at the confusion matrix:

```
confusionMatrix(data = y_hat_knn, reference = factor(test_set$color))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    b    y
##          b  833   41
##          y   35 4092
##
##                Accuracy : 0.9848
##                  95% CI : (0.981, 0.988)
##     No Information Rate : 0.8264
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9472
##
##  Mcnemar's Test P-Value : 0.5663
##
##             Sensitivity : 0.9597
##             Specificity : 0.9901
##          Pos Pred Value : 0.9531
##          Neg Pred Value : 0.9915
##              Prevalence : 0.1736
##          Detection Rate : 0.1666
##    Detection Prevalence : 0.1748
```

```
##       Balanced Accuracy : 0.9749
##
##        'Positive' Class : b
##
```

Let's visualize the confusion matrix:

```
match <- test_set$color == y_hat_knn
positive <- y_hat_knn == "b"
cm <- case_when(
  match & positive ~ "TP",
  match & !positive ~ "TN",
  !match & positive ~ "FP",
  !match & !positive ~ "FN",
  TRUE ~ "oops"
)

ggplot(test_set, mapping = aes(x = x1, y = x2, fill = cm)) +
  geom_tile() +
  geom_tile(data = train_set, mapping = aes(x = x1, y = x2, fill = color)) +
  scale_fill_manual(values = c("yellow","black", "blue", "red","yellow","black"), limits = c("TN","TP",
  theme_classic()
```

Let's play around with this a bit:

- How little data can we use to reconstruct the image? Up the value of `p` in `createDataPartition` to remove more pixels.

- What value of k works best? For 80% corruption `p=.80`, test out values of k = 1,3,5,7,9,11.

Cross Validation

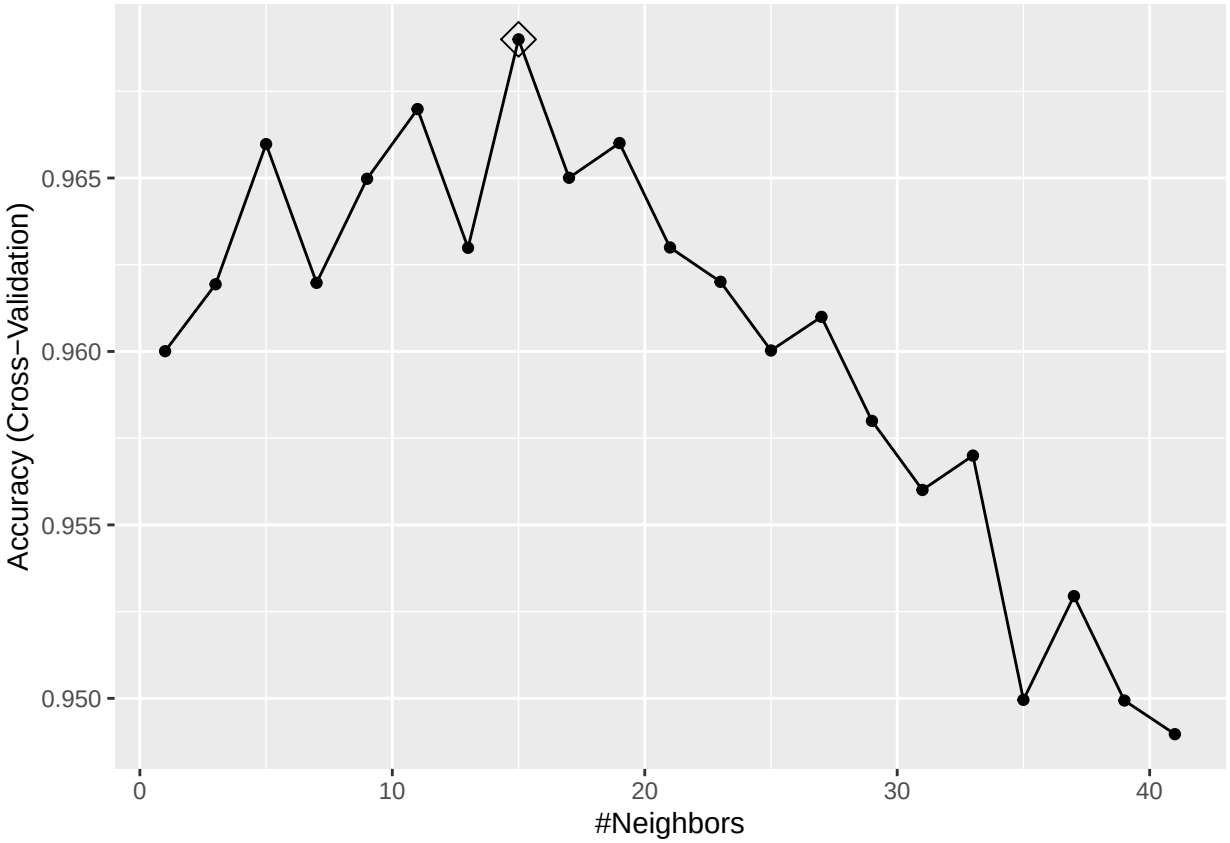
The question, "What value k works best?" is a very typical question in machine learning. The k in kNN is what is commonly called a hyper-parameter. It is a value that controls the performance of the algorithm that needs to be set by the user. Most algorithms have one or more hyper parameters. This raises the question, "How do I choose k (or any other hyper-parameter)?

To do this we need some way of estimating the error rate. We could use our test set to estimate the error rate, but the problem is that if we use the test set to evaluate many models (different values of k), it is likely that our model will fit to the peculiarities of the one test set. The rule for test sets is that they should only be used once (at the very end) to assess model performance. You shouldn't use the test set to make model decisions.

Instead, a common approach is to use K-fold cross validation. The basic idea is that we split our training set into K parts (this K is different from the k nearest neighbors). For each part, we train a model on the other K-1 parts and measure the error on the remaining part that wasn't used for training the model. We repeat this for each of the K parts, then average the errors on each part to get an estimate of the overall error. Typically 5-fold or 10-fold cross-validation is used.

Let's see how to do 10-fold cross-validation with the `caret` package:

```
set.seed(42) # This is for reproducibility, so that everyone gets the same answers.
test_index <- createDataPartition(smiley$color, p = 0.90, list = FALSE)
test_set <- smiley[test_index,]
train_set <- smiley[-test_index,]

control = trainControl(method = "cv", number = 10)
model_knn_cv <- train(color ~ .,
                      data = train_set,
                      method = "knn",
                      tuneGrid = data.frame(k=seq(1,41,2)),
                      trControl = control)
ggplot(model_knn_cv, highlight = TRUE)
```



It looks like k = 15 neighbors is the most accurate choice. The `caret` package knows this and will use k = 15 when we put out model into the predict function:

```
y_hat_knn_cv <- predict(model_knn_cv, newdata = test_set[,1:2])
confusionMatrix(data = y_hat_knn_cv, reference = factor(test_set$color))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    b    y
##          b 1295   46
##          y  267 7393
##
##                Accuracy : 0.9652
##                  95% CI : (0.9612, 0.9689)
##     No Information Rate : 0.8265
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8716
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8291
##             Specificity : 0.9938
##          Pos Pred Value : 0.9657
##          Neg Pred Value : 0.9651
##              Prevalence : 0.1735
```

```
##          Detection Rate : 0.1439
##    Detection Prevalence : 0.1490
##       Balanced Accuracy : 0.9114
##
##        'Positive' Class : b
##
```

Let's visualize this confusion matrix:

```
match <- test_set$color == y_hat_knn_cv
positive <- y_hat_knn_cv == "b"
cm <- case_when(
  match & positive ~ "TP",
  match & !positive ~ "TN",
  !match & positive ~ "FP",
  !match & !positive ~ "FN",
  TRUE ~ "oops"
)

ggplot(test_set, mapping = aes(x = x1, y = x2, fill = cm)) +
  geom_tile() +
  geom_tile(data = train_set, mapping = aes(x = x1, y = x2, fill = color)) +
  scale_fill_manual(values = c("yellow","black", "blue", "red","yellow","black"), limits = c("TN","TP",
  theme_classic()
```

Cross-validation with Cohen's kappa

But wait, didn't we come to the conclusion last time that accuracy might not be the best measure of performance. Yes, we did. We could use the F_1 measure, but it is little complicated to setup, see https://stackoverflow.com/questions/37666516/caret-package-custom-metric. Instead, we will use Cohen's kappa, which is another method that is robust under class imbalance (see https://en.wikipedia.org/wiki/Cohen's_kappa for details). Cohen's kappa compares the observed accuracy to the accuracy that you would expect from guessing. The formula is:

$$\kappa = (accuracy - accuracy_{exp})/(1 - accuracy_{exp})$$

where

$$accuracy_{exp} = (n_{1,pred} * n_{1,actual} + n_{2,pred} * n_{2,actual})/n^2$$

A value of kappa = 1 means perfect accuracy, and a value of kappa = 0 means that the accuracy is only what we would expect by chance. It is even possible to get negative values for kappa when the accuracy is worse than what we would expect by chance.

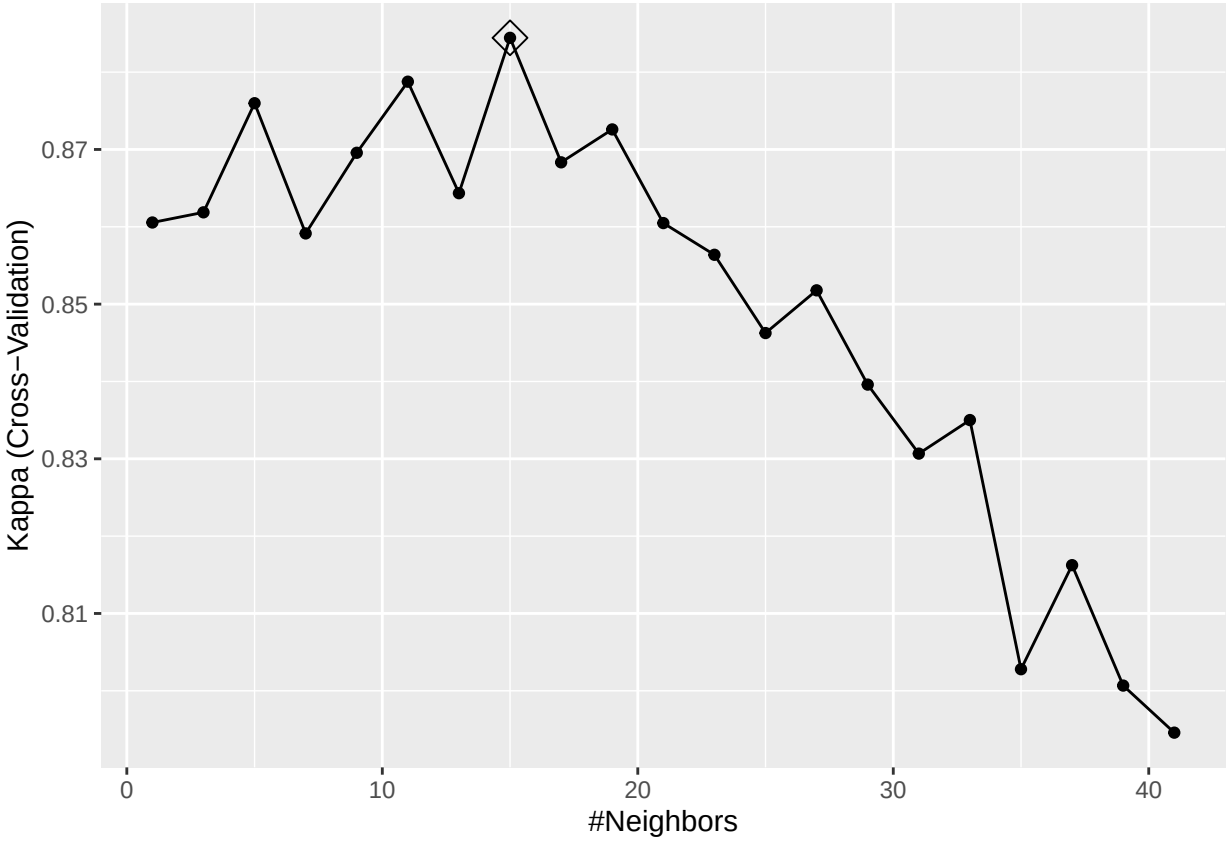
To use Cohen's kappa in our cross-validation, we specify `metric = Kappa` inside our train command (note the capital K in Kappa):

```
set.seed(42) # This is for reproducibility, so that everyone gets the same answers.
test_index <- createDataPartition(smiley$color, p = 0.90, list = FALSE)
test_set <- smiley[test_index,]
train_set <- smiley[-test_index,]

control = trainControl(method = "cv", number = 10)
model_knn_cv <- train(color ~ .,
                      data = train_set,
                      method = "knn",
                      tuneGrid = data.frame(k=seq(1,41,2)),
                      trControl = control,
                      metric = "Kappa")
model_knn_cv
```

```
## k-Nearest Neighbors
##
## 999 samples
##   2 predictor
##   2 classes: 'b', 'y'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 899, 898, 899, 899, 899, 898, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.9600081  0.8605557
##    3  0.9619376  0.8618699
##    5  0.9659778  0.8759791
##    7  0.9619780  0.8591407
##    9  0.9649780  0.8695703
##   11  0.9669881  0.8787617
##   13  0.9629879  0.8643470
```

```
##    15  0.9689980  0.8844347
##    17  0.9650079  0.8683358
##    19  0.9660079  0.8725908
##    21  0.9629980  0.8604795
##    23  0.9620079  0.8563788
##    25  0.9600281  0.8462612
##    27  0.9609978  0.8517837
##    29  0.9579978  0.8395940
##    31  0.9560079  0.8306684
##    33  0.9569978  0.8350081
##    35  0.9499578  0.8027899
##    37  0.9529479  0.8162468
##    39  0.9499378  0.8006733
##    41  0.9489677  0.7945882
##
## Kappa was used to select the optimal model using the largest value.
## The final value used for the model was k = 15.
```

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



It looks like k=15 is still the best number of neighbors when using kappa as our preformance measure.

Class Probabilities

An alternate way of thinking about kNN is that we are approximating the class probabilities by looking at the proportion of neighbors in that class. That is, if we have a data point x_0 we can approximate the probability that x_0 has class 1 by:

$$Pr(Y = 1|X = x_0) \approx \frac{1}{k} \sum_{i \in neighbors} I(y_i = 1)$$

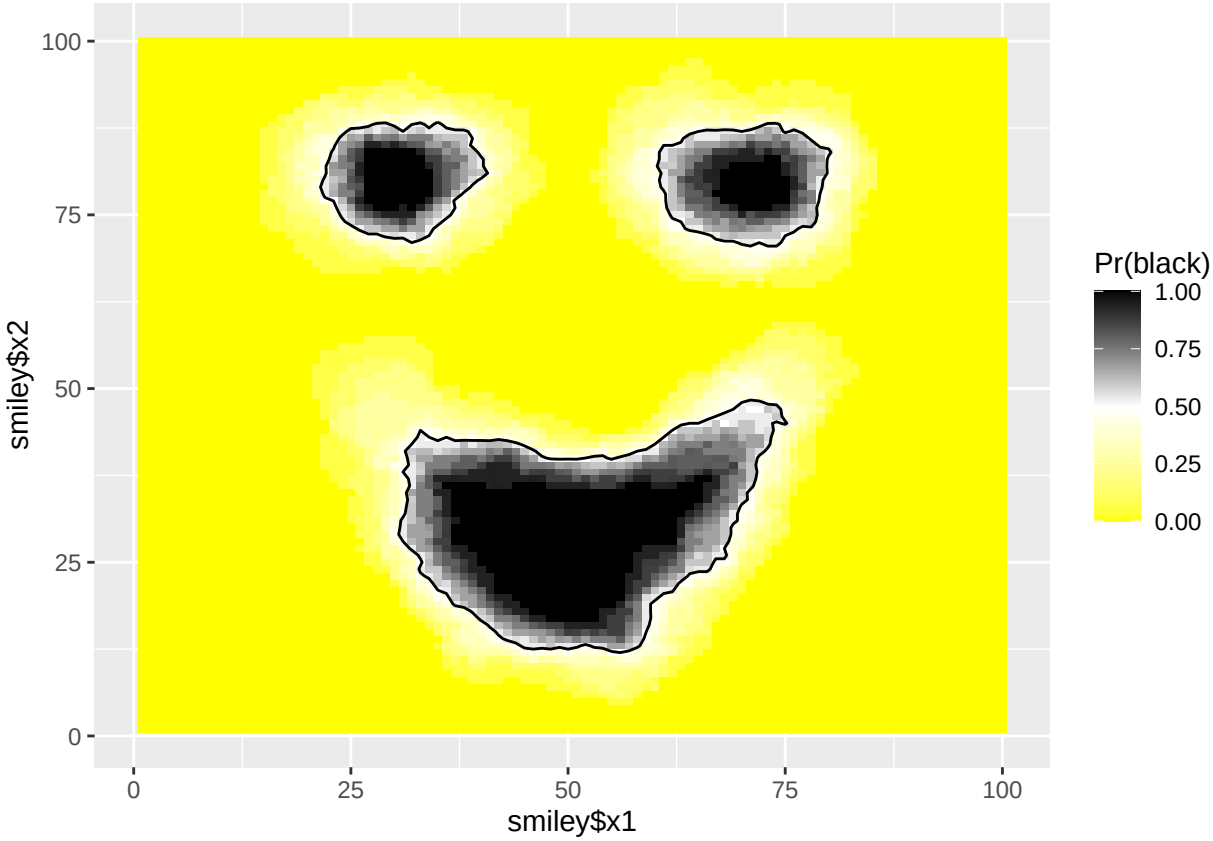
We then classify x_0 to the class with the highest probability.

If we ask `caret` is willing to compute these approximate class probabilities for us. We just need to add `type = "prob"` to the predict command. This time, I will give the whole grid of (x1,x2) points, so that we can create a probability map.

```
p_hat_knn_cv <- predict(model_knn_cv, newdata = smiley[,1:2], type = "prob")
str(p_hat_knn_cv)
```

```
## 'data.frame':    10000 obs. of  2 variables:
##  $ b: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ y: num  1 1 1 1 1 1 1 1 1 1 ...
```

```
ggplot(p_hat_knn_cv, mapping = aes(x = smiley$x1, y = smiley$x2, z = b, fill = b)) +
  geom_raster()+
  scale_fill_gradientn(colors=c("yellow","white","black"), name = "Pr(black)") +
  stat_contour(breaks=c(0.5),color="black")
```

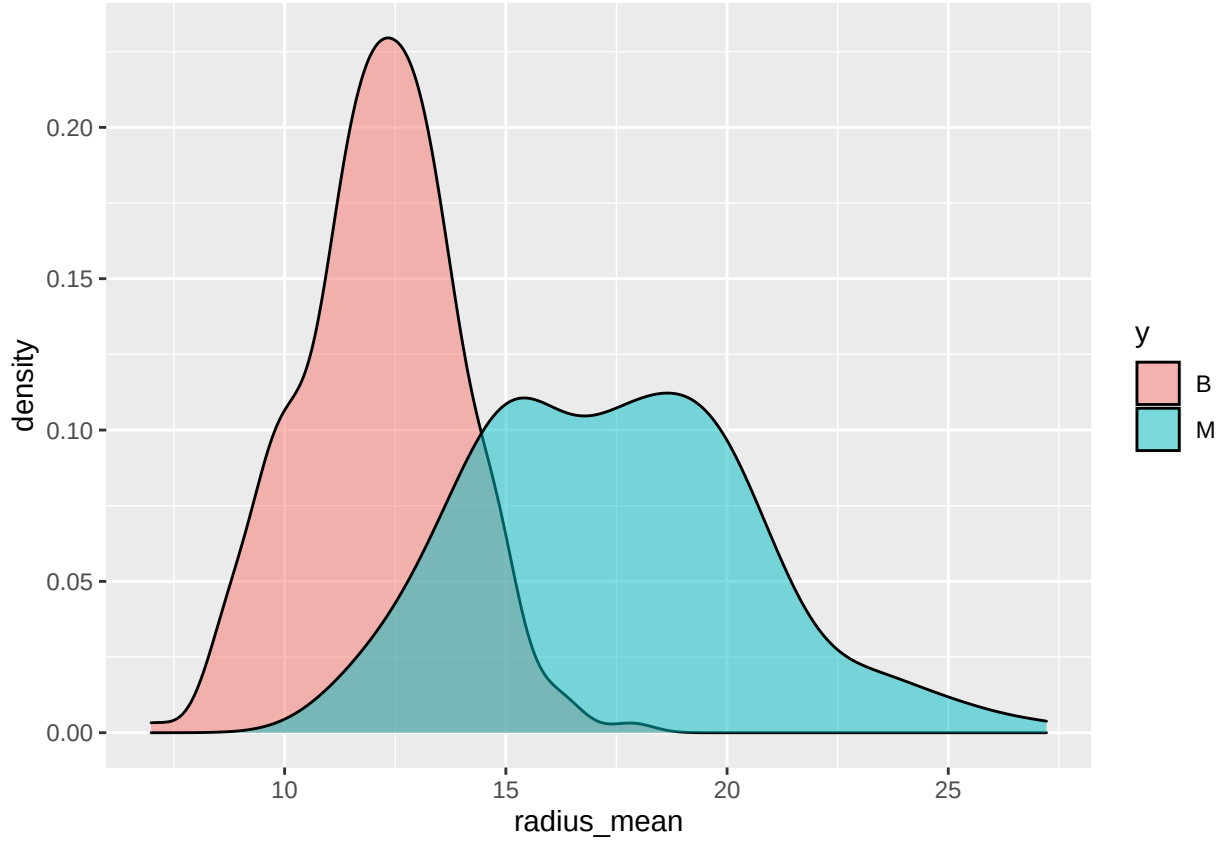


Homework

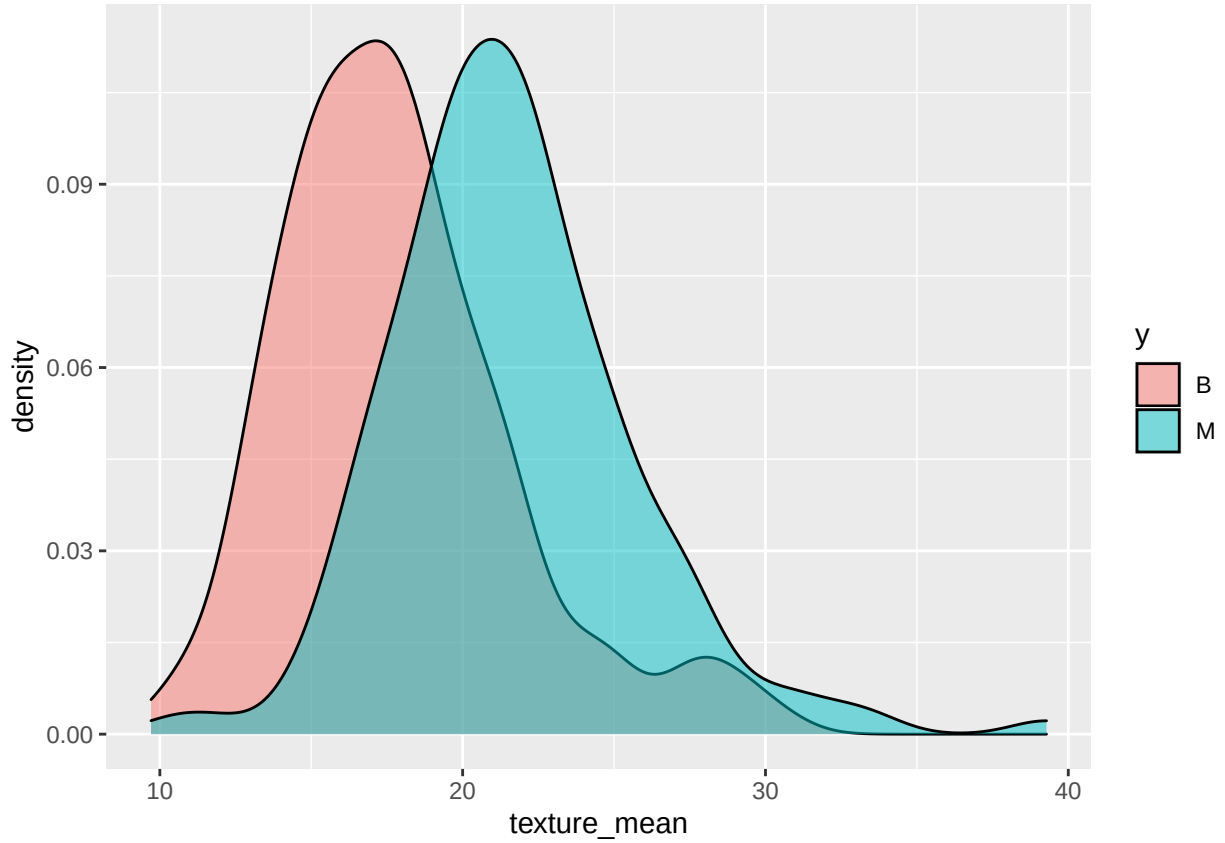
Let's return to the brca dataset. Now, we will use the predictors: mean radius and mean texture.

```
data(brca)
brca_data <- data.frame(brca$x) %>%
  select(radius_mean, texture_mean) %>%
```

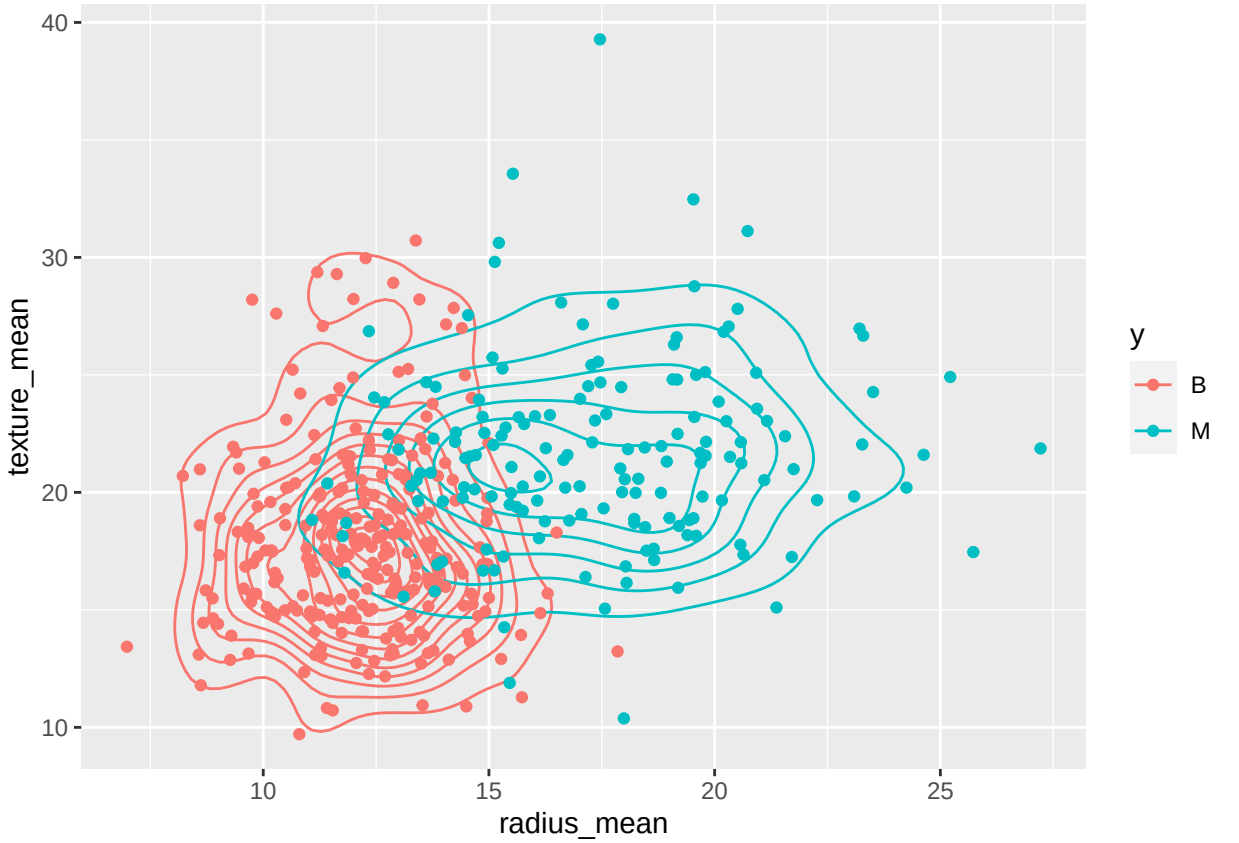
```
  data.frame(y = brca$y)

set.seed(42)
test_index <- createDataPartition(brca_data$y, p = 0.25, list = FALSE)
test_brca <- brca_data[test_index,]
train_brca <- brca_data[-test_index,]

ggplot(train_brca)+
  geom_density(mapping = aes(x =  radius_mean, fill = y), alpha = 0.5)
```




```
ggplot(train_brca)+
  geom_density(mapping = aes(x =  texture_mean, fill = y), alpha = 0.5)
```




```
ggplot(train_brca)+
  geom_density2d(mapping = aes(x = radius_mean, y = texture_mean, color = y))+
  geom_point(mapping = aes(x = radius_mean, y = texture_mean, color = y))
```



Problem 1

Create a kNN model using k = 1 neighbor to predict whether the mass is malignant or benign. Visualize the model results on the test set using color for the actual class and shape for whether the point is correctly. Also, compute the confusion matrix (treating malignant as the positive class).

```
## Your code goes here.
```

Problem 2

For the k=1 model, create a probability map for the probability that the mass is malignant. The command `brca_grid <- expand.grid(radius_mean = seq(7,30,0.1), texture_mean = seq(10,35,0.1))` will create a grid of radius and texture values that you can use as new data to predict probabilities. Since k=1, what values can the predicted probability take?

```
## Your code goes here.
```

Problem 3

Create a kNN model using k = 11 neighbors to predict whether the mass is malignant or benign. Visualize the model results on the test set using color for the actual class and shape for whether the point is correctly. Also, compute the confusion matrix (treating malignant as the positive class).

```
## Your code goes here.
```

Problem 4

For the k=11 model, create a probability map for the probability that the mass is malignant. You can reuse the `brca_grid` from problem 2. Since k=11, what values can the predicted probability take?

```
## Your code goes here.
```

Problem 5

Use 10-fold cross-validation (with Kappa as the metric) to select the best value for the number of neighbors (use k = seq(1,41,2)). What value of k gives the best Kappa?

```
## Your code goes here.
```

Problem 6

Visualize the model results (using the best k) on the test set using color for the actual class and shape for whether the point is correctly. Also, compute the confusion matrix (treating malignant as the positive class).

```
## Your code goes here.
```

Problem 7

For the best k model, create a probability map for the probability that the mass is malignant.

```
## Your code goes here.
```