HW6

Hengle Li

3/6/2020

```
library(tidyverse)
library(caret)
library(rsample)
library(tidymodels)
library(pROC)
library(kernlab)
library(rcfss)
library(e1071)
library(patchwork)

set.seed(6758)
```

Conceptual

Non-linear separation

```
df_train <- tibble(X1 = runif(100, min = 0, max = 20),
                   X2 = runif(100, min = 0, max = 20),
                   #Y is not a linear function of X1, X2
                   Y = X1 * X2)

df_train <- df_train %>%
  mutate(result = ifelse(Y <= 90, "fail", "success")) %>%
  select(X1, X2, result) %>%
  mutate(result = as.factor(result))

df_test <- tibble(X1 = runif(100, min = 0, max = 20),
                  X2 = runif(100, min = 0, max = 20),
                  #Y is not a linear function of X1, X2
                  Y = X1 * X2)

df_test <- df_test %>%
  mutate(result = ifelse(Y <= 90, "fail", "success")) %>%
  select(X1, X2, result) %>%
  mutate(result = as.factor(result))
```

```
cv_10 <- trainControl(method = "cv",
                      number = 10,
```

```
                      savePredictions = "final",
                      classProbs = TRUE)

#for train()
X1_train <- df_train %>% dplyr::select(X1, X2)
Y1_train <- df_train$result

X1_test <- df_test %>% dplyr::select(X1, X2)
Y1_test <- df_test$result
```

```
#in the train() here, parameters are not set
#we can extract the best tuned parameters later
svm_radial <- train(
  x = X1_train,
  y = Y1_train,
  method = "svmRadial",
  trControl = cv_10
)
```

```
#same as above
svm_linear <- train(
  x = X1_train,
  y = Y1_train,
  method = "svmLinear",
  trControl = cv_10
)
```

```
#svm models contain training data and corresponding predictions
predr_train <- tibble(truth = Y1_train,
                      pred = predict(svm_radial, newdata = X1_train))

pred_radial_train <- predr_train %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 100)

pred_radial_train$accuracy
```

```
## [1] 100
```

```
predl_train <- tibble(truth = Y1_train,
                      pred = predict(svm_linear, newdata = X1_train))

pred_linear_train <- predl_train %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 100)

pred_linear_train$accuracy
```

```
## [1] 95
```

It appears with the training data, svm with a radial kernel can reach 100 percent accuracy, while svm with a linear kernel can reach only 95 percent accuracy. Although 100 % is unusual, svm with a radial kernel does outperform svm with linear kernel here.

- Normally, `geom_contour()` should be doing a fine job in demonstrating the data points and decision boundary. But I have a problem doing that. The issue was posted online for help here
- To compare the models in an alternative way, we can extract the parameters from train() and use them to fit an SVM with svm() and then make the plots. The resulting two models should be basically the same as produced by train()

```
svm_radial <- svm(result ~.,
                  data = df_train,
                  kernel = "radial",
                  cost = svm_radial$bestTune$C,
                  sigma = svm_radial$bestTune$sigma)

svm_linear <- svm(result ~.,
                  data = df_train,
                  kernel = "linear",
                  cost = svm_linear$bestTune$C)
```

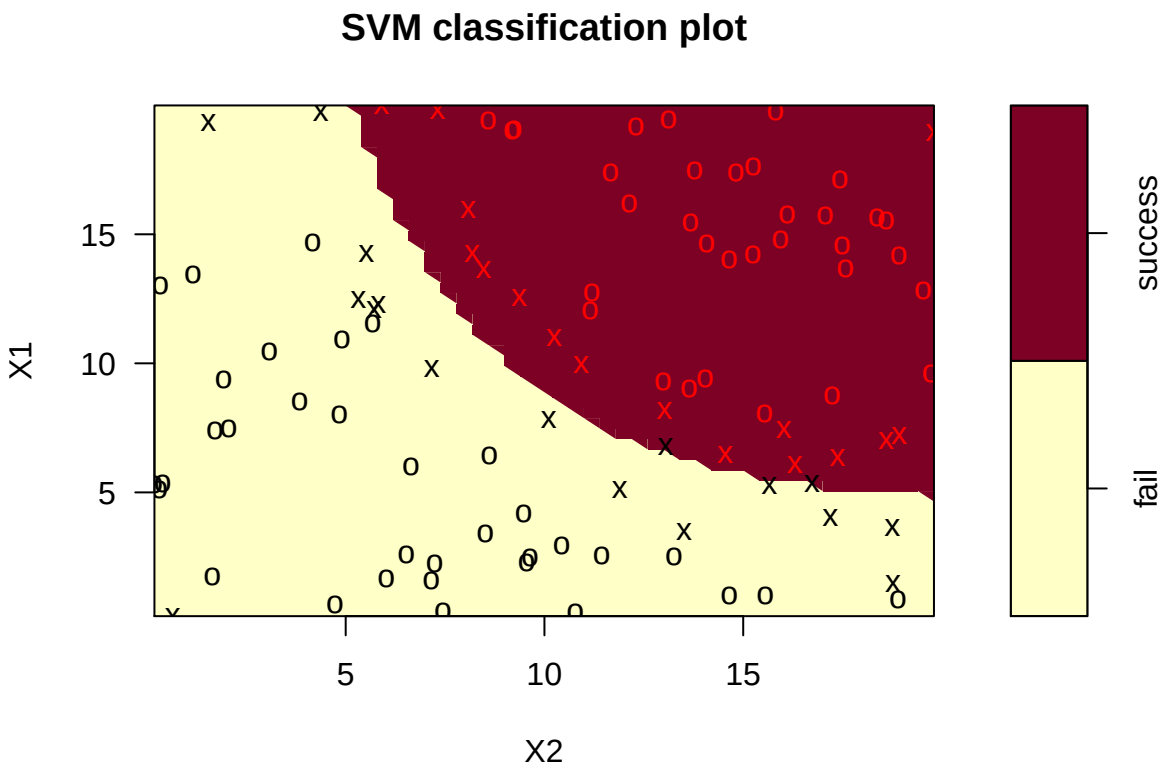
```
pred_radial_test <- tibble(truth = Y1_test,
                           pred = predict(svm_radial, newdata = X1_test))

radial_rate1 <- pred_radial_test %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 100)

radial_rate1$accuracy
```

```
## [1] 99
```

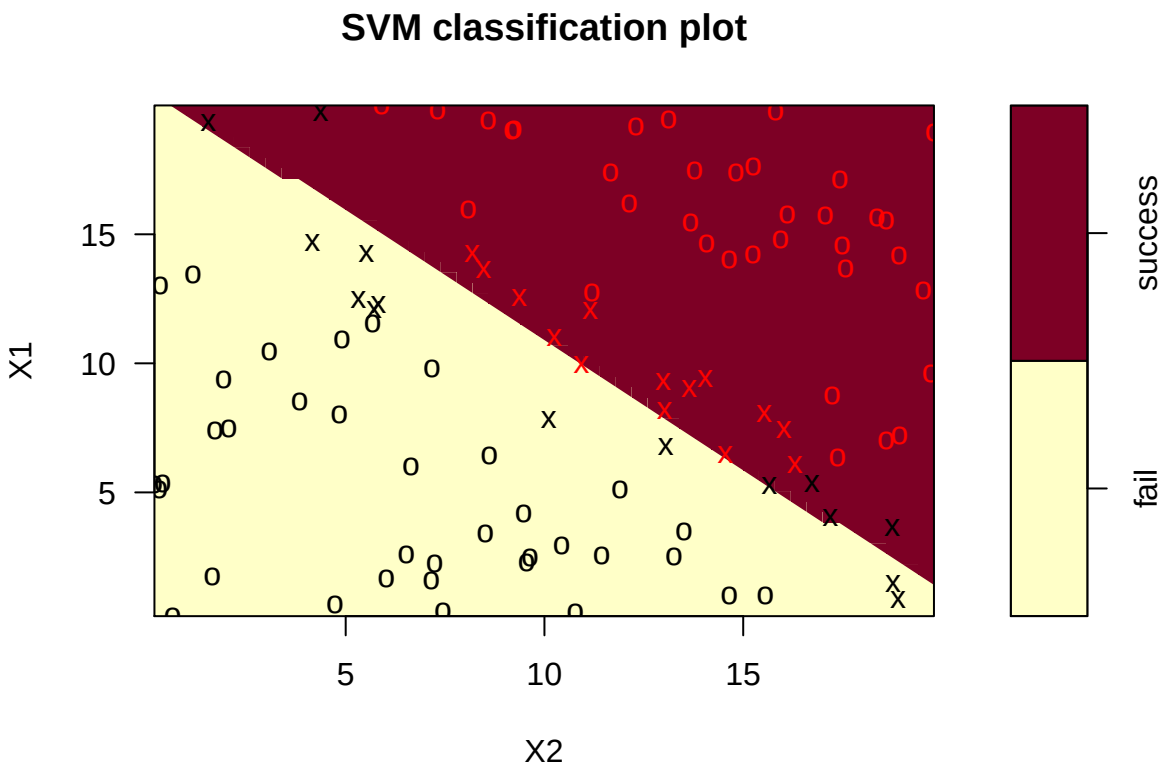
```
pred_linear_test <- tibble(truth = Y1_test,
                           pred = predict(svm_linear, newdata = X1_test))

linear_rate1 <- pred_linear_test %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 100)

linear_rate1$accuracy
```

```
## [1] 92
```

```
plot(svm_radial, df_train)
```

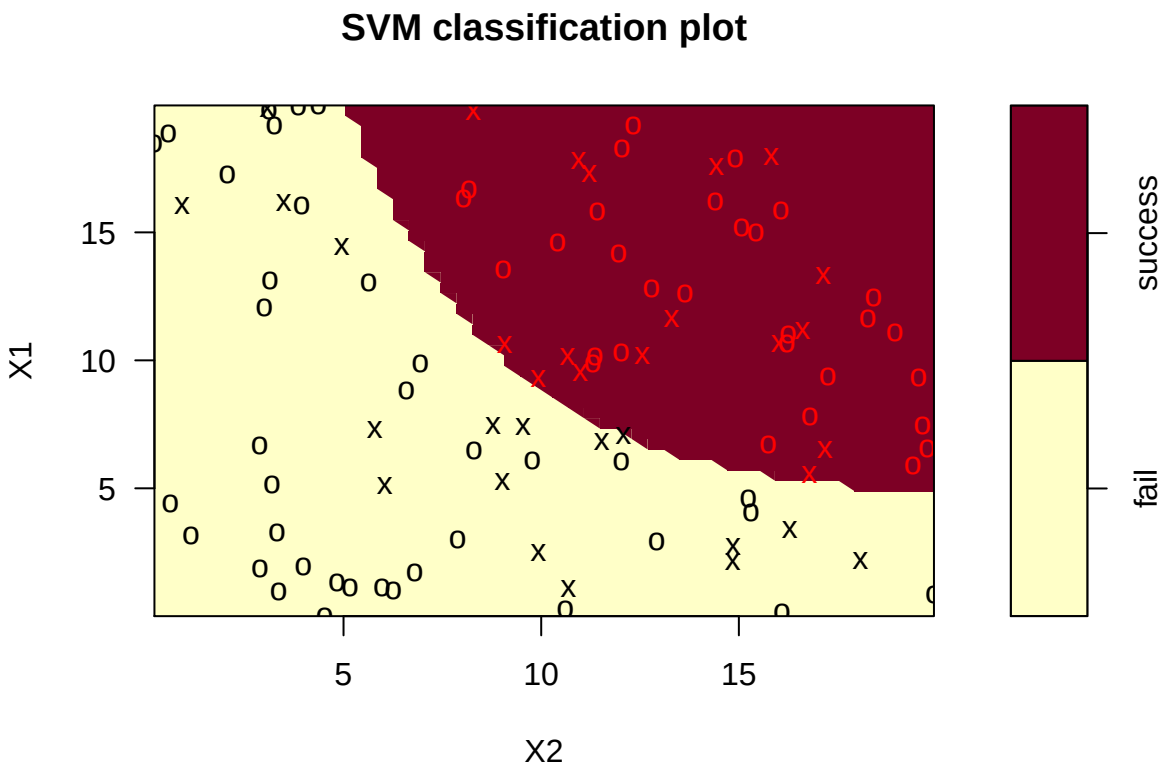



```
plot(svm_linear, df_train)
```

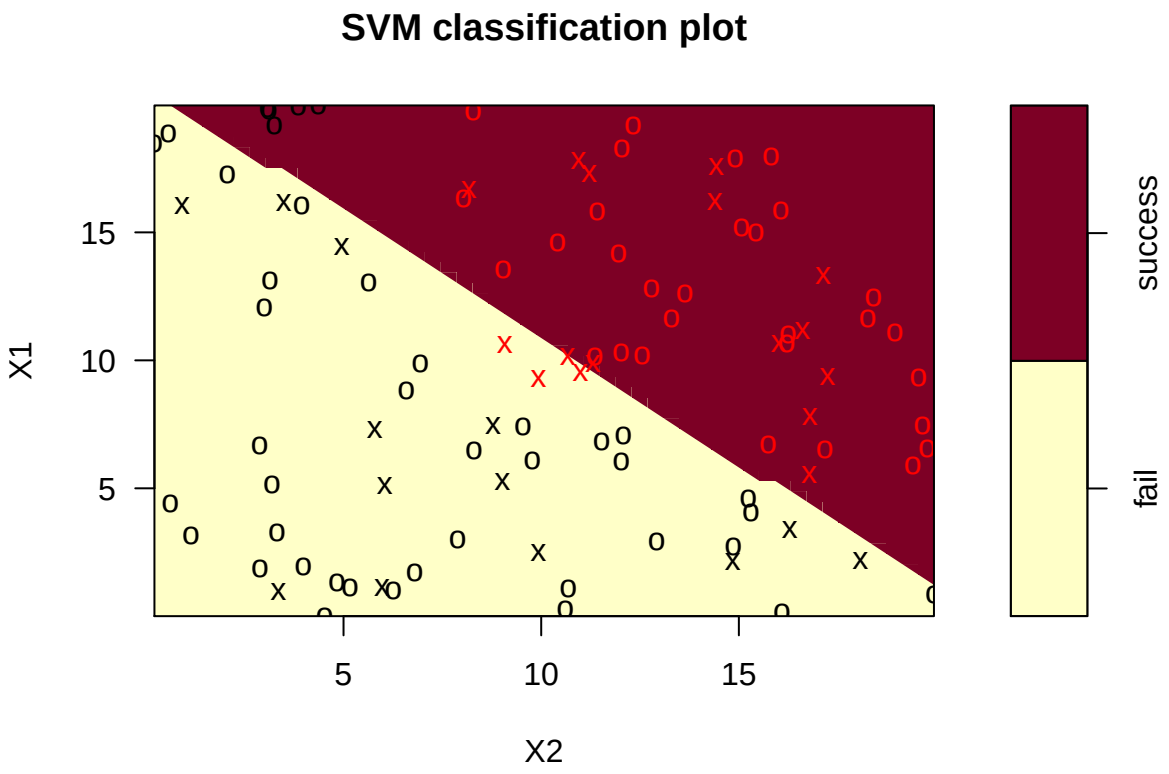


As we can see, in training performance, although both models perform remarkably well, despite that svm with a linear kernel is slightly worse than svm with a radial kernel. That is no wonder, however, because the decision boundary is not linear in the first place.

```
plot(svm_radial, df_test)
```




```
plot(svm_linear, df_test)
```



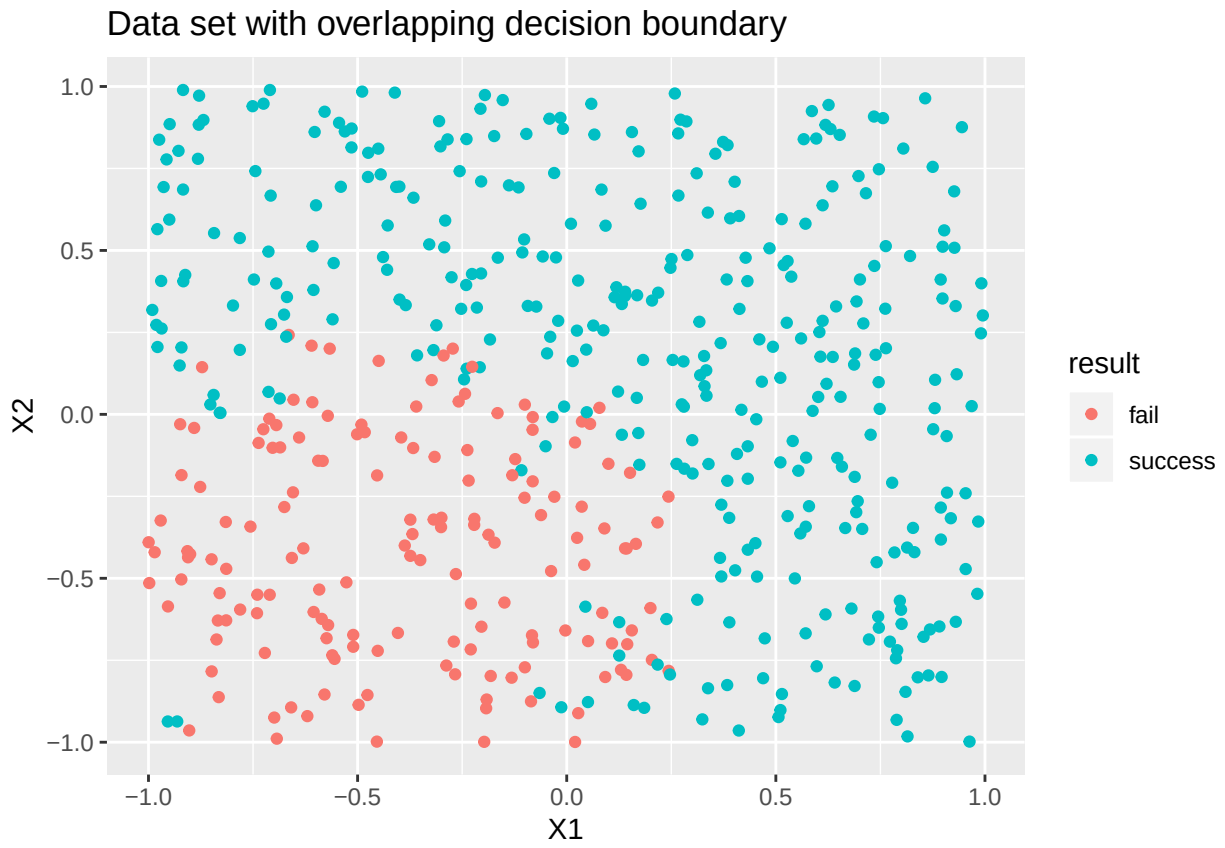
Again, the accuracy of svm with a radial kernel is surprisingly high. Its test accuracy rate is as high as 99 percent. In contrast, the test accuracy rate of svm with a linear kernel is 92 percent. Although the latter test accuracy rate is still quite high, svm with a radial kernel nonetheless outperforms svm with a linear kernel.

SVM vs. logistic regression

```
df2_train <- tibble(X1 = runif(500, -1, 1),
                    X2 = runif(500, -1, 1),
                    e = rnorm(500, 0, 0.2)) %>%
  mutate(log_Y = X1 + X1^2 + X2 + X2^2 + e) %>%
  mutate(Y = logit2prob(log_Y)) %>%
  mutate(result = ifelse(Y > 0.5, "success", "fail"))

df2_test <- tibble(X1 = runif(500, -1, 1),
                    X2 = runif(500, -1, 1),
                    e = rnorm(500, 0, 0.2)) %>%
  mutate(log_Y = X1 + X1^2 + X2 + X2^2 + e) %>%
  mutate(Y = logit2prob(log_Y)) %>%
  mutate(result = ifelse(Y > 0.5, "success", "fail"))
```

```
df2_train %>%
  ggplot(aes(x = X1, y = X2, color = result)) +
```

```
  geom_point() +
  ggtitle(label = "Data set with overlapping decision boundary")
```




```
#for training
X2_train <- df2_train %>% dplyr::select(X1, X2)
Y2_train <- df2_train$result
#for testing
X2_test <- df2_test %>% dplyr::select(X1, X2)
Y2_test <- df2_test$result

lm_mod2 <- train(
  x = X2_train,
  y = Y2_train,
  trControl = cv_10,
  method = "glm",
  family = "binomial"
)
```

```
pred_lm <- tibble(X1 = df2_train$X1,
                  X2 = df2_train$X2,
                  truth = df2_train$result,
                  pred = predict(lm_mod2, newdata = df2_train))

graph_lm <- pred_lm %>% ggplot(aes(x = X1, y = X2, color = pred)) +
  geom_point() +
```

```
  labs(title = "Predictions with logistic regression")
```

```
poly_mod <- train(
  result ~ poly(X1, 2) + poly(X2, 2),
  data = df2_train,
  trControl = cv_10,
  method = "glm",
  family = "binomial"
)
```

```
pred_poly <- tibble(X1 = df2_train$X1,
                    X2 = df2_train$X2,
                    truth = df2_train$result,
                    pred = predict(poly_mod, newdata = df2_train))

graph_poly <- pred_poly %>% ggplot(aes(x = X1, y = X2, color = pred)) +
  geom_point() +
  labs(title = "Predictions with polynomial logistic regression")
```

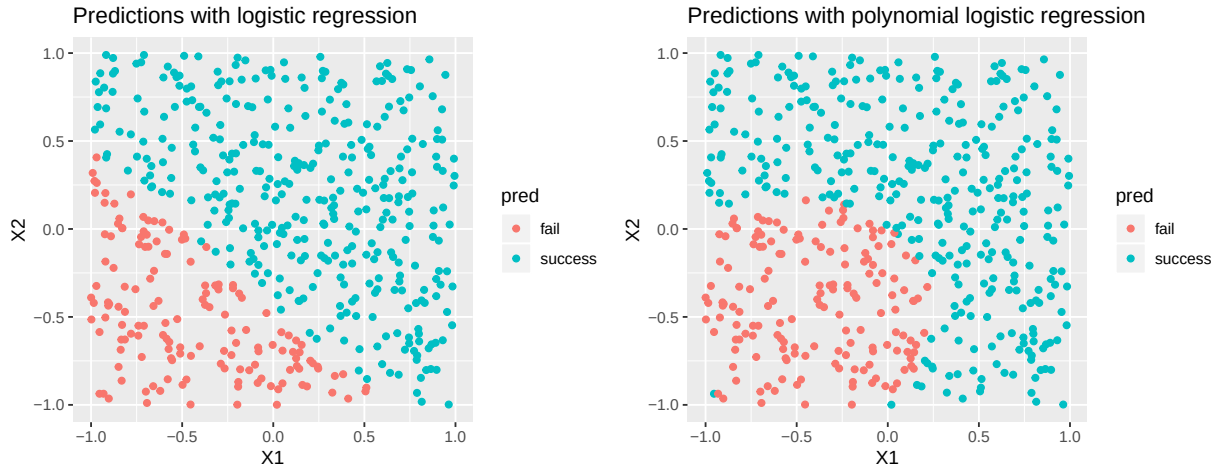
```
svm_linear2 <- train(
  x = X2_train,
  y = Y2_train,
  method = "svmLinear",
  trControl = cv_10
)
```

```
pred_svml2 <- tibble(X1 = df2_train$X1,
                     X2 = df2_train$X2,
                     truth = df2_train$result,
                     pred = predict(svm_linear2, newdata = X2_train))

graph_sl <- pred_svml2 %>%
  ggplot(aes(x = X1, y = X2, color = pred)) +
  geom_point() +
  labs(title = "Predictions with linear svm")
```

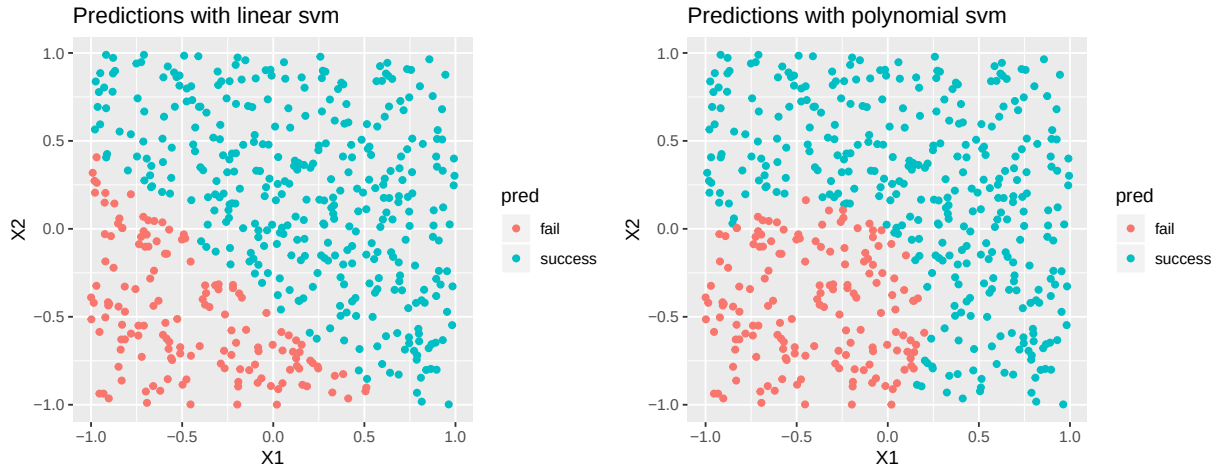
```
svm_radial2 <- train(
  x = X2_train,
  y = Y2_train,
  method = "svmRadial",
  trControl = cv_10
)
```

```
pred_svmp2 <- tibble(X1 = df2_train$X1,
                     X2 = df2_train$X2,
                     truth = df2_train$result,
                     pred = predict(svm_radial2, newdata = X2_train))

graph_sp <- pred_svmp2 %>%
  ggplot(aes(x = X1, y = X2, color = pred)) +
  geom_point() +
  labs(title = "Predictions with polynomial svm")
```

```
graph_lm + graph_poly
```




```
graph_sl + graph_sp
```



By comparing the distributions of linear models and polynomial models, we can see that logistic regression and supported vector machine produce similar decision boundaries. Specifically, linear models produce linear decision boundaries, and polynomial models produce non-linear boundaries.

```
lm_acc2_tr <- pred_lm %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

poly_acc2_tr <- pred_poly %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

svml_acc_tr <- pred_svml2 %>%
  count(truth == pred) %>%
```

```
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

svmp_acc_tr <- pred_svmp2 %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

lm_acc2_tr$accuracy
```

```
## [1] 85.4
```

```
poly_acc2_tr$accuracy
```

```
## [1] 93.8
```

```
svml_acc_tr$accuracy
```

```
## [1] 85.2
```

```
svmp_acc_tr$accuracy
```

```
## [1] 93.8
```

```
lm_acc2_tt <- tibble(X1 = df2_test$X1,
                     X2 = df2_test$X2,
                     truth = df2_test$result,
                     pred = predict(lm_mod2, newdata = X2_test)) %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

poly_acc2_tt <- tibble(X1 = df2_test$X1,
                     X2 = df2_test$X2,
                     truth = df2_test$result,
                     pred = predict(poly_mod, newdata = X2_test)) %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

svml_acc2_tt <- tibble(X1 = df2_test$X1,
                     X2 = df2_test$X2,
                     truth = df2_test$result,
                     pred = predict(svm_linear2, newdata = X2_test)) %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

svmp_acc2_tt <- tibble(X1 = df2_test$X1,
                     X2 = df2_test$X2,
```

```
                    truth = df2_test$result,
                    pred = predict(svm_radial2, newdata = X2_test)) %>%
  count(truth == pred) %>%
  spread("truth == pred", n) %>%
  mutate(accuracy = 100* `TRUE` / 500)

lm_acc2_tt$accuracy
```

```
## [1] 87.2
```

```
poly_acc2_tt$accuracy
```

```
## [1] 90.8
```

```
svml_acc2_tt$accuracy
```

```
## [1] 87
```

```
svmp_acc2_tt$accuracy
```

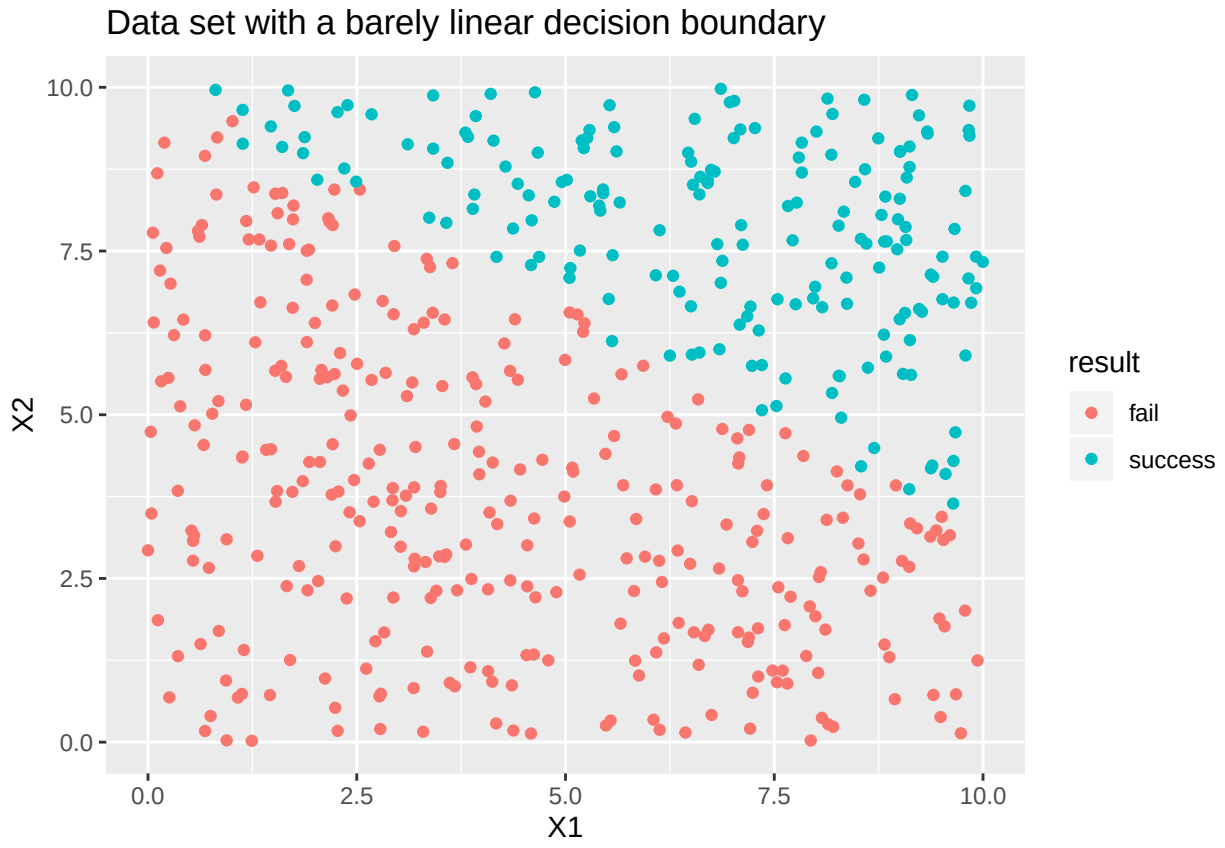
```
## [1] 90.8
```

- Calculation of the accuracy rates shows that in estimating non-linear decision boundaries, models with linear components only are outperformed by their counterparts with non-linear components in both training and testing.
- In logistic regression, the polynomial model still outperform the simple linear model by 3.6 %. In supported vector machine, the accuracy rate between linear kernel and radial kernel is 3.8 %.
- This result is hardly surprising, because the decision boundary, though overlapped, is non-linear. That gives an advantage to polynomial and non-linear models. Perhaps a proper strategy in finding an appropriate model is to first examine the data, and decide whether the decision boundary is linear.
- Moreover, in both logistic regression and svm, models with non-linear components take longer to fit than their linear counterparts. That suggests even though they may be more accurate and efficient in estimating non-linear decision boundaries, they are more complicated, and therefore harder to interpret in actual applications.
- Between logistic regression and support vector machine, there's little difference in this example. Logistic regression should perform better than SVM when there is a large number of features but a small sample; SVM should perform better than logistic regression when there is a smally number of features but a large sample.
- However, in this case, both the number of features and the sample size are small. It's hard to tell which approach performs better than the other.

Tuning cost

```
df3_train <- tibble(X1 = runif(n = 500, min = 0, max = 10),
                    X2 = runif(n = 500, min = 0, max = 10),
                    e = rnorm(n = 500, mean = 0, sd = 0.5),
                    Y = 2* (X1) + 3* (X2) + e) %>%
  mutate(result = ifelse(Y <= 30, "fail", "success"))
```

```
df3_test <- tibble(X1 = runif(n = 500, min = 0, max = 10),
                   X2 = runif(n = 500, min = 0, max = 10),
                   e = rnorm(n = 500, mean = 0, sd = 0.5),
                   Y = 2* (X1) + 3* (X2) + e) %>%
  mutate(result = ifelse(Y <= 30, "fail", "success"))

df3_train %>%
  ggplot(aes(X1, X2, color = result)) +
  geom_point() +
  labs(title = "Data set with a barely linear decision boundary")
```




```
X3_train <- df3_train %>%
  dplyr::select(X1, X2)
Y3_train <- df3_train$result

X3_test <- df3_test %>%
  dplyr::select(X1, X2)
Y3_test <- df3_test$result
```

```
#prepare tuning grid for costs
tuning_grid <- tibble(C = seq(from = 0.05, to = 1, by = 0.05),
                      train_wrong = NA,
                      test_wrong = NA,
                      train_accuracy = NA,
                      test_accuracy = NA)
```

```
for(i in seq_along(tuning_grid$C)){
  #set up model
  model <- train(
    x = X3_train,
    y = Y3_train,
    method = "svmLinear",
    #C is the parameter for cost in svmLinear
    tuneGrid = expand.grid(C = tuning_grid$C[i]),
    trControl = cv_10
  )
  #save the CV training outcome
  train_out <- model$pred %>%
    count(obs == pred) %>%
    spread("obs == pred", n)
  #enter training error counts
  tuning_grid$train_wrong[i] <- train_out$`FALSE`
  #save the CV testing outcome
  test_out <- tibble(truth = Y3_test,
                     pred = predict(model, newdat = X3_test)) %>%
    count(truth == pred) %>%
    spread("truth == pred", n)
  #enter testing error counts
  tuning_grid$test_wrong[i] <- test_out$`FALSE`
}
```

```
#calculate CV error rates
tuning_grid <- tuning_grid %>%
  mutate(train_accuracy = train_wrong / 500,
         test_accuracy = test_wrong / 500)

tuning_grid
```

```
## # A tibble: 20 x 5
##        C train_wrong test_wrong train_accuracy test_accuracy
##    <dbl>       <int>      <int>          <dbl>         <dbl>
##  1  0.05           7          8          0.014         0.016
##  2  0.1            9          5          0.018         0.01
##  3  0.15          10          5          0.02          0.01
##  4  0.2            8          4          0.016         0.008
##  5  0.25          11          6          0.022         0.012
##  6  0.3           11          4          0.022         0.008
##  7  0.35           9          7          0.018         0.014
##  8  0.4           11          4          0.022         0.008
##  9  0.45          11          8          0.022         0.016
## 10  0.5           11          8          0.022         0.016
## 11  0.55          13          5          0.026         0.01
## 12  0.6            8          6          0.016         0.012
## 13  0.65           9          4          0.018         0.008
## 14  0.7           10          5          0.02          0.01
## 15  0.75           9          5          0.018         0.01
## 16  0.8            8          5          0.016         0.01
```

```
## 17  0.85           8          5          0.016         0.01
## 18  0.9            9          4          0.018         0.008
## 19  0.95          10          5          0.02          0.01
## 20  1              7          5          0.014         0.01
```

- When cost is 0.05, or 1, the SVM model obtains the smallest number of traininig errors (7), which means when cost is 0.05 or 1, the SVM model has the lowest training error rate.
- When cost is 0.2, 0.3, 0.4, 0.65, or 0.9, the SVM model has the smaller number of testing errors (4), which means then it has the lowest testing error rate.
- It appears that the models with the lowest testing error rate are all outperformed in training by a few other models with a different cost. However, it's not necessarily the case that a model with a lower cost can always perform better in testing despite worse training performance than a model with a higher cost.
- As we can see, the 5 different costs with the lowest testing error rate spread across entire the range of cost. Nonetheless, it's true that among these 5the model with a smaller cost tend to perform less well than those with a higher cost in training. Still, at cost = 0.2, the model has a very low error rate in training and a lowest error rate in testing. It may be an exception.
- On the other hand, the range of cost is limited in this example, because it takes longer and longer to train the model as cost value increases. Therefore, when cost goes above 1, we cannot tell whether it's true that a model with a lower cost can perform better in testing despite worse training performance than a model with a higher cost.
- If we consider computation time and testing accuracy, then the models with a lower cost do appear better than those with a higher cost.

Application: Predicting attitudes towards racist college professors

```
#need to sest colrac as factor for classifiction
gss_train <- read_csv("data/gss_train.csv") %>%
  mutate(colrac = as.factor(colrac))
gss_test <- read_csv("data/gss_test.csv") %>%
  mutate(colrac = as.factor(colrac))

#because colrac is still represented in numbers
#there will be an error message in train()
#unless classProbs is set to F,
#or colrac switched to characters
cv101 <- trainControl(method = "cv",
                      number = 10,
                      savePredictions = TRUE,
                      classProbs = FALSE)
```

Support vector classifier

```
#the max of cost is set to 2
#because past 2 there's no change in error rate
tune_svl <- tibble(C = seq(from = 0.1, to = 2, by = 0.1),
                   train_error = NA,
                   test_error = NA)
```

```
for(i in seq_along(tune_svl$C)){
  #set up model
  model <- train(
    colrac ~ .,
    data = gss_train,
    method = "svmLinear",
    #C is the parameter for cost in svmLinear
    tuneGrid = expand.grid(C = tune_svl$C[i]),
    #trControl sets the function to save training outcomes
    trControl = cv101
  )
  #extract the CV training outcome
  train_out <- model$pred %>%
    count(obs == pred) %>%
    spread("obs == pred", n)
  #enter training error rate
  tune_svl$train_error[i] <- train_out$`FALSE` / nrow(gss_train)
  #save the CV testing outcome
  test_out <- tibble(truth = gss_test$colrac,
                     pred = predict(model, newdat = gss_test)) %>%
    count(truth == pred) %>%
    spread("truth == pred", n)
  #enter testing error rate
  tune_svl$test_error[i] <- test_out$`FALSE` / nrow(gss_test)
}
```

```
tune_svl
```

```
## # A tibble: 20 x 3
##        C train_error test_error
##    <dbl>       <dbl>      <dbl>
##  1   0.1       0.204      0.215
##  2   0.2       0.217      0.217
##  3   0.3       0.207      0.215
##  4   0.4       0.202      0.217
##  5   0.5       0.207      0.219
##  6   0.6       0.213      0.217
##  7   0.7       0.203      0.217
##  8   0.8       0.211      0.217
##  9   0.9       0.207      0.217
## 10   1         0.205      0.217
## 11   1.1       0.207      0.219
## 12   1.2       0.210      0.217
## 13   1.3       0.207      0.217
## 14   1.4       0.211      0.219
## 15   1.5       0.209      0.219
## 16   1.6       0.201      0.219
## 17   1.7       0.207      0.219
## 18   1.8       0.202      0.217
## 19   1.9       0.213      0.219
## 20   2         0.205      0.219
```

```
tune_svl[which.min(tune_svl$test_error),]
```

```
## # A tibble: 1 x 3
##       C train_error test_error
##   <dbl>       <dbl>      <dbl>
## 1   0.1       0.204      0.215
```

```
tune_svl[which.min(tune_svl$train_error),]
```

```
## # A tibble: 1 x 3
##       C train_error test_error
##   <dbl>       <dbl>      <dbl>
## 1   1.6       0.201      0.219
```

```
tune_svl[3,]
```

```
## # A tibble: 1 x 3
##       C train_error test_error
##   <dbl>       <dbl>      <dbl>
## 1   0.3       0.207      0.215
```

- As the data show, at cost = 0.1, the model has the lowest test error rate, while at cost = 1.6, the model has the lowest training error rate.
- Here, the results support the claim that an SVM with a lower cost may perform better than an SVM with a higher cost in test performance, despite its worse training performance.
- When cost = 1.6, the test performance is among the worst. This may be due to overfitting.
- When cost = 0.1, the training performance is in fact not bad either, though not the very best. This makes cost = 0.1 the optimal setting for SVM with a linear kernel here.
- When we compare other cost values that also have a very low, if not the lowest test error rate, say cost = 0.3, we will find they have worse training performance. Note that the difference between consecutive cost values is small, which may be a reason why two cost values simultaneously have the lowest test error rate.

Support vector machine with radial kernels

```
tune_rad <- expand_grid(C = seq(from = 0.1, to = 2, by = 0.5),
                        sigma = c(0, 0.1, 0.5, 1),
                        train_error = NA,
                        test_error = NA)
```

```
for(i in seq_along(tune_rad$C)){
  #set up model
  model <- train(
    colrac ~ .,
    data = gss_train,
    method = "svmRadial",
    #C is the parameter for cost in svmLinear
    tuneGrid = expand.grid(C = tune_rad$C[i],
                           sigma = tune_rad$sigma[i]),
```

```
    #trControl sets the function to save training outcomes
    trControl = cv101
  )
  #extract the CV training outcome
  train_out <- model$pred %>%
    count(obs == pred) %>%
    spread("obs == pred", n)
  #enter training error rate
  tune_rad$train_error[i] <- train_out$`FALSE` / nrow(gss_train)
  #save the CV testing outcome
  test_out <- tibble(truth = gss_test$colrac,
                     pred = predict(model, newdat = gss_test)) %>%
    count(truth == pred) %>%
    spread("truth == pred", n)
  #enter testing error rate
  tune_rad$test_error[i] <- test_out$`FALSE` / nrow(gss_test)
}
```

```
tune_rad
```

```
## # A tibble: 16 x 4
##        C sigma train_error test_error
##    <dbl> <dbl>       <dbl>      <dbl>
##  1   0.1   0         0.475      0.462
##  2   0.1   0.1       0.475      0.462
##  3   0.1   0.5       0.475      0.462
##  4   0.1   1         0.475      0.462
##  5   0.6   0         0.475      0.462
##  6   0.6   0.1       0.434      0.440
##  7   0.6   0.5       0.475      0.462
##  8   0.6   1         0.475      0.462
##  9   1.1   0         0.475      0.462
## 10   1.1   0.1       0.292      0.325
## 11   1.1   0.5       0.475      0.462
## 12   1.1   1         0.475      0.462
## 13   1.6   0         0.475      0.462
## 14   1.6   0.1       0.296      0.318
## 15   1.6   0.5       0.475      0.462
## 16   1.6   1         0.475      0.462
```

```
tune_rad[which.min(tune_rad$train_error),]
```

```
## # A tibble: 1 x 4
##       C sigma train_error test_error
##   <dbl> <dbl>       <dbl>      <dbl>
## 1   1.1   0.1       0.292      0.325
```

```
tune_rad[which.min(tune_rad$test_error),]
```

```
## # A tibble: 1 x 4
##       C sigma train_error test_error
##   <dbl> <dbl>       <dbl>      <dbl>
## 1   1.6   0.1       0.296      0.318
```

- The optimal parameters for SVM with a radial kernel is cost = 1.6, and sigma = 0.1.
- For some reason, many models have the same training error rate and test error rate. Nonetheless, the model with the lowest training error rate actually has the second lowest test error rate, while the model with the lowest test error rate has the second lowest trainign error rate. Both of the two models have sigma = 0.1.
- Moreover, the difference between the lowest and second lowest test error rate is less than 1%, which is probably statistically insignificant. Which one is better than the other depends more on subjective judgment than hard facts. The optimal parameters chosen here may take more time to compute than the second best model. It'd better be worth it because it takes a long time for the grid search.

Support vector machine with polynomial kernel

```
tune_poly <- expand_grid(C = seq(from = 0.1, to = 1, by = 0.2),
                         scale = seq(from = 0.1, to = 1, by = 0.2),
                         degree = c(1, 2, 3),
                         train_error = NA,
                         test_error = NA)
```

```
for(i in seq_along(tune_poly$C)){
  #set up model
  model <- train(
    colrac ~ .,
    data = gss_train,
    method = "svmPoly",
    #C is the parameter for cost in sumLinear
    tuneGrid = expand.grid(C = tune_poly$C[i],
                           degree = tune_poly$degree[i],
                           scale = tune_poly$scale[i]),
    #trControl sets the function to save training outcomes
    trControl = cv101
  )
  #extract the CV training outcome
  train_out <- model$pred %>%
    count(obs == pred) %>%
    spread("obs == pred", n)
  #enter training error rate
  tune_poly$train_error[i] <- train_out$`FALSE` / nrow(gss_train)
  #save the CV testing outcome
  test_out <- tibble(truth = gss_test$colrac,
                     pred = predict(model, newdat = gss_test)) %>%
    count(truth == pred) %>%
    spread("truth == pred", n)
  #enter testing error rate
  tune_poly$test_error[i] <- test_out$`FALSE` / nrow(gss_test)
}
```

```
tune_poly
```

```
## # A tibble: 75 x 5
##        C scale degree train_error test_error
##    <dbl> <dbl>  <dbl>       <dbl>      <dbl>
```

```
##  1   0.1   0.1      1       0.203      0.213
##  2   0.1   0.1      2       0.213      0.237
##  3   0.1   0.1      3       0.226      0.264
##  4   0.1   0.3      1       0.201      0.215
##  5   0.1   0.3      2       0.268      0.284
##  6   0.1   0.3      3       0.220      0.270
##  7   0.1   0.5      1       0.209      0.215
##  8   0.1   0.5      2       0.267      0.298
##  9   0.1   0.5      3       0.225      0.268
## 10   0.1   0.7      1       0.205      0.217
## # ... with 65 more rows
```

```
tune_poly[which.min(tune_poly$train_error),]
```

```
## # A tibble: 1 x 5
##       C scale degree train_error test_error
##   <dbl> <dbl>  <dbl>       <dbl>      <dbl>
## 1   0.1   0.9      1       0.198      0.215
```

```
tune_poly[which.min(tune_poly$test_error),]
```

```
## # A tibble: 1 x 5
##       C scale degree train_error test_error
##   <dbl> <dbl>  <dbl>       <dbl>      <dbl>
## 1   0.1   0.1      1       0.203      0.213
```

- As it appears, the model with the lowest training error rate and the model with the loest test error rate have the same values for cost and degree. In fact, if we have a closer look at the outcome, we will find there are more than one optimal model in terms of test error rate, including the one with the lowest training error rate.
- In this case, the optimal parameters for SVM with a polynomial kernel should be cost = 0.1, scale = 0.1, degree = 1. It is the optimal model not only because it has the lowest test error rate, but also because its training performance is very good. That indicates this model has a relatively lower risk of overfitting than the others while keeps a high accuracy in prediction.

Across fits

- Comparing all three kinds of kernels, it appears that SVM with a polynoimal kernel performs the best in testing.
- Nonetheless, the difference between linear kernel and polynomial kernel in test error rate is not big at all: 0.0020284. It's questionable whether this difference is statistically significant.
- On the other hand, SVM with a radial kernel is indeed outperformed by the other two. Its lowest test errror is almost 10% higher than that of the other two. That implies SVM with a radial kernel does not fit the data here very well. It's likely that the relation between `colrac` and the other variables in the data set is linear, and perhaps involves polynomial terms.
- Comparing the three models' training error rate can also induce the same conclusion: the training error is lowest for polynomial kernel, then linear kernel, and the last radial kernel. The difference between the former two is very small, while the training error rate for the radial kernel is much larger.
- Interestingly,linear kernel and polynomial kernel have the same value for cost (0.1), while the cost for the optimal radial kernel is much higher (1.6). Considering how a larger cost can increase both the computation time and the risk of overfitting, radial kernel looks even less desirable in this setting.

- We can't say which kernel is better than the others in general. The fit will always depend on the data in question. Here the best fit is polynomial kernel. But if the decision boundary is non-linear, then a radial kernel is likely to perform better than the other two.