CMSC/LING/STAT 208: Machine Learning

Abhishek Chakraborty [Much of the content in these slides have been adapted from *ISLR2* by James et al. and *HOMLR* by Boehmke & Greenwell]

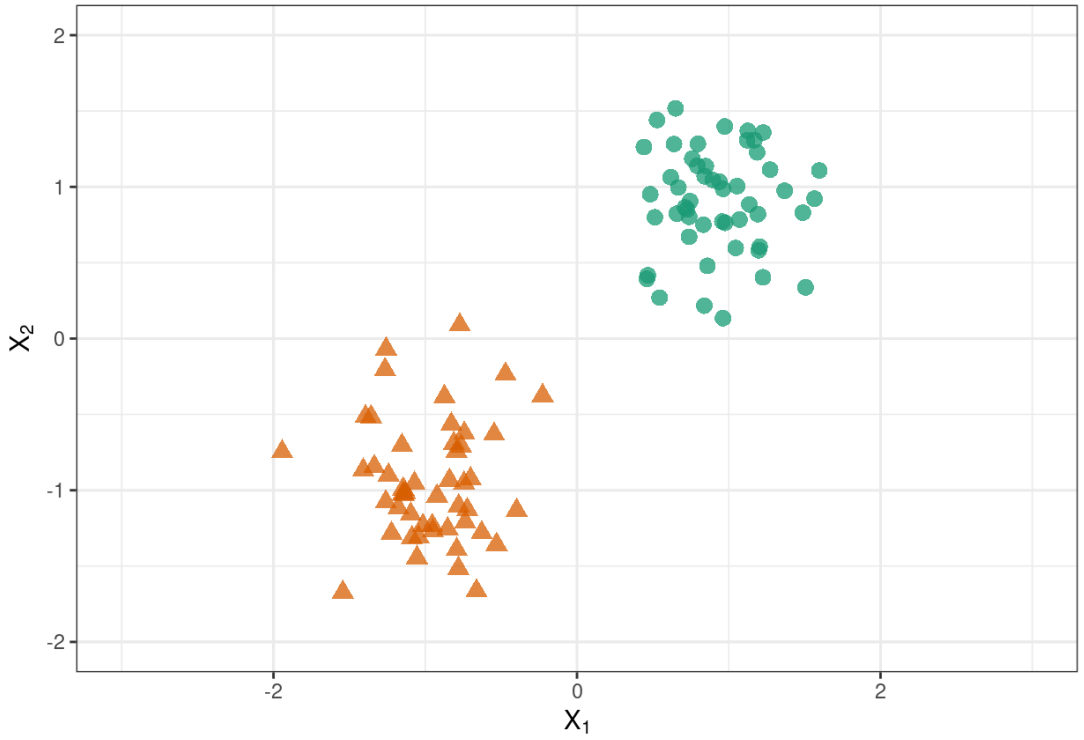
Support Vector Machines (SVM)

- One of the best “out of the box” classifiers.
- Mostly intended for two-class classification problems.

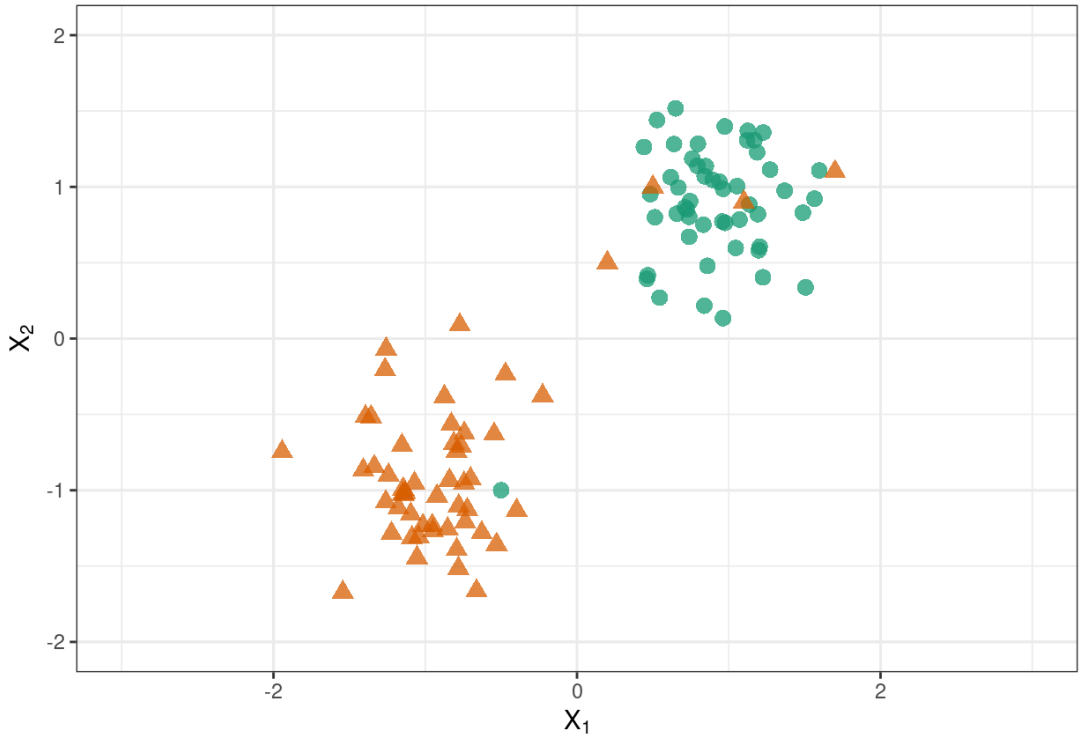
Idea: **Try and find a plane that separates the classes in some feature space**.

- We will talk about
  - Maximal Margin Classifier
  - Support Vector Classifier
  - Support Vector Machine

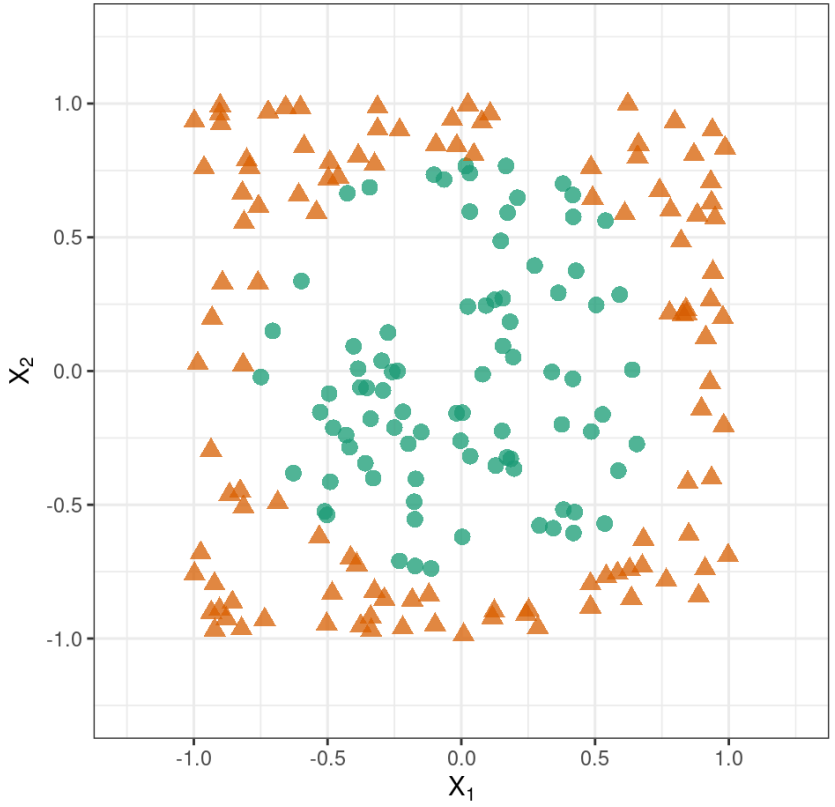
SVM: Motivation 1



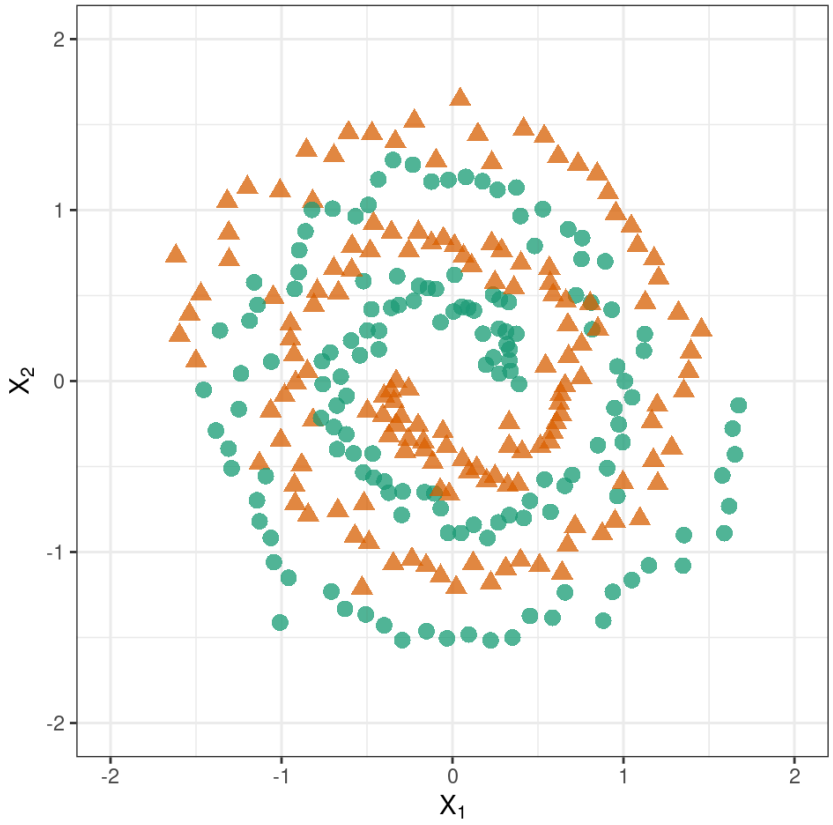
SVM: Motivation 2



SVM: Motivation 3



SVM: Motivation 4



Hyperplane

- In p-dimensions, a **hyperplane** is a flat affine subspace of dimension $p - 1$.
- Mathematical form of a hyperplane

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0$$

- When $p = 2$, a hyperplane is a line.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$$

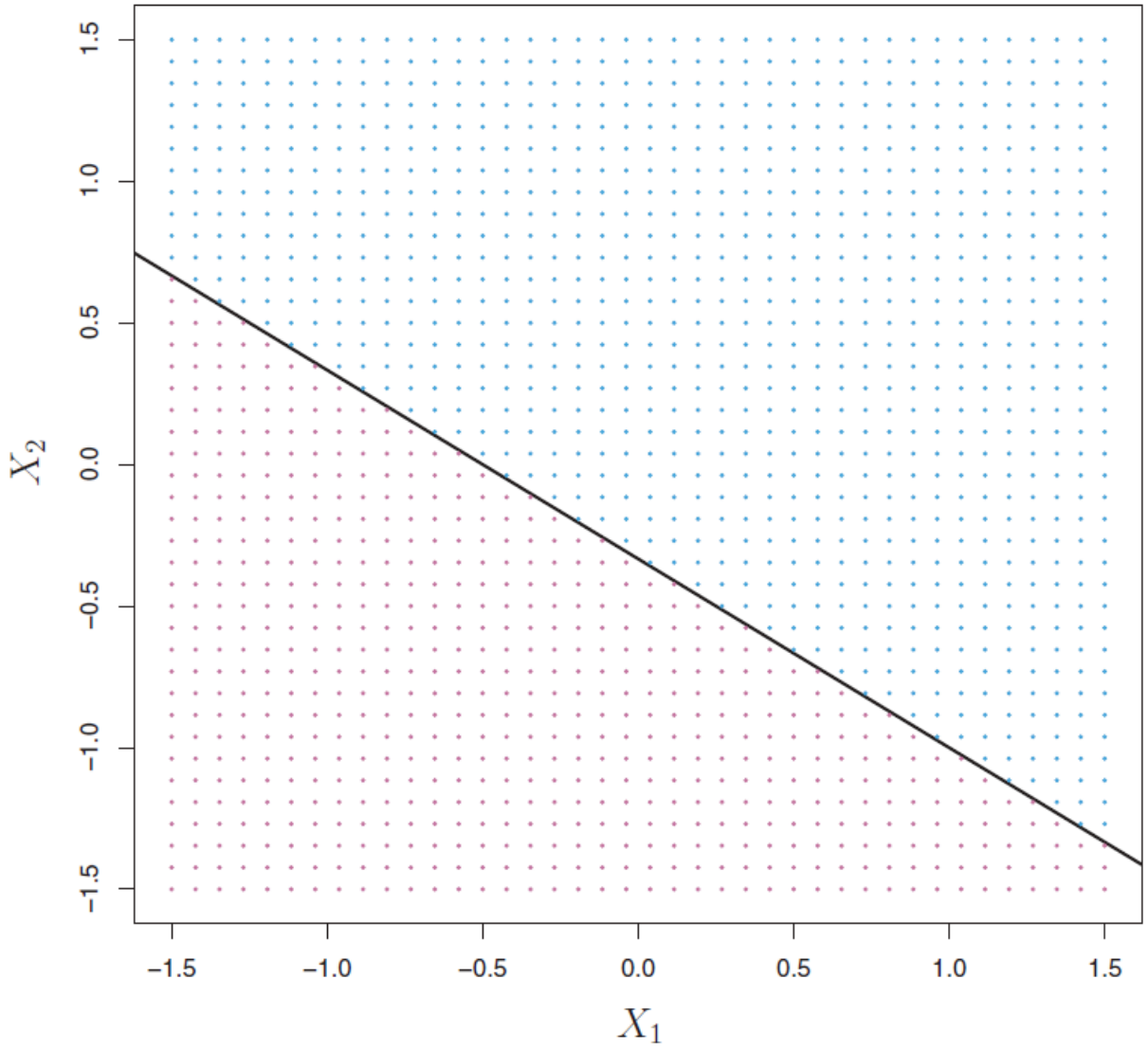
- If $\beta_0 = 0$, the hyperplane goes through the origin, otherwise not.
- A hyperplane divides the p-dim space into 2 halves.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p > 0$$

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p < 0$$

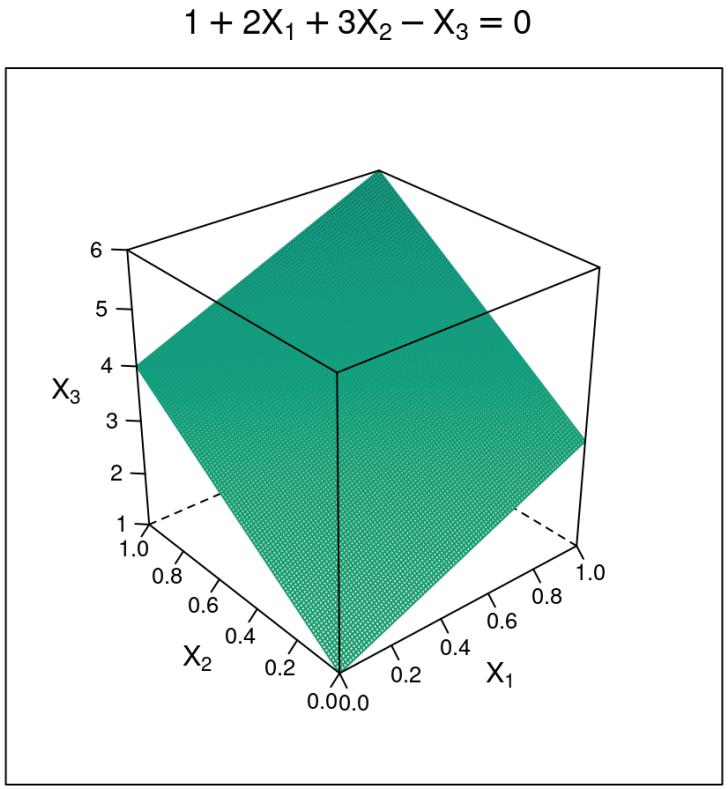
Hyperplane

Hyperplane in 2-dimensions: $1 + 2X_1 + 3X_2 = 0$



Hyperplane

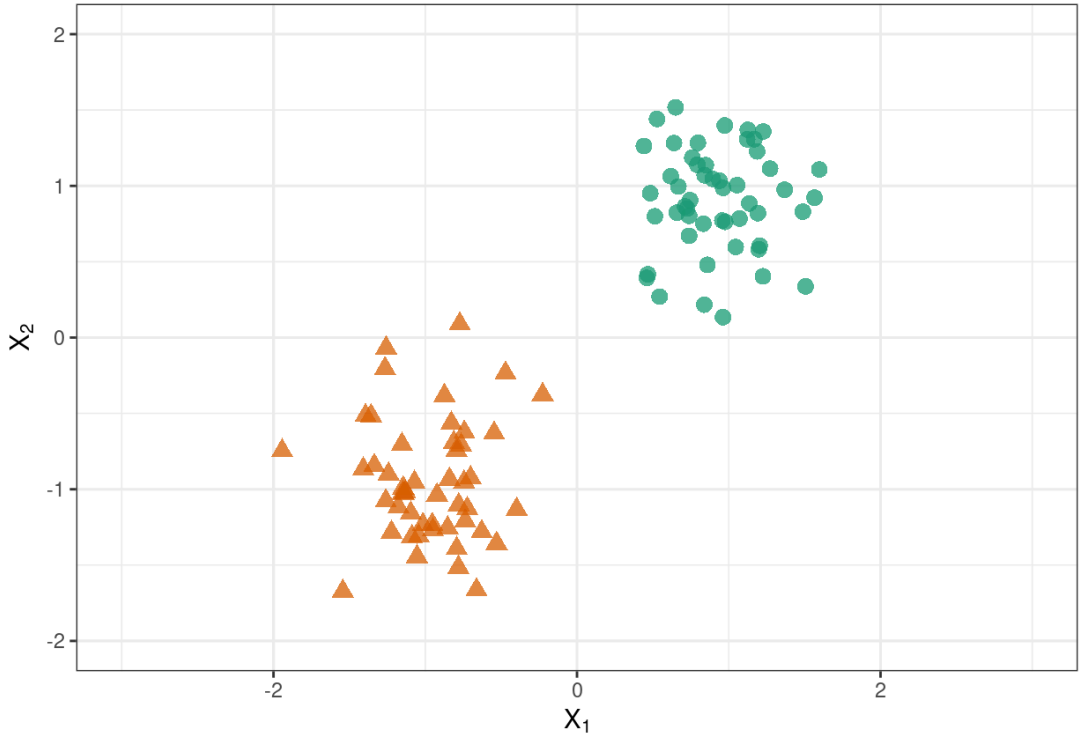
Hyperplane in 3-dimensions: $1 + 2X_1 + 3X_2 - X_3 = 0$



Separating Hyperplane

For a two-class problem, suppose that it is possible to construct a hyperplane that separates the training observations perfectly according to their class labels.

Such a hyperplane is known as a **separating hyperplane**.



Separating Hyperplane

For a p-dimensional two-class problem,

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} > 0 \text{ if } y_i = 1,$$

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} < 0 \text{ if } y_i = -1.$$

Equivalently, for $i = 1, 2, \ldots, n$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) > 0$$

Consider a test observation x^*, we compute
$f(x^*) = \beta_0 + \beta_1\ x_1^* + \beta_2\ x_2^* + \ldots + \beta_p\ x_p^*$

If $f(x^*)$ is positive, assign class label 1, otherwise, assign class label -1.

A classifier based on a separating hyperplane leads to a linear decision boundary.

Optimal Separating Hyperplane (OSH)

This is also known as the **maximal margin classifier (MMC)** or **hard margin classifier (HMC)**.

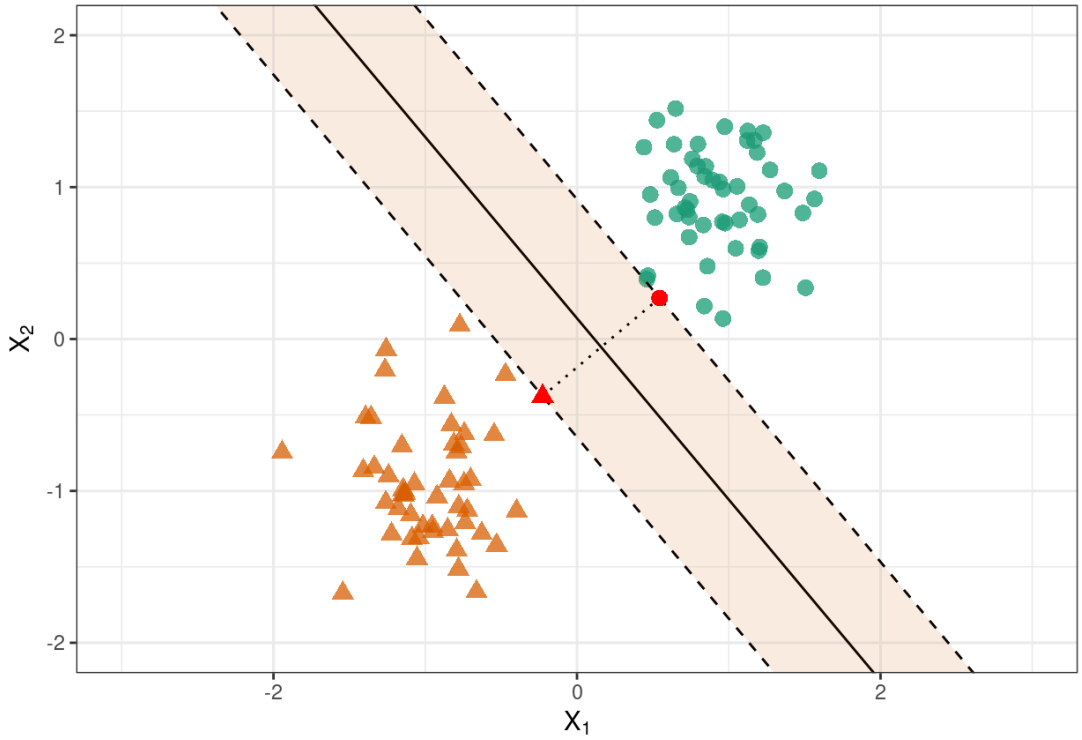
- One that makes the biggest gap or **margin** between the two classes.
- One that is farthest from the training observations.

Margin: The minimal (perpendicular) distance from the observations to the hyperplane. Denoted by M.

The maximal margin hyperplane is the separating hyperplane for which the margin is largest, that is, the hyperplane that has the farthest minimum distance to the training observations.

Optimal Separating Hyperplane

```
##  Setting default kernel parameters
```



Optimal Separating Hyperplane

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{maximize}}\, M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M \;\; \forall\, i = 1, \ldots, n$$

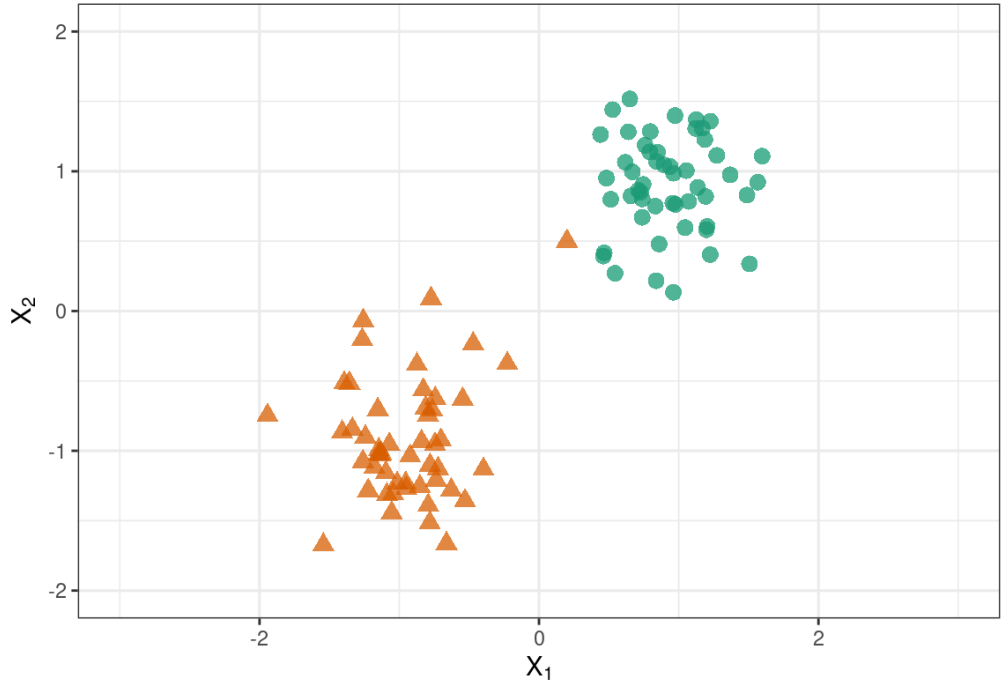
The second constraint guarantees that each observation will be on the correct side of the hyperplane (M positive).

The first constraint ensures that the perpendicular distance from i^{th} observation to the hyperplane is

$$y_i \left(\beta_0 + \beta_1\, x_{i1} + \beta_2\, x_{i2} + \ldots + \beta_p\, x_{ip}\right)$$

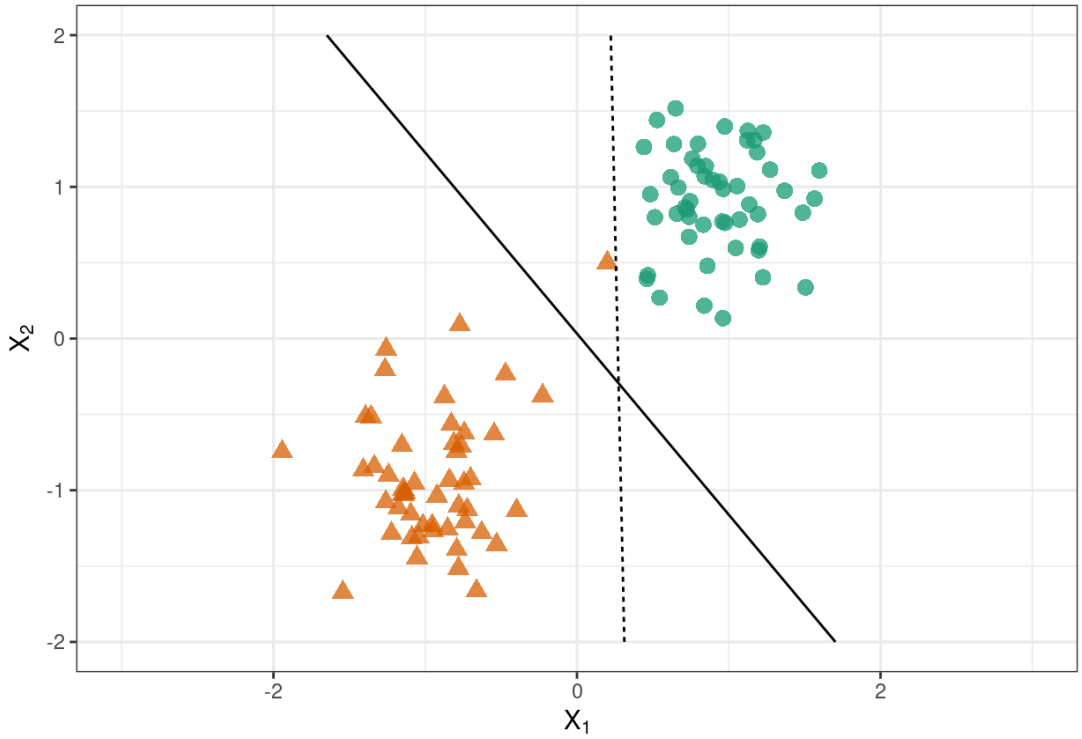
Optimal Separating Hyperplane: Issue 1

The optimal separating hyperplane fits the data too hard.



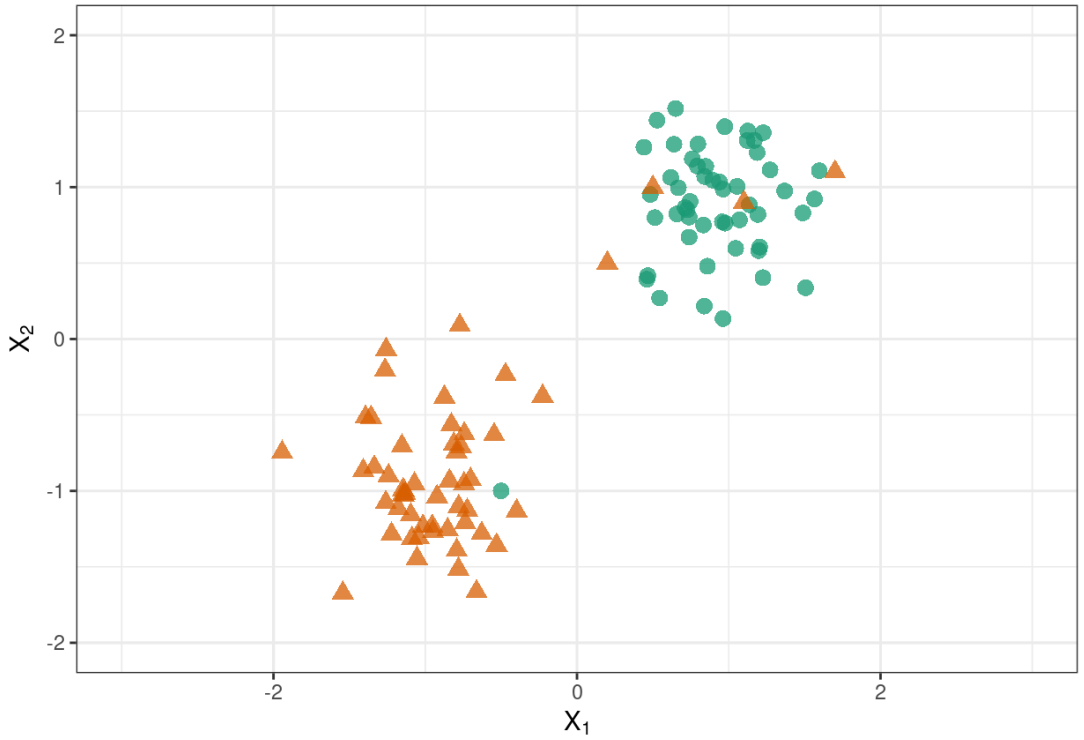
Optimal Separating Hyperplane: Issue 1

The optimal separating hyperplane fits the data too hard.



Optimal Separating Hyperplane: Issue 2

An optimal separating hyperplane may not always be possible to construct, that is, non-separable data. This is often the case, unless $n < p$.



Support Vector Classifier (SVC)

We might be willing to misclassify a few observations for

- greater robustness to individual observations, and
- better classify **most** of the observations.

This leads us to the **support vector classifier**. Also called the **soft margin classifier**.

The margin is **soft** because it can be violated by some of the training observations.

Support Vector Classifier

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n}{\text{maximize}} \quad M$$

$$\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i)$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C,$$

M: width of the margin

$\epsilon_1, \ldots, \epsilon_n$: **Slack variables**

C: **Budget** (tuning parameter)

Support Vector Classifier

- $\epsilon_i = 0$:
- $\epsilon_i > 0$:
- $\epsilon_i > 1$:
- Support vectors:
- $C = 0$:
- $C > 0$:

Support Vector Classifier

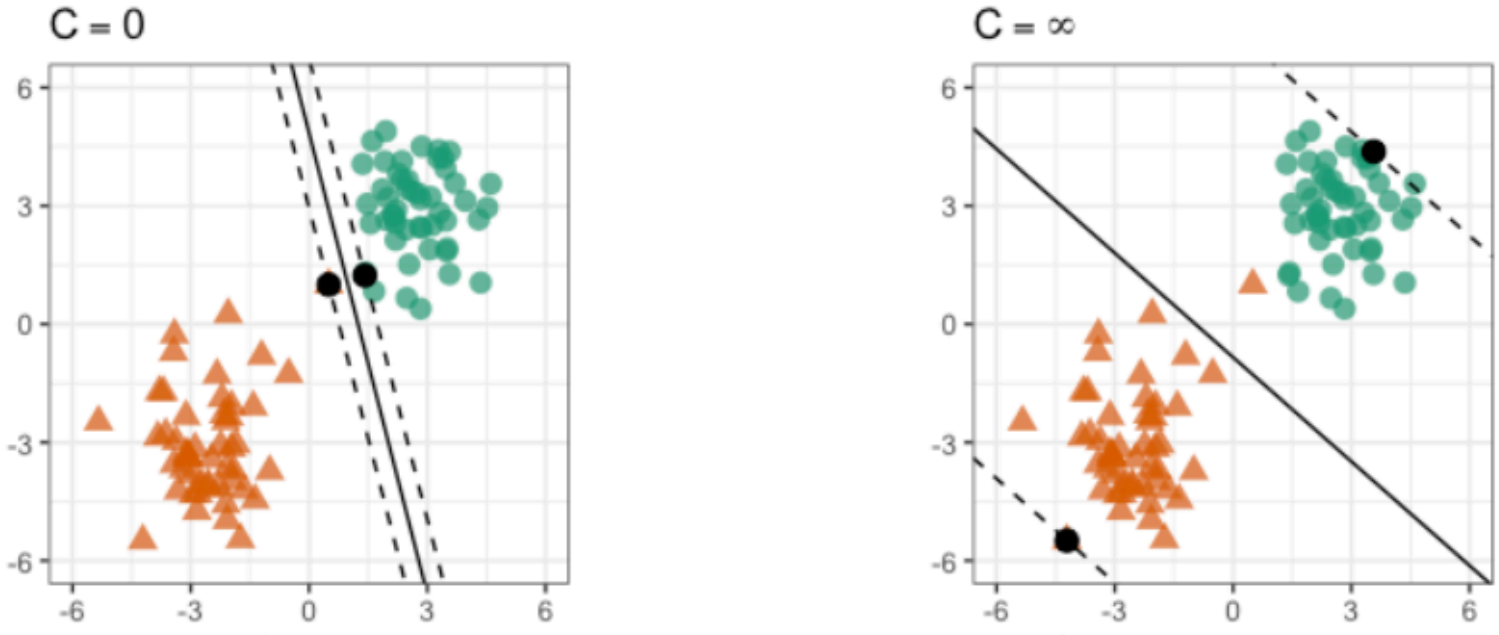


Figure adapted from *Hands-On Machine Learning with R*, Boehmke and Greenwell

Support Vector Classifier

```
set.seed(052323)   # set seed

# implement CV to find optimal C
svc_cv <- train(y ~ .,
                data = svcdata,
                method = "svmLinear",
                trControl = trainControl(method = "repeatedcv", number = 10, repeats = 5),
                tuneLength = 20,
                metric = "Accuracy")

svc_cv$bestTune   # optimal C
```

```
##   C
## 1 1
```

```
# fit model with optimal C
final_model_svc <- ksvm(y ~ .,
                        data = svcdata,
                        kernel = "vanilladot",
                        C = svc_cv$bestTune$C,
                        prob.model = TRUE)       # needed to obtain predicted probabilities
```

```
##  Setting default kernel parameters
```

Support Vector Classifier

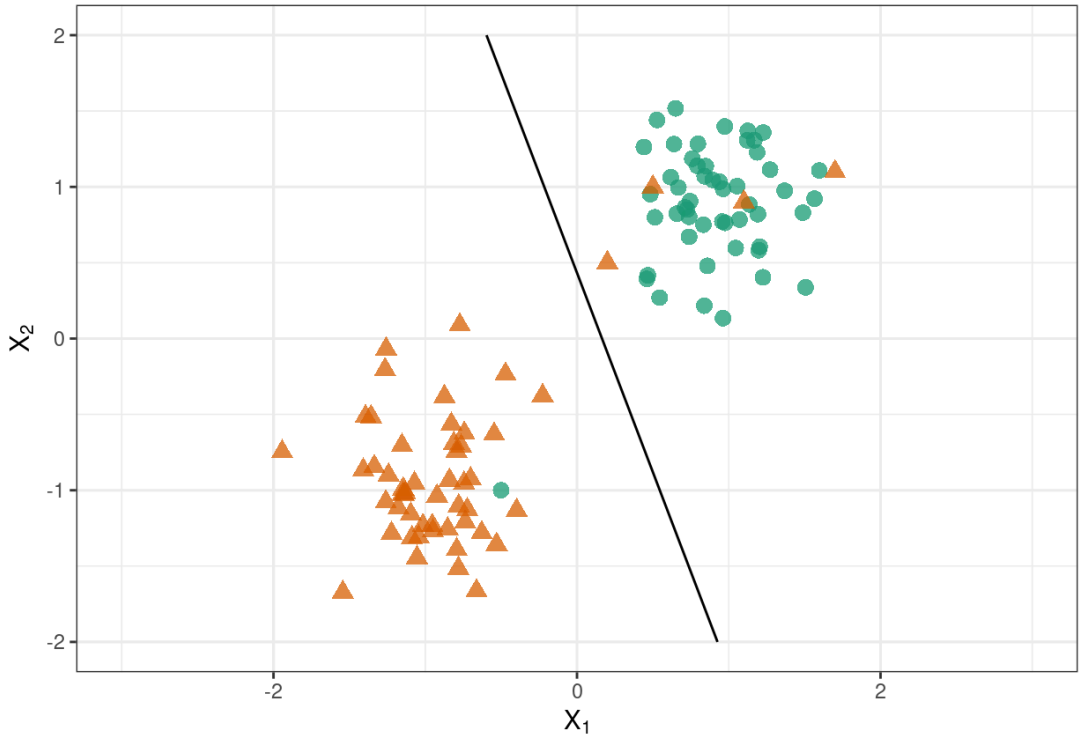
```
final_model_svc   # number of support vectors
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 16
##
## Objective Function Value : -14.2888
## Training error : 0.047619
## Probability model included.
```

```
alphaindex(final_model_svc)   # which observations are support vectors
```

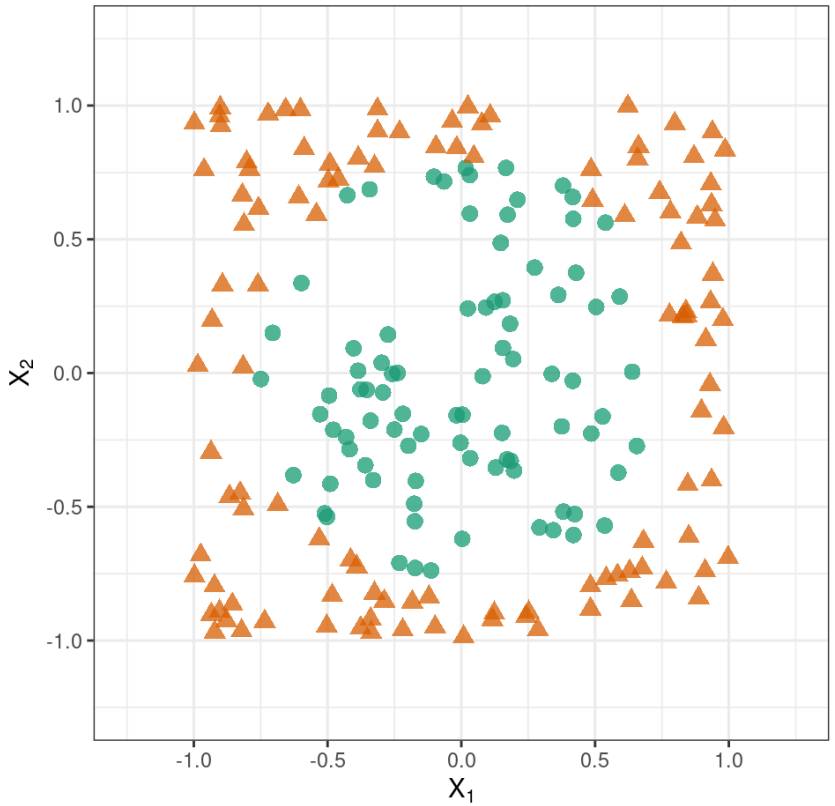
```
## [[1]]
##  [1]  16  17  21  22  31  49  55  56  88  96  97 101 102 103 104 105
```

Support Vector Classifier



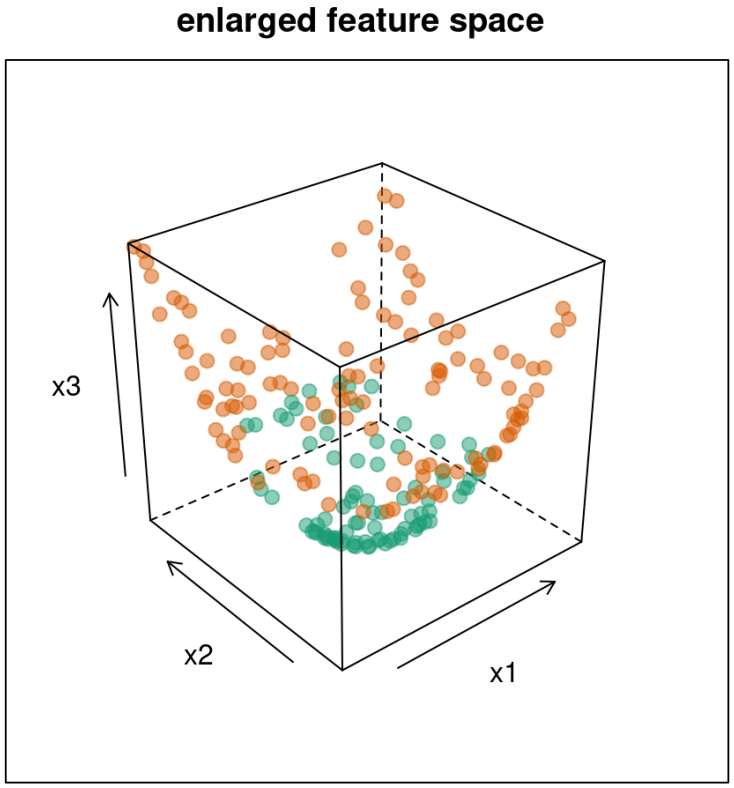
Non-linear Boundaries

Why support vector classifiers are not enough?



Feature Expansion

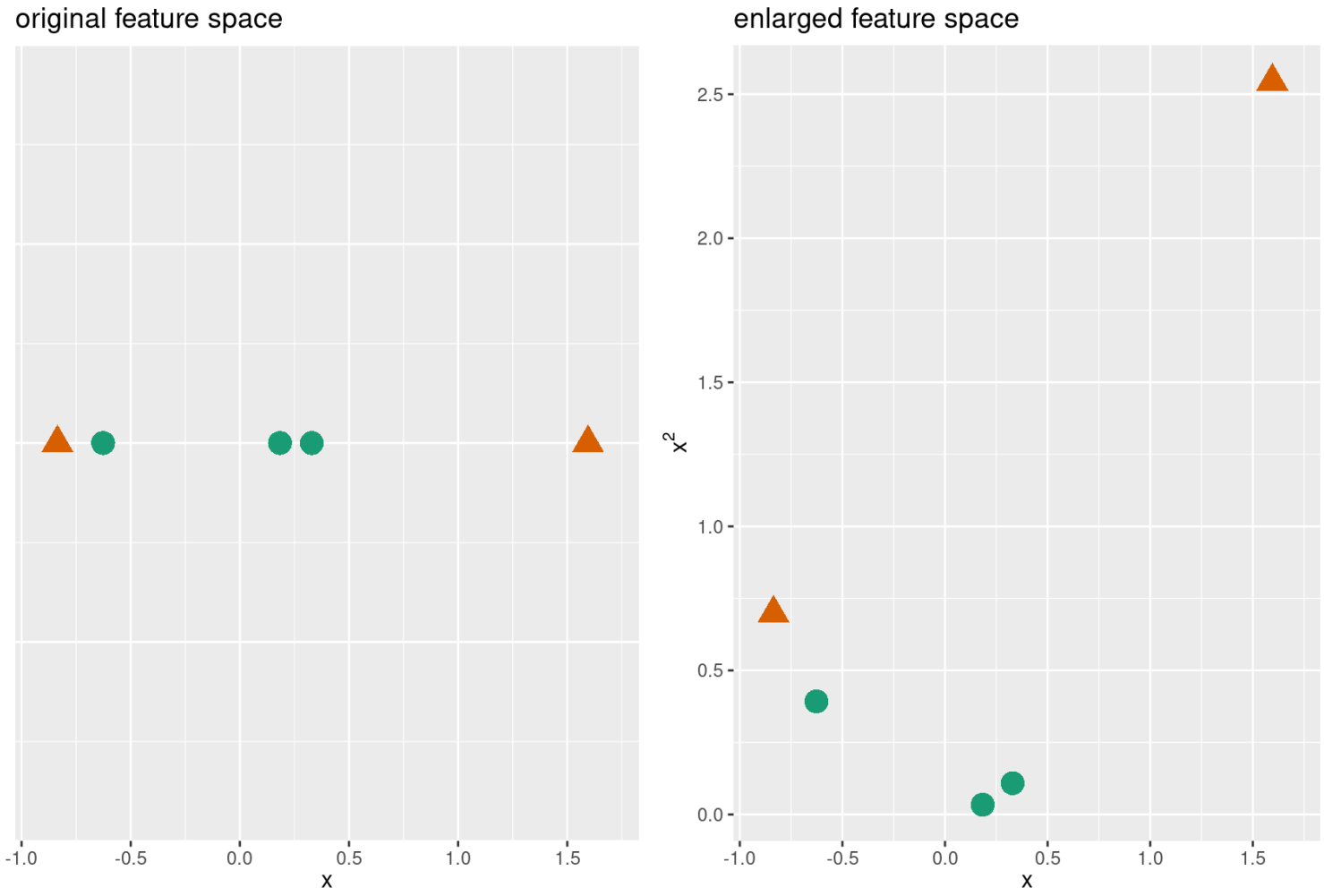
Feature space has been enlarged by adding a third feature, $X_3 = X_1^2 + X_2^2$



Feature Expansion

- The problem of non-linear boundaries can be solved by enlarging the feature space (like in linear regression) using transformations of predictors.
- Fit a support vector classifier in the enlarged space.
- Results in non-linear boundaries in the original space.

Feature Expansion



Feature Expansion

A **kernel function** quantifies the similarity between two observations. It helps in transforming the original feature space to an enlarged feature space where the data points can be separated by a linear boundary.

Commonly used **kernel functions** are

- **Polynomial Kernel of degree d**

$$k(x_i, x_{i'}) = \left(1 + scale \sum_{j=1}^{p} x_{ij}\ x_{i'j}\right)^{degree}$$

- **Radial Basis Function Kernel**

$$k(x_i, x_{i'}) = \exp\left(-\sigma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right)$$

A support vector classifier with a non-linear kernel is known as a **support vector machine**.

Non-linear Boundaries: Circle dataset

We train an SVM with the polynomial kernel.

```
set.seed(052323)   # set seed

# implement CV to find optimal parameters
param_grid_poly <- expand.grid(degree = c(1, 2, 3, 4),
                          scale = c(0.5, 1, 2),
                          C = c(0.001, 0.1, 1, 10, 10))

svm_poly_cv <- train(y ~ .,
                     data = circle,
                     method = "svmPoly",
                     trControl = trainControl(method = "repeatedcv", number = 10, repeats = 5),
                     tuneGrid = param_grid_poly,
                     metric = "Accuracy")


svm_poly_cv$bestTune
```

```
##    degree scale  C
## 28      3   0.5 10
```

```
max(svm_poly_cv$results$Accuracy)
```

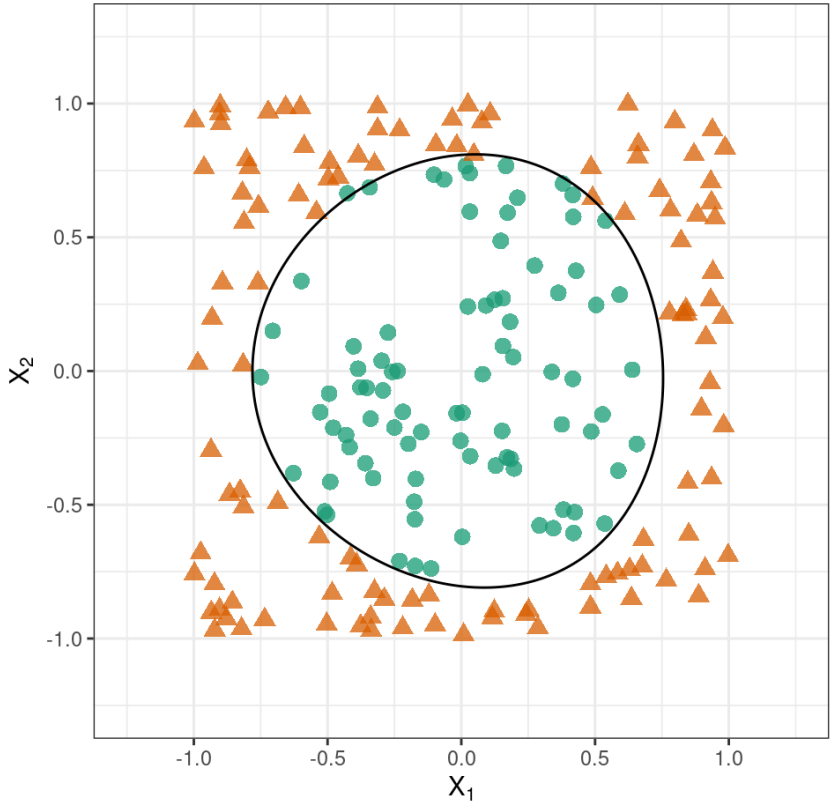
```
## [1] 0.9812757
```

Non-linear Boundaries: Circle dataset

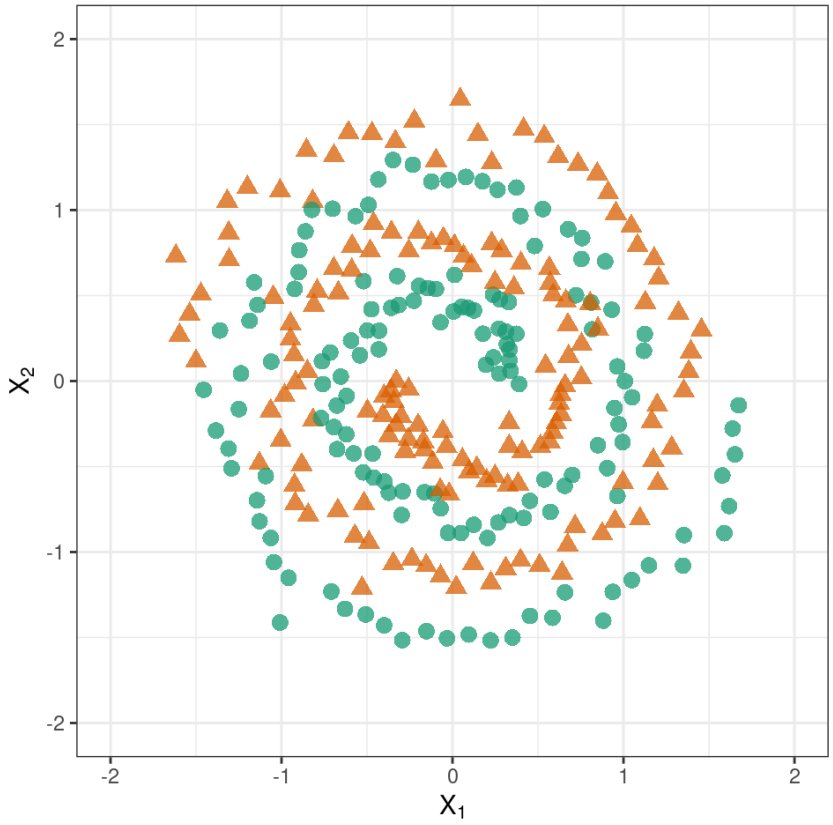
```
# fit model with optimal parameters
final_model_svm_poly <- ksvm(y ~ .,
                             data = circle,
                             kernel = "polydot",
                             kpar = list(degree = svm_poly_cv$bestTune$degree,
                                         scale = svm_poly_cv$bestTune$scale,
                                         offset = 1),
                             C = svm_poly_cv$bestTune$C,
                             prob.model = TRUE)

final_model_svm_poly
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 10
##
## Polynomial kernel function.
##  Hyperparameters : degree =  3  scale =  0.5  offset =  1
##
## Number of Support Vectors : 20
##
## Objective Function Value : -101.3903
## Training error : 0.015
## Probability model included.
```

Non-linear Boundaries: Circle dataset



Non-linear Boundaries: Spirals dataset



Non-linear Boundaries: Spirals dataset

We train an SVM with the radial basis function kernel.

```
set.seed(052323)   # set seed

# implement CV to find optimal parameters
param_grid_radial <- expand.grid(sigma = c(0.5, 1, 1.5, 2),
                                 C = c(0.001, 0.01, 1, 5, 10, 100))

svm_radial_cv <- train(y ~ .,
                       data = spirals,
                       method = "svmRadial",
                       tuneGrid = param_grid_radial,
                       trControl = trainControl(method = "repeatedcv", number = 10, repeats = 5),
                       metric = "Accuracy")

svm_radial_cv$bestTune
```

```
##    sigma   C
## 18   1.5 100
```

```
max(svm_radial_cv$results$Accuracy)
```

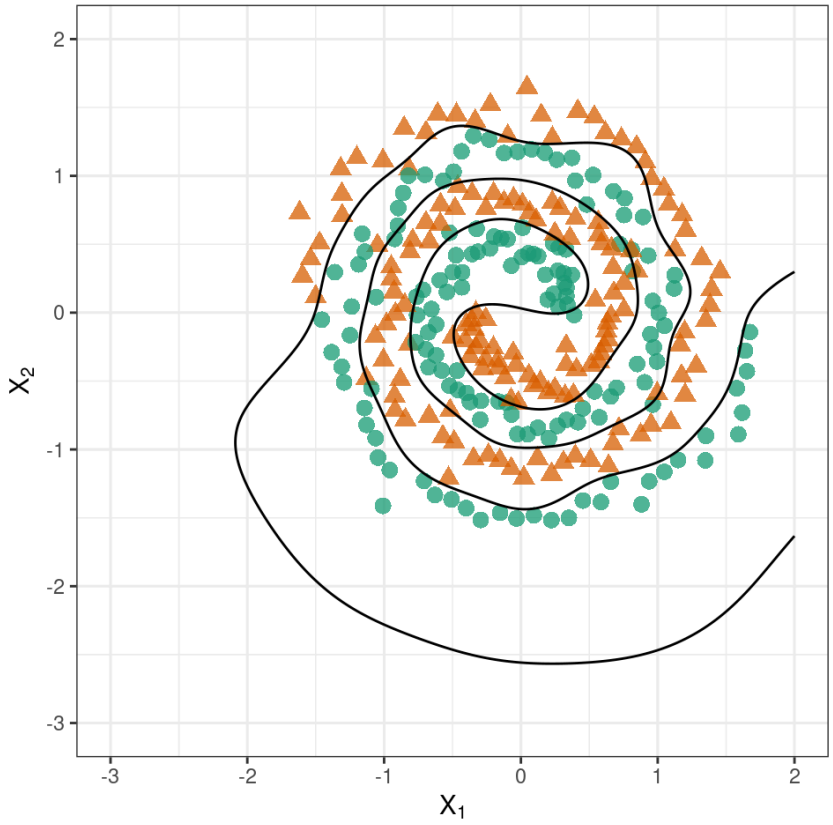
```
## [1] 0.89
```

Non-linear Boundaries: Spirals dataset

```
# fit model with optimal parameters
final_model_svm_radial <- ksvm(y ~ .,
                               data = spirals,
                               kernel = "rbfdot",
                               kpar = list(sigma = svm_radial_cv$bestTune$sigma),
                               C = svm_radial_cv$bestTune$C,
                               prob.model = TRUE)

final_model_svm_radial
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 100
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  1.5
##
## Number of Support Vectors : 109
##
## Objective Function Value : -5906.027
## Training error : 0.05
## Probability model included.
```

Non-linear Boundaries: Spirals dataset



Summary

- SVMs are black-box algorithms. Lack interpretability.
- One of the best methods for two-class classification problems.
- If we wish to estimate probabilities, logistic regression is the way to go.
- For non-linear boundaries, SVMs are popular.

Summary of Supervised Learning Methods

Method	R and/or C	Tuning Params	Feature Eng	Interpretable	Computation
Linear Reg					
Logistic Reg					
KNN					
LASSO					
Single tree					
Bagged trees					
Random forest					
Boosted trees					
SVM					

Practice - Regression

You will work with the **`boston.rds`** dataset.

The task is to predict **`medv`** (median value of owner-occupied homes in $1000s) using the rest of the variables as predictors.

- Use a 70-30 split.
- Use 5-fold CV with 1 repeat.
- Parameter grids are in Rmd file.

Practice - Regression

```
boston <- readRDS("boston.rds")   # load dataset

# response Y - 'medv'
```

```
glimpse(boston)

# all features are numerical except 'chas' which is nominal categorical (output not displayed here)
```

Practice - Regression

```
summary(boston)  # missing entries in 'nox', 'dis'
```

```
##       crim                zn              indus        chas         nox        
##  Min.   : 0.00632   Min.   :  0.00   Min.   : 0.46   0:471   Min.   :0.3850  
##  1st Qu.: 0.08205   1st Qu.:  0.00   1st Qu.: 5.19   1: 35   1st Qu.:0.4530  
##  Median : 0.25651   Median :  0.00   Median : 9.69           Median :0.5380  
##  Mean   : 3.61352   Mean   : 11.36   Mean   :11.14           Mean   :0.5558  
##  3rd Qu.: 3.67708   3rd Qu.: 12.50   3rd Qu.:18.10           3rd Qu.:0.6240  
##  Max.   :88.97620   Max.   :100.00   Max.   :27.74           Max.   :0.8710  
##                                                              NA's   :73      
##        rm              age              dis              rad        
##  Min.   :3.561   Min.   :  2.90   Min.   : 1.130   Min.   : 1.000  
##  1st Qu.:5.886   1st Qu.: 45.02   1st Qu.: 2.100   1st Qu.: 4.000  
##  Median :6.208   Median : 77.50   Median : 3.191   Median : 5.000  
##  Mean   :6.285   Mean   : 68.57   Mean   : 3.788   Mean   : 9.549  
##  3rd Qu.:6.623   3rd Qu.: 94.08   3rd Qu.: 5.188   3rd Qu.:24.000  
##  Max.   :8.780   Max.   :100.00   Max.   :12.127   Max.   :24.000  
##                                   NA's   :20                       
##       tax           ptratio          lstat            medv      
##  Min.   :187.0   Min.   :12.60   Min.   : 1.73   Min.   : 5.00  
##  1st Qu.:279.0   1st Qu.:17.40   1st Qu.: 6.95   1st Qu.:17.02  
##  Median :330.0   Median :19.05   Median :11.36   Median :21.20  
##  Mean   :408.2   Mean   :18.46   Mean   :12.65   Mean   :22.53  
##  3rd Qu.:666.0   3rd Qu.:20.20   3rd Qu.:16.95   3rd Qu.:25.00  
##  Max.   :711.0   Max.   :22.00   Max.   :37.97   Max.   :50.00  
##
```

Practice - Regression

```
nearZeroVar(boston, saveMetrics = TRUE)   # no zv/nzv features
```

```
##         freqRatio percentUnique zeroVar   nzv
## crim     1.000000    99.6047431   FALSE FALSE
## zn      17.714286     5.1383399   FALSE FALSE
## indus    4.400000    15.0197628   FALSE FALSE
## chas    13.457143     0.3952569   FALSE FALSE
## nox      1.235294    15.8102767   FALSE FALSE
## rm       1.000000    88.1422925   FALSE FALSE
## age     10.750000    70.3557312   FALSE FALSE
## dis      1.250000    78.6561265   FALSE FALSE
## rad      1.147826     1.7786561   FALSE FALSE
## tax      3.300000    13.0434783   FALSE FALSE
## ptratio  4.117647     9.0909091   FALSE FALSE
## lstat    1.000000    89.9209486   FALSE FALSE
## medv     2.000000    45.2569170   FALSE FALSE
```

Practice - Regression

```
set.seed(208)

# split the data into training and test sets
index <- createDataPartition(boston$medv, p = 0.7, list = FALSE)

boston_train <- boston[index, ]

boston_test <- boston[-index, ]
```

```
set.seed(208)   # set seed

# create recipe and blueprint, prepare and apply blueprint

blueprint <- recipe(medv ~ ., data = boston_train) %>%
  step_impute_mean(nox, dis) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  step_dummy(chas)

prepare <- prep(blueprint, training = boston_train)

baked_train <- bake(prepare, new_data = boston_train)

baked_test <- bake(prepare, new_data = boston_test)
```

Practice - Regression

```
set.seed(208)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 1)   # CV specifications
```

```
set.seed(208)

# linear regression
lm_cv <- train(blueprint,
               data = boston_train,
               method = "lm",
               trControl = cv_specs,
               metric = "RMSE")
```

```
set.seed(208)

lambda_grid <- 10^seq(-3, 3, length = 100)   # grid of lambda values to search over

lasso_cv <- train(blueprint,
                  data = boston_train,
                  method = "glmnet",   # for lasso
                  trControl = cv_specs,
                  tuneGrid = expand.grid(alpha = 1, lambda = lambda_grid),  # alpha = 1 implements lasso
                  metric = "RMSE")
```

Practice - Regression

```
set.seed(208)

# KNN
k_grid <- expand.grid(k = seq(1, 10, by = 1))

knn_cv <- train(blueprint,
                data = boston_train,
                method = "knn",
                trControl = cv_specs,
                tuneGrid = k_grid,
                metric = "RMSE")
```

```
set.seed(208)

# single regression tree
tree_cv <- train(blueprint,
                 data = boston_train,
                 method = "rpart",
                 trControl = cv_specs,
                 tuneLength = 20,
                 metric = "RMSE")
```

```
set.seed(208)

# bagging
bag_fit <- bagging(medv ~ .,
                   data = baked_train,
                   nbagg = 500,
                   coob = TRUE,
                   control = rpart.control(minsplit = 2, cp = 0, xval = 0))
```

Practice - Regression

```
set.seed(208)

# random forest
param_grid_rf <- expand.grid(mtry = seq(1, 12, 1),    # for random forest
                             splitrule = "variance",
                             min.node.size = 2)

rf_cv <- train(blueprint,
               data = boston_train,
               method = "ranger",
               trControl = cv_specs,
               tuneGrid = param_grid_rf,
               metric = "RMSE")
```

```
set.seed(208)

# gradient boosting
out <- capture.output(gbm_cv <- train(blueprint,
                data = boston_train,
                method = "gbm",
                trControl = cv_specs,
                tuneLength = 5,
                metric = "RMSE"))
```

Practice - Regression

```
# optimal CV Accuracies

min(lm_cv$results$RMSE)
```

```
## [1] 5.000492
```

```
min(lasso_cv$results$RMSE)
```

```
## [1] 4.999851
```

```
min(knn_cv$results$RMSE)
```

```
## [1] 4.162816
```

```
min(tree_cv$results$RMSE)
```

```
## [1] 4.540773
```

Practice - Regression

```
# optimal CV Accuracies

bag_fit$err
```

```
## [1] 3.400379
```

```
min(rf_cv$results$RMSE)
```

```
## [1] 3.29768
```

```
min(gbm_cv$results$RMSE)
```

```
## [1] 3.318968
```

Random forest results in the best RMSE.

Practice - Regression

```
# optimal hyperparameters

lasso_cv$bestTune
```

```
##    alpha      lambda
## 14     1 0.006135907
```

```
knn_cv$bestTune
```

```
##   k
## 3 3
```

```
tree_cv$bestTune
```

```
##           cp
## 7 0.003078828
```

Practice - Regression

```
# optimal hyperparameters

rf_cv$bestTune
```

```
##   mtry splitrule min.node.size
## 4    4  variance             2
```

```
gbm_cv$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 24     200                 5       0.1             10
```

Practice - Regression

```
set.seed(208)

# build final model

final_model <- ranger(formula = medv ~ .,
                      data = baked_train,
                      num.trees = 500,
                      mtry = rf_cv$bestTune$mtry,
                      splitrule = "variance",
                      min.node.size = 2,
                      importance = "impurity")
```

```
# obtain predictions on test data

final_model_preds <- predict(final_model, data = baked_test, type = "response")

sqrt(mean((final_model_preds$predictions - baked_test$medv)^2))   # test set RMSE
```

```
## [1] 4.174782
```

Practice - Classification

You will work with the **`Sonar`** data from the **`mlbench`** package.

The task is to predict **`Class`** ('R' if the object is a rock and 'M' if it is a mine (metal cylinder)) using the rest of the variables as predictors.

- Use a 70-30 split.
- Use 5-fold CV with 1 repeat.
- Parameter grids are in Rmd file.

Practice - Classification

```
library(mlbench)   # load library

data(Sonar)     # load dataset
```

```
glimpse(Sonar)   # all features are numerical (output not displayed here)
```

```
sum(is.na(Sonar))  # no missing entries
```

```
## [1] 0
```

```
nearZeroVar(Sonar, saveMetrics = TRUE)   # no zv/nzv features (output not displayed here)
```

Practice - Classification

```
set.seed(208)   # set seed

# split the data into training and test sets
index <- createDataPartition(Sonar$Class, p = 0.7, list = FALSE)

Sonar_train <- Sonar[index, ]

Sonar_test <- Sonar[-index, ]
```

A blueprint is not required for this dataset since the features already seem centered and scaled, and there are no other feature engineering steps to implement.

Practice - Classification

```
set.seed(208)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 1)   # CV specifications
```

```
set.seed(208)

# logistic regression
logistic_cv <- train(Class ~ .,
                     data = Sonar_train,
                     method = "glm",
                     family = "binomial",
                     trControl = cv_specs,
                     metric = "Accuracy")
```

```
set.seed(208)

# KNN
k_grid <- expand.grid(k = seq(1, 10, by = 1))

knn_cv <- train(Class ~ .,
                data = Sonar_train,
                method = "knn",
                trControl = cv_specs,
                tuneGrid = k_grid,
                metric = "Accuracy")
```

Practice - Classification

```
set.seed(208)

# single classification tree
tree_cv <- train(Class ~ .,
                 data = Sonar_train,
                 method = "rpart",
                 trControl = cv_specs,
                 tuneLength = 20,
                 metric = "Accuracy")
```

```
set.seed(208)

# bagging
bag_fit <- bagging(Class ~ .,
                   data = Sonar_train,
                   nbagg = 500,
                   coob = TRUE,
                   control = rpart.control(minsplit = 2, cp = 0, xval = 0))
```

Practice - Classification

```
set.seed(208)

# random forest
param_grid_rf <- expand.grid(mtry = seq(1, 30, 1),    # for random forest
                             splitrule = "gini",
                             min.node.size = 2)

rf_cv <- train(Class ~ .,
               data = Sonar_train,
               method = "ranger",
               trControl = cv_specs,
               tuneGrid = param_grid_rf,
               metric = "Accuracy")
```

```
set.seed(208)

# gradient boosting
out <- capture.output(gbm_cv <- train(Class ~ .,
                data = Sonar_train,
                method = "gbm",
                trControl = cv_specs,
                tuneLength = 5,
                metric = "Accuracy"))
```

Practice - Classification

```
set.seed(208)

# SVM with linear kernel

param_grid_linear <- expand.grid(C = c(0.001, 0.1, 1, 5, 10, 100))

svc_cv <- train(Class ~ .,
                data = Sonar_train,
                method = "svmLinear",
                trControl = cv_specs,
                tuneGrid = param_grid_linear,
                metric = "Accuracy")
```

Practice - Classification

```
set.seed(208)

# SVM with polynomial kernel

param_grid_poly <- expand.grid(degree = c(1, 2, 3, 4),
                               scale = c(0.5, 1, 1.5, 2),
                               C = c(0.001, 0.1, 1, 5, 10, 100))

svm_poly_cv <- train(Class ~ .,
                     data = Sonar_train,
                     method = "svmPoly",
                     trControl = cv_specs,
                     tuneGrid = param_grid_poly,
                     metric = "Accuracy")
```

Practice - Classification

```
set.seed(208)

# SVM with radial basis function kernel

param_grid_radial <- expand.grid(sigma = c(0.5, 1, 1.5, 2),
                                 C = c(0.001, 0.1, 1, 5, 10, 100))

svm_radial_cv <- train(Class ~ .,
                       data = Sonar_train,
                       method = "svmRadial",
                       tuneGrid = param_grid_radial,
                       trControl = cv_specs,
                       metric = "Accuracy")
```

Practice - Classification

```
# optimal CV Accuracies

max(logistic_cv$results$Accuracy)
```

```
## [1] 0.7255172
```

```
max(knn_cv$results$Accuracy)
```

```
## [1] 0.774023
```

```
max(tree_cv$results$Accuracy)
```

```
## [1] 0.7457471
```

```
1- bag_fit$err
```

```
## [1] 0.7876712
```

```
max(rf_cv$results$Accuracy)
```

```
## [1] 0.8416092
```

```
max(gbm_cv$results$Accuracy)
```

```
## [1] 0.8625287
```

Practice - Classification

```
# optimal CV Accuracies

max(svc_cv$results$Accuracy)
```

```
## [1] 0.7666667
```

```
max(svm_poly_cv$results$Accuracy)
```

```
## [1] 0.8016092
```

```
max(svm_radial_cv$results$Accuracy)
```

```
## [1] 0.5616092
```

Gradient boosting results in the best accuracy.

Practice - Classification

```
# optimal hyperparameters

knn_cv$bestTune
```

```
##   k
## 1 1
```

```
tree_cv$bestTune
```

```
##          cp
## 2 0.0255418
```

```
rf_cv$bestTune
```

```
##    mtry splitrule min.node.size
## 26   26      gini             2
```

```
gbm_cv$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 23     150                 5       0.1             10
```

Practice - Classification

```
# optimal hyperparameters

svc_cv$bestTune
```

```
##   C
## 3 1
```

```
svm_poly_cv$bestTune
```

```
##    degree scale   C
## 26      2   0.5 0.1
```

```
svm_radial_cv$bestTune
```

```
##   sigma C
## 4   0.5 5
```

Practice - Classification

```
set.seed(208)

# build final model

final_model <- gbm(formula = ifelse(Class == "M", 0, 1) ~ .,
                   data = Sonar_train,
                   n.trees = gbm_cv$bestTune$n.trees,
                   interaction.depth = gbm_cv$bestTune$interaction.depth,
                   n.minobsinnode = gbm_cv$bestTune$n.minobsinnode,
                   shrinkage = gbm_cv$bestTune$shrinkage)
```

```
## Distribution not specified, assuming bernoulli ...
```

```
# obtain predictions on test data

final_model_prob_preds <- predict(final_model, newdata = Sonar_test, type = "response")  # probability predictions

threshold <- 0.5 # set threshold

final_model_class_preds <- ifelse(final_model_prob_preds > threshold, 1, 0)  # class label predictions
```

Practice - Classification

```
# confusion matrix

confusionMatrix(data = factor(final_model_class_preds), reference = factor(ifelse(Sonar_test$Class == "M", 0, 1)))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 30  6
##          1  3 23
##
##                Accuracy : 0.8548
##                  95% CI : (0.7422, 0.9314)
##     No Information Rate : 0.5323
##     P-Value [Acc > NIR] : 8.133e-08
##
##                   Kappa : 0.7066
##
##  Mcnemar's Test P-Value : 0.505
##
##             Sensitivity : 0.9091
##             Specificity : 0.7931
##          Pos Pred Value : 0.8333
##          Neg Pred Value : 0.8846
##              Prevalence : 0.5323
##          Detection Rate : 0.4839
##    Detection Prevalence : 0.5806
##       Balanced Accuracy : 0.8511
##
##        'Positive' Class : 0
##
```