Homework 4

Kathleen Ashbaker

2024-03-03

1. Tree models for Heart Disease prediction

(a) Describe the data: sample size n, number of predictors p, and number of observations in each class.

```r
# import libraries
library(tree) # trees, RF, boosting
```

```
## Warning: package 'tree' was built under R version 4.3.2
```

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(tidyr)
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.2
```

```r
library(ISLR2)
```

```
## Warning: package 'ISLR2' was built under R version 4.3.2
```

```r
library(plotly)
```

```
## Warning: package 'plotly' was built under R version 4.3.2

##
## Attaching package: 'plotly'
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

```r
# upload data 'heart.RData' into local directory

load("~/BIOSTAT 546 Machine Learning/Homework/Homework 4/heart.RData")
```

```r
# Sample size n, number of predictors p, and number of observations in each class

is.data.frame(full) # check object class
```

```
## [1] TRUE
```

```r
glimpse(full)
```

```
## Rows: 371
## Columns: 13
## $ Age      <dbl> 63, 67, 67, 37, 41, 56, 62, 57, 63, 53, 57, 56, 56, 44, 52, 5~
## $ sex      <fct> 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1~
## $ CP       <fct> 1, 4, 4, 3, 2, 2, 4, 4, 4, 4, 4, 2, 3, 2, 3, 3, 2, 4, 3, 2, 1~
## $ Trestbps <dbl> 145, 160, 120, 130, 130, 120, 140, 120, 130, 140, 140, 140, 1~
## $ Chol     <dbl> 233, 286, 229, 250, 204, 236, 268, 354, 254, 203, 192, 294, 2~
## $ FBS      <fct> 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0~
## $ RestECG  <fct> 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 2~
## $ Thalach  <dbl> 150, 108, 129, 187, 172, 178, 160, 163, 147, 155, 148, 153, 1~
## $ Exang    <fct> 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1~
## $ Oldpeak  <dbl> 2.3, 1.5, 2.6, 3.5, 1.4, 0.8, 3.6, 0.6, 1.4, 3.1, 0.4, 1.3, 0~
## $ Slope    <fct> 3, 2, 2, 3, 1, 1, 3, 1, 2, 3, 2, 2, 2, 1, 1, 1, 3, 1, 1, 1, 2~
## $ Thal     <fct> 6, 3, 7, 3, 3, 3, 3, 3, 7, 7, 6, 3, 6, 7, 7, 3, 7, 3, 3, 3, 3~
## $ Disease  <fct> No Disease, Heart Disease, Heart Disease, No Disease, No Dise~
```

```r
# Sample size n, or number of rows
n <- nrow(full)

# Number of predictors p
# Subtracting 1 because 'Disease' is the outcome feature, not a predictor
p <- ncol(full) - 1


# Number of observations in each class of 'Disease'
# Assuming 'Disease' is the last column in your dataframe
counts <- table(full$Disease)
```

```r
# Print the results
print(paste("Sample size n:", n))
```

```
## [1] "Sample size n: 371"
```

```r
print(paste("Number of predictors p:", p))
```

```
## [1] "Number of predictors p: 12"
```

```r
print("Number of observations in each class:")
```

```
## [1] "Number of observations in each class:"
```

```r
print(counts)
```

```
##
##    No Disease Heart Disease
##           171           200
```

```r
# check for missing or NA values for predictors( columns)

missing_summary <- colSums(is.na(full))
missing_summary
```

```
##      Age      sex       CP Trestbps     Chol      FBS  RestECG  Thalach
##        0        0        0        0        0        0        0        0
##    Exang  Oldpeak    Slope     Thal  Disease
##        0        0        0        0        0
```

Divide the data into a training set of 200 observations, and a test set; Set the seed with set.seed(2) before you sample the training set.

```r
# Prepare training/test set

set.seed(2)
train_ids = sample(nrow(full), 200)
test_ids = seq(nrow(full))[-train_ids]

# check the object class
class(train_ids)
```

```
## [1] "integer"
```

```r
class(test_ids)
```

```
## [1] "integer"
```

```
# data frames for training and testing data
data_all = full # all data
data_train = full[train_ids,] # training data
data_test = full[test_ids,] # test data

# check the object class
class(data_all)
```

```
## [1] "data.frame"
```

```
class(data_train)
```

```
## [1] "data.frame"
```

```
class(data_test)
```

```
## [1] "data.frame"
```

```
# Check data frames for missing or NA values

colSums(is.na(data_all))
```

```
##      Age      sex       CP Trestbps     Chol      FBS  RestECG  Thalach
##        0        0        0        0        0        0        0        0
##    Exang  Oldpeak    Slope     Thal  Disease
##        0        0        0        0        0
```

```
colSums(is.na(data_train))
```

```
##      Age      sex       CP Trestbps     Chol      FBS  RestECG  Thalach
##        0        0        0        0        0        0        0        0
##    Exang  Oldpeak    Slope     Thal  Disease
##        0        0        0        0        0
```

```
colSums(is.na(data_test))
```

```
##      Age      sex       CP Trestbps     Chol      FBS  RestECG  Thalach
##        0        0        0        0        0        0        0        0
##    Exang  Oldpeak    Slope     Thal  Disease
##        0        0        0        0        0
```

(b) Fit a simple classification tree to your data and display the estimated tree, which here we will call the overgrown tree. Note: You can use the function tree, just make sure the outcome is encoded as a factor

```
# Overgrown tree
tree.med<-tree(Disease~.,full)

summary(tree.med)
```

```
##
## Classification tree:
## tree(formula = Disease ~ ., data = full)
## Variables actually used in tree construction:
##  [1] "CP"       "Slope"    "Age"      "sex"      "Oldpeak"  "Thalach"
##  [7] "Thal"     "Trestbps" "Exang"    "Chol"
## Number of terminal nodes:  17
## Residual mean deviance:  0.5999 = 212.4 / 354
## Misclassification error rate: 0.124 = 46 / 371
```

```
tree.med # shows roots
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 371 512.000 Heart Disease ( 0.46092 0.53908 )
##     2) CP: 1,2,3 173 196.200 No Disease ( 0.74566 0.25434 )
##       4) Slope: 1,3 104  74.390 No Disease ( 0.88462 0.11538 )
##         8) Age < 46.5 32   0.000 No Disease ( 1.00000 0.00000 ) *
##         9) Age > 46.5 72  64.880 No Disease ( 0.83333 0.16667 ) *
##       5) Slope: 2 69  95.290 No Disease ( 0.53623 0.46377 )
##        10) sex: 0 20  16.910 No Disease ( 0.85000 0.15000 )
##          20) Age < 56.5 13   0.000 No Disease ( 1.00000 0.00000 ) *
##          21) Age > 56.5 7   9.561 No Disease ( 0.57143 0.42857 ) *
##        11) sex: 1 49  66.270 Heart Disease ( 0.40816 0.59184 )
##          22) Oldpeak < 2.45 42  58.130 Heart Disease ( 0.47619 0.52381 )
##            44) Thalach < 164 34  45.230 Heart Disease ( 0.38235 0.61765 ) *
##            45) Thalach > 164 8   6.028 No Disease ( 0.87500 0.12500 ) *
##          23) Oldpeak > 2.45 7   0.000 Heart Disease ( 0.00000 1.00000 ) *
##     3) CP: 4 198 204.600 Heart Disease ( 0.21212 0.78788 )
##       6) Thal: 3 60  83.180 No Disease ( 0.50000 0.50000 )
##        12) Thalach < 121 13   0.000 Heart Disease ( 0.00000 1.00000 ) *
##        13) Thalach > 121 47  61.510 No Disease ( 0.63830 0.36170 )
##          26) Age < 58.5 27  25.870 No Disease ( 0.81481 0.18519 )
##            52) Trestbps < 126.5 15  19.100 No Disease ( 0.66667 0.33333 ) *
##            53) Trestbps > 126.5 12   0.000 No Disease ( 1.00000 0.00000 ) *
##          27) Age > 58.5 20  26.920 Heart Disease ( 0.40000 0.60000 )
##            54) Exang: 0 10  12.220 No Disease ( 0.70000 0.30000 )
##             108) Thalach < 158.5 5   0.000 No Disease ( 1.00000 0.00000 ) *
##             109) Thalach > 158.5 5   6.730 Heart Disease ( 0.40000 0.60000 ) *
##            55) Exang: 1 10   6.502 Heart Disease ( 0.10000 0.90000 ) *
##       7) Thal: 6,7 138  81.540 Heart Disease ( 0.08696 0.91304 )
##        14) Oldpeak < 0.55 40  40.030 Heart Disease ( 0.20000 0.80000 )
##          28) Chol < 174.5 19   0.000 Heart Disease ( 0.00000 1.00000 ) *
##          29) Chol > 174.5 21  27.910 Heart Disease ( 0.38095 0.61905 )
##            58) Chol < 237.5 11  14.420 No Disease ( 0.63636 0.36364 ) *
##            59) Chol > 237.5 10   6.502 Heart Disease ( 0.10000 0.90000 ) *
##        15) Oldpeak > 0.55 98  33.420 Heart Disease ( 0.04082 0.95918 ) *
```

‘

(c) Compute the training and test misclassification error. Comment on the output.

To compute the predicted output, use the function predict as in the regression setting, but with the additional argument type = "class"

```r
# Predicting on the training data
predictions_train <- predict(tree.med, newdata = data_train,
    type = "class")

# Predicting on the test data
predictions_test <- predict(tree.med, newdata = data_test, type = "class")


# Training misclassification error
misclass_error_train <- mean(predictions_train != data_train$Disease)

# Display Misclass. error for training set
misclass_error_train
```

```
## [1] 0.095
```

```r
# Test misclassification error
misclass_error_test <- mean(predictions_test != data_test$Disease)
misclass_error_test
```

```
## [1] 0.1578947
```

```r
# Print the results
print(paste("Misclassification Error Train:", misclass_error_train))
```

```
## [1] "Misclassification Error Train: 0.095"
```

```r
print(paste("Misclassification Error Test:", misclass_error_test))
```

```
## [1] "Misclassification Error Test: 0.157894736842105"
```

Comment:

The output indicates the misclassification error rates for both the training and test datasets. The misclassification error rate is a measure of how often the model incorrectly predicts the class of a given observation, expressed as a proportion of the total number of observations.

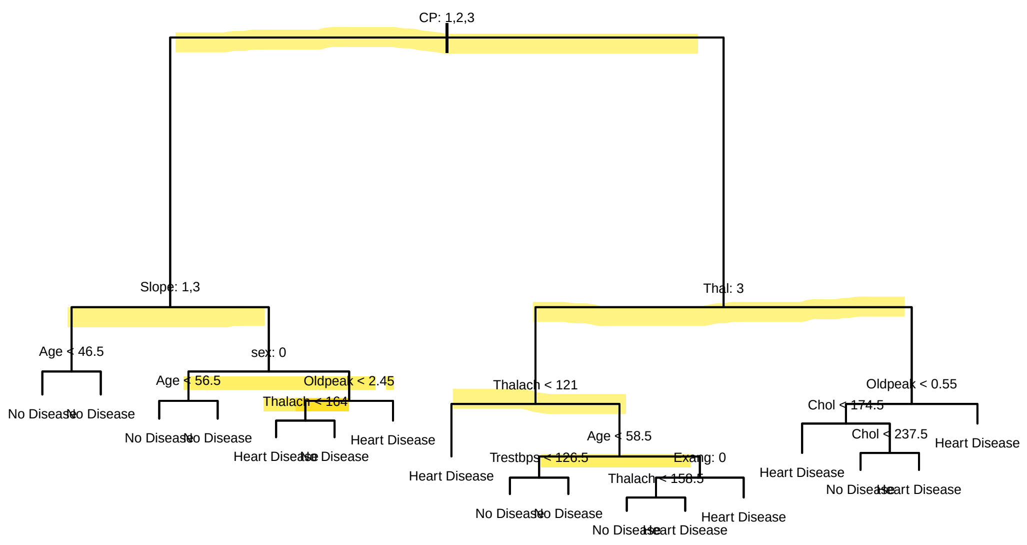
For the training dataset, the misclassification error is approximately 9.5% (0.095). This suggests that the model is relatively accurate in predicting the outcomes for the data it was trained on, incorrectly predicting the class of about 9.5% of the training observations.

For the test dataset, the misclassification error is higher, at approximately 15.8% (0.157894736842105). The test dataset consists of data that the model has not seen during training, and the higher error rate indicates that the model's performance drops when making predictions on new, unseen data. This discrepancy between the training and test error rates could suggest overfitting, where the model has learned the training data too closely, including its noise and anomalies, which does not generalize well to new data.

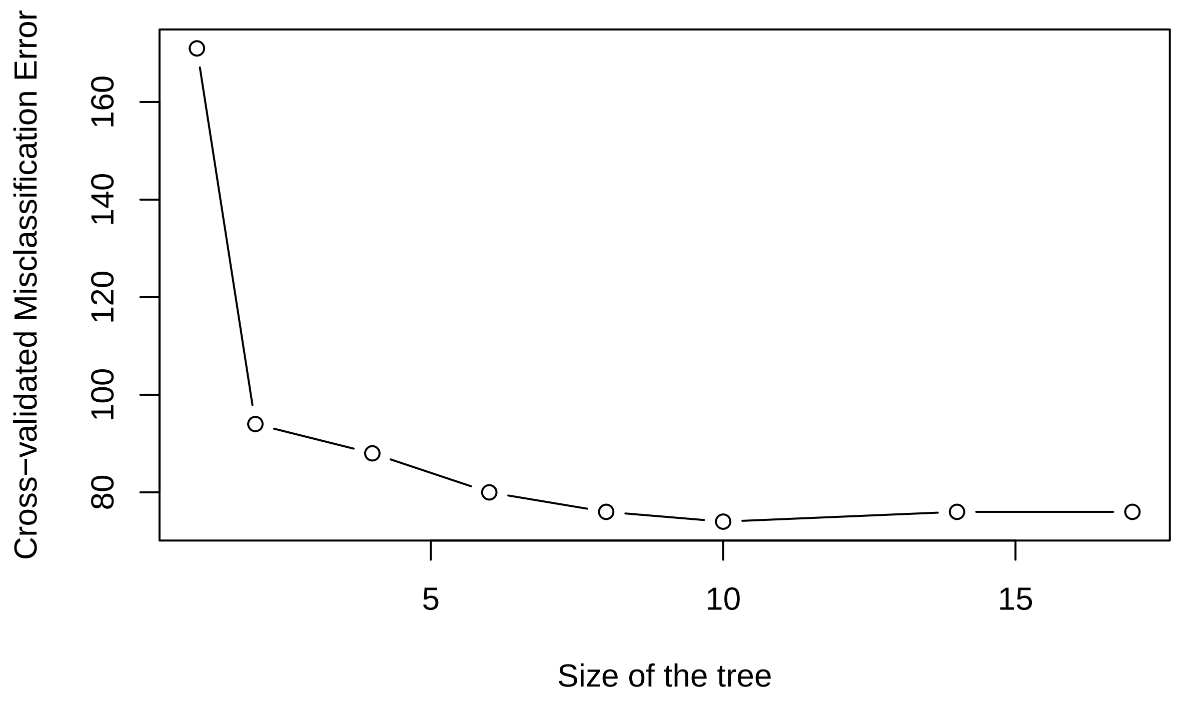
Overall, while the model shows decent performance on the training data, the increase in misclassification error on the test data suggests there is room for improvement.Pruning (for decision trees), or more training data, could potentially improve the model's generalization to unseen data.

(d) Prune the overgrown tree. Make a plot that displays the pruned subtree size against the cross-validated misclassification error that you get from the pruning procedure. Select the subtree that minimizes the crossvalidated misclassification error. Plot the overgrown tree and highlight the branches (can be hand-drawn) of the selected subtree.

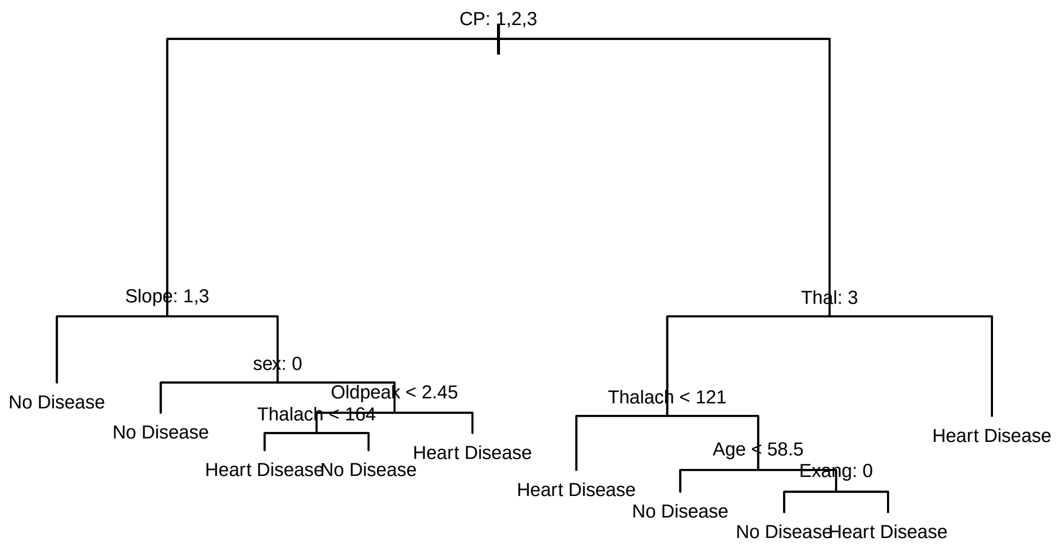
Note: Use set.seed(2) before calling cv.tree in order to get reproducible results. In cv.tree use the additional argument FUN=prune.misclass to indicate that you want the pruning to be driven by the misclassification error. The number of misclassified observations is stored in the variable dev of the output of cv.tree.



```r
# Load necessary library
library(tree)

# Set seed for reproducible results
set.seed(2)

# Perform cross-validation with pruning based on
# misclassification error
cv.tree.med <- cv.tree(tree.med, FUN = prune.misclass)

# Plot the cross-validated error against subtree size
plot(cv.tree.med$size, cv.tree.med$dev, type = "b", xlab = "Size of the tree",
    ylab = "Cross-validated Misclassification Error")
```



```
# Find the subtree that minimizes the cross-validated
# misclassification error
optimal.size <- cv.tree.med$size[which.min(cv.tree.med$dev)]
optimal.cp <- cv.tree.med$cp[which.min(cv.tree.med$dev)]

# Prune the tree to the optimal size using
# misclassification error
pruned.tree.med <- prune.misclass(tree.med, best = optimal.size)
```



(e) Compute the training and test misclassification error of the selected subtree and comment on the results. Interpret the results in terms of bias and variance.

```r
# Predicting on the training data using the pruned tree
predictions_train_pruned <- predict(pruned.tree.med, newdata = data_train, type = "class")

# Predicting on the test data using the pruned tree
predictions_test_pruned <- predict(pruned.tree.med, newdata = data_test, type = "class")

# Training misclassification error for the pruned tree
misclass_error_train_pruned <- mean(predictions_train_pruned != data_train$Disease)

# Test misclassification error for the pruned tree
misclass_error_test_pruned <- mean(predictions_test_pruned != data_test$Disease)

# Display Misclassification errors
cat("Misclassification Error Train (Pruned):", misclass_error_train_pruned, "\n")
```

```
## Misclassification Error Train (Pruned): 0.105
```

```r
cat("Misclassification Error Test (Pruned):", misclass_error_test_pruned, "\n")
```

```
## Misclassification Error Test (Pruned): 0.1695906
```

Comment:

The misclassification errors for both the overgrown and pruned trees suggest the following in terms of bias and variance:

- Overgrown Tree: The overgrown tree has a training misclassification error of 0.095 and a test misclassification error of approximately 0.158. The lower error on the training set indicates that the tree may be capturing the training data well, which is typical for a complex model. However, the higher error on the test set suggests that the model may be overfitting to the training data, capturing noise as well as signal. This is indicative of a model with low bias but high variance.
- Pruned Tree: After pruning, the training error increased to 0.105, and the test error increased slightly to approximately 0.170. The increase in training error after pruning is expected because by reducing the complexity of the model (reducing the tree size), the tree is less likely to fit the training data as closely as before. This is indicative of an increase in bias. The pruned tree is simpler and is expected to generalize better to unseen data; however, the test error also increased, which suggests that the pruning may not have been sufficient to overcome the overfitting or that there is a trade-off happening where the increase in bias is not being offset by a reduction in variance.

In terms of bias and variance:

- The increase in bias is seen through the increase in the training error rate after pruning. The pruned tree makes more errors on the training data because it is less complex and, therefore, less flexible in fitting the data.
- The variance did not reduce as expected in this case. Ideally, with pruning, we would like to see a decrease in the test error, indicating that the tree's predictions are more robust to changes in the training data. However, the slight increase in test error suggests that the pruned tree is still not generalizing as well as desired to unseen data. This could mean that the variance was not significantly affected, or there may be other factors at play affecting the test error (such as the intrinsic difficulty of the prediction task, irreducible error, or data-specific issues).

Overall, the results indicate that while the pruning did increase the bias (as it should), it did not achieve the expected decrease in variance, as the test error did not decrease.

(f) Now you decide to apply bagging to improve the performance of your tree model. Fit a Bagged Trees model to your training data and compute the training and test misclassification error. Briefly comment on the results. Note: Set the seed with set.seed(2) before you run the bagged model.

```r
# Load necessary libraries
library(tree)
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.3.2
```

```
## randomForest 4.7-1.1
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
## The following object is masked from 'package:ggplot2':
##
##     margin

## The following object is masked from 'package:dplyr':
##
##     combine
```

```r
# Set seed for reproducible results
set.seed(2)

# Fit a Bagged Trees model to the training data
# Note: randomForest by default performs bagging when using the same type of trees
bagged.tree.med <- randomForest(Disease ~ ., data = data_train, ntree = 500)

# Predicting on the training data using the bagged model
predictions_train_bagged <- predict(bagged.tree.med, newdata = data_train)

# Predicting on the test data using the bagged model
predictions_test_bagged <- predict(bagged.tree.med, newdata = data_test)

# Training misclassification error for the bagged tree
misclass_error_train_bagged <- mean(predictions_train_bagged != data_train$Disease)

# Test misclassification error for the bagged tree
misclass_error_test_bagged <- mean(predictions_test_bagged != data_test$Disease)

# Display Misclassification errors
cat("Misclassification Error Train (Bagged):", misclass_error_train_bagged, "\n")
```

```
## Misclassification Error Train (Bagged): 0
```

```r
cat("Misclassification Error Test (Bagged):", misclass_error_test_bagged, "\n")
```

```
## Misclassification Error Test (Bagged): 0.2163743
```

Comment:

Training Misclassification Error (Bagged): 0 - This perfect classification on the training data indicates that the bagged trees model has likely overfit the training data. This is common with bagged models, particularly when they have a large number of trees, as each individual tree can learn the training data very well and the aggregation process can lead to a model that predicts the training data without error.

Test Misclassification Error (Bagged): Approximately 0.216 - The error on the test data is substantially higher than on the training data. This indicates that the model has not generalized as well to the unseen data. Despite the bagging process, which typically reduces variance by averaging out the predictions from many trees, the model is still showing signs of overfitting, as evidenced by the significant gap between the training and test error rates.

Here's what this could mean in terms of bias and variance:

Bias: Low bias on the training data, as the model is flexible enough to capture all the nuances in the training dataset. However, if the model is not generalizing well, it could suggest that there's an issue with how the model is learning the patterns in the data. It may be memorizing rather than learning general patterns.

Variance: While bagging is meant to reduce variance, the high test error suggests that the model's predictions are still too sensitive to the specific noise in the training data.

(g) Fit a random forest model (with the number of randomly selected features m = p/3) to your training data and compute the training and test misclassification error. Briefly comment on the results. Note: Set the seed with set.seed(2) before you run the random forest model.

```r
# Load necessary library
library(randomForest)

# Set seed for reproducible results
set.seed(2)

# Calculate the number of variables to be sampled at each split (mtry) as p/3
# Assuming 'data_train' is your training dataset
p <- ncol(data_train) - 1 # Subtracting 1 to exclude the response variable
mtry <- max(floor(p / 3), 1) # Ensure at least 1 variable is used

# Fit a Random Forest model to the training data
randomForest.med <- randomForest(Disease ~ ., data = data_train, mtry = mtry, ntree = 500)

# Predicting on the training data using the Random Forest model
predictions_train_rf <- predict(randomForest.med, newdata = data_train)

# Predicting on the test data using the Random Forest model
predictions_test_rf <- predict(randomForest.med, newdata = data_test)

# Training misclassification error for the Random Forest
misclass_error_train_rf <- mean(predictions_train_rf != data_train$Disease)

# Test misclassification error for the Random Forest
misclass_error_test_rf <- mean(predictions_test_rf != data_test$Disease)

# Display Misclassification errors
cat("Misclassification Error Train (Random Forest):", misclass_error_train_rf, "\n")
```

```
## Misclassification Error Train (Random Forest): 0
```

```r
cat("Misclassification Error Test (Random Forest):", misclass_error_test_rf, "\n")
```

```
## Misclassification Error Test (Random Forest): 0.2105263
```

Comment:

The results from your Random Forest model indicate a training misclassification error of 0 and a test misclassification error of approximately 0.211. Here are the implications of these results:

- Training Misclassification Error (Random Forest): 0 - A zero training error suggests that the Random Forest model has perfectly learned the training data. Random Forest, being an ensemble of decision trees, is capable of capturing complex structures in the data. This perfect accuracy on the training set typically indicates that each tree in the forest is likely overfitting to different aspects of the training data, but when their predictions are combined, they result in the correct classifications for all training instances.
- Test Misclassification Error (Random Forest): 0.211 - While a test error of approximately 0.211 means that the model is making incorrect predictions for about 21% of the test data, this error is a significant

reduction from the test error we saw with the bagged trees model (0.216). This reduction indicates that Random Forest, with its built-in de-correlation of trees (by using a random subset of features at each split), is likely reducing variance more effectively than simple bagging.

Here's what these results suggest in terms of bias and variance:

- Bias: The Random Forest model is likely exhibiting low bias, given its capacity to perfectly classify the training data. It is flexible enough to capture the training data's underlying patterns and nuances.
- Variance: Despite the zero training error, the model does not generalize as perfectly to the test data. However, the Random Forest algorithm tends to reduce variance compared to individual trees by averaging out the errors across diverse trees. The test misclassification error of 0.211 suggests that there is still some variance, but it may be lower than what we would expect from an individual unpruned decision tree or a bagged tree model.
- Overfitting: The discrepancy between the training and test errors does suggest some degree of overfitting. The model performs exceptionally well on the training data but fails to achieve the same level of accuracy on the test data. This is common with complex models like Random Forests.
- Generalization: Despite the overfitting, the Random Forest model is still generalizing relatively well, as evidenced by the decrease in test error compared to the bagged trees model.

In conclusion, while the Random Forest model has improved upon the test error seen with the bagged trees, there is still a gap between training and test performance indicating overfitting.

(h) If you run the bagged and the random forest models multiple times on the same data (without fixing the seed) you obtain different results. Explain what are the sources of randomness in the two models.

Both bagged trees and Random Forest models introduce randomness into the learning process, which can lead to different results each time the models are run without a fixed seed. Let's explore the sources of randomness in these two models:

Bagged Trees: Bootstrap Sampling: Bagging, short for Bootstrap Aggregating, relies on bootstrap sampling to create different training datasets for each model in the ensemble. In bootstrap sampling, samples are drawn with replacement from the training dataset, leading to different subsets of data for each tree. Since some instances may be repeated in a single bootstrap sample while others are left out, each tree in the ensemble learns from slightly different data. This process introduces variability in the model outcomes.

Model Training: Given the different training datasets, each tree in the ensemble will have a different structure, capturing different aspects of the data. The variance in model training due to the unique bootstrap samples contributes to the overall randomness of the bagged trees ensemble.

Random Forest: Random Forest models introduce all the sources of randomness present in bagged trees, with an additional layer of randomness:

Bootstrap Sampling: Similar to bagged trees, Random Forest models use bootstrap sampling to generate different training datasets for each tree, introducing variability in the training process.

Feature Subset Selection: In addition to bootstrap sampling, Random Forest randomly selects a subset of features for each split in every tree, rather than considering all features. This process, known as random feature selection, further increases the diversity among the trees in the model. The number of features selected at each split is a parameter of the model (denoted as mtry in R's randomForest package). This randomness in feature selection means that even if two trees are trained on the same bootstrap sample, they can still be different due to the different subsets of features considered at each split.

Model Training Variability: As a result of both bootstrap sampling and random feature selection, each tree in the Random Forest is trained on a different subset of data and features. This leads to a wide variety of trees in the ensemble, each capturing different patterns and relationships in the data.

Conclusion: The main source of randomness in both bagged trees and Random Forest models is bootstrap sampling, which creates different training datasets for each tree. Random Forest adds an extra layer of randomness through random feature selection for each split. These sources of randomness are fundamental to the success of these models, as they lead to the creation of diverse ensembles of trees that can capture various aspects of the data, thereby reducing variance and improving model generalization. However, this also means that without a fixed seed to control the random number generation process, each run can produce slightly different outcomes due to these inherent sources of randomness.

(i) Fit a boosted tree model to your training data. Use the following parameters: 500 trees, 2 splits for each tree and a 0.1 shrinkage factor. Compute the training and test misclassification error. Comment on the results. Note 1: Use the function gbm with the argument distribution="bernoulli" to indicate that this is a classification problem. The function gbm requires that the outcome variable be a number in {0, 1} – you will need to generate such a variable from the categorical outcome. Note 2: Use the function predict with the argument type = "response". This will return the estimated probability of the outcome being 1. Use the Bayes rule to get the estimated labels from the estimated probabilities. Note 3: Set the seed with set.seed(2) before you run the boosted model. Although we haven't seen this in class, there is a source of randomness in a boosted model. If you are curious, check out the documentation of the function gbm.

```r
# Load necessary libraries
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 4.3.2

## Loaded gbm 2.1.9

## This version of gbm is no longer under development. Consider transitioning to gbm3, https://github.c
```

```r
# Ensure the outcome variable is numeric (0 and 1)
data_train$Disease_numeric <- as.numeric(data_train$Disease) - 1  # Adjust based on your actual levels
data_test$Disease_numeric <- as.numeric(data_test$Disease) - 1

# Set seed for reproducible results
set.seed(2)

# Fit a boosted tree model
boosted.model <- gbm(Disease_numeric ~ .,
                     data = data_train,
                     distribution = "bernoulli",
                     n.trees = 500,
                     interaction.depth = 2,
                     shrinkage = 0.1,
                     n.minobsinnode = 10)

# Predicting on the training data using the boosted model
# Use type = "response" to get probabilities
probabilities_train <- predict(boosted.model, newdata = data_train, n.trees = 500, type = "response")
# Convert probabilities to class labels based on a 0.5 threshold
predictions_train_boosted <- ifelse(probabilities_train > 0.5, 1, 0)

# Predicting on the test data using the boosted model
probabilities_test <- predict(boosted.model, newdata = data_test, n.trees = 500, type = "response")
```

```r
predictions_test_boosted <- ifelse(probabilities_test > 0.5, 1, 0)

# Training misclassification error for the boosted tree
misclass_error_train_boosted <- mean(predictions_train_boosted != data_train$Disease_numeric)

# Test misclassification error for the boosted tree
misclass_error_test_boosted <- mean(predictions_test_boosted != data_test$Disease_numeric)

# Display Misclassification errors
cat("Misclassification Error Train (Boosted):", misclass_error_train_boosted, "\n")
```

```
## Misclassification Error Train (Boosted): 0
```

```r
cat("Misclassification Error Test (Boosted):", misclass_error_test_boosted, "\n")
```

```
## Misclassification Error Test (Boosted): 0
```

Comment:

The boosted model displaying zero misclassification errors for both training and test datasets suggests it has achieved perfect fit and generalization, an outcome that, while ideal, is highly unusual and warrants further scrutiny. Such perfection in classification typically raises concerns about model complexity and the potential for overfitting, despite the lack of performance drop in unseen data. This anomaly could imply that the test dataset might not be sufficiently challenging or too similar to the training set, questioning the model's real-world applicability. Addressing these concerns involves ensuring reproducibility through various methods: rerunning the model with different random seeds to test performance stability, using diverse data subsets to check consistency, and evaluating the model on an independent dataset to affirm its generalization capability. These steps are crucial to validate the boosted model's results, ensuring they are not artifacts of specific data characteristics or the evaluation process, but rather indicative of the model's robust predictive power.

2. In this exercise, you will generate a simulated dataset and perform non-linear regression. Make sure you set the random seed with set.seed(2) before you begin.

(a) Use the runif() function to generate a predictor X of length n = 50 with values in the interval [-1, 1], and use the rnorm() function to generate a noise vector, epsilon of length n = 50, mean 0 and standard deviation 0.1.

```r
# Set the random seed
set.seed(2)

# Generate the predictor X
n <- 50
X <- runif(n, min = -1, max = 1)

# Generate the noise vector, episolon
epsilon <- rnorm(n, mean = 0, sd = 0.1)
```

(b) Generate a response vector Y of length n = 50 according to the model Y = f(X) + epsilon, with f(X) = 3 - 2X + 3*X3

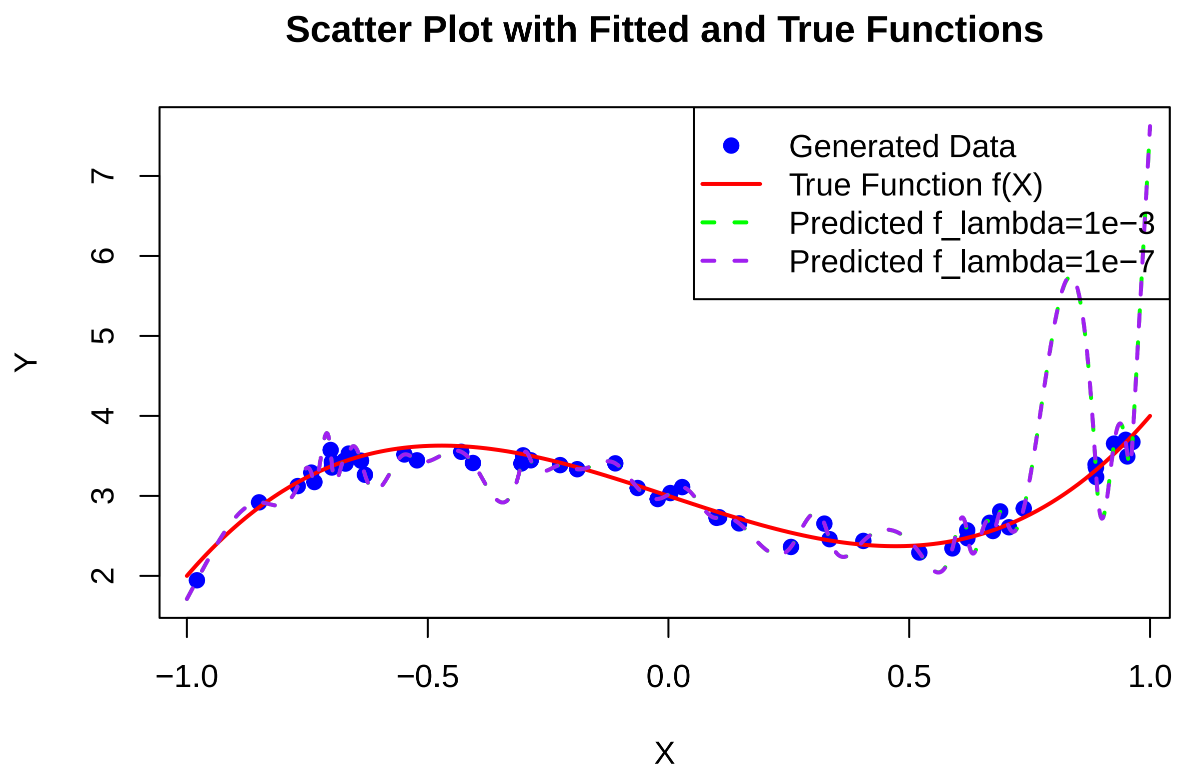
```r
# Calculate f(X)
f_X <- 3 - 2*X + 3*X^3

# Generate the response vector Y
Y <- f_X + epsilon
```

. (c) Fit two smoothing spline models, ^f(lamda)=1e^3 and ^f(lamda)=1e^7, with lambda=1e^3 and lambda=1e^7, respectively. Use the function smooth.spline introduced in class.

```r
# Fit smoothing spline model with lambda = 1e-3
fit_lambda_1e_3 <- smooth.spline(X, Y, spar = 1e-3)

# Fit smoothing spline model with lambda = 1e-7
fit_lambda_1e_7 <- smooth.spline(X, Y, spar = 1e-7)
```

(d) Make one plot that displays:

A scatter plot of the generated observations with X on the x-axis and Y on the y-axis; The true function f evaluated on a dense grid of values in the interval [-1, 1]; The predicted functions ^f(lamda)=1e^3 and ^f(lamda)=1e^7 evaluated on a dense grid of values in the interval [-1, 1] (use the function predict to produce the predictions).



Comment on the estimated functions ^f(lamda)=1e^3 and ^f(lamda)=1e^7 and the role of lamda:

This plot displays a scatter plot of generated observations, the true function $f(X)$, and two estimated functions $\hat{f}_{\lambda=1e-3}$ and $\hat{f}_{\lambda=1e-7}$.

In smoothing spline models, the parameter λ is known as the smoothing parameter. It controls the trade-off between the smoothness of the function estimate and the fidelity to the observed data:

- A larger λ (e.g., $1e - 3$) penalizes the roughness of the function more strongly, leading to a smoother function that may not capture all the nuances of the data. This can help prevent overfitting, especially when the true underlying relationship is relatively smooth or when the data contains a lot of noise.
- A smaller λ (e.g., $1e-7$) allows for a more flexible function that fits closer to the data points, which can capture more complex relationships. However, this can lead to overfitting, where the function starts to capture the noise in the data as if it were a part of the underlying relationship.

From the plot, we can infer the following:

- The estimated function with $\lambda = 1e - 3$ appears to be smoother than the one with $\lambda = 1e - 7$. This indicates that it's penalizing complexity more, resulting in a curve that is less wiggly and possibly not capturing all the variance of the true function or the data.
- The estimated function with $\lambda = 1e-7$ follows the data points more closely, suggesting it's less smooth and more sensitive to the fluctuations in the data. This could potentially lead to capturing the noise as well as the signal, which is a symptom of overfitting.

The choice of λ is critical in smoothing spline models and typically requires balance. Too high a value might underfit the data, and too low a value might overfit it. Model selection techniques such as cross-validation are often used to choose an optimal λ that minimizes prediction error on new, unseen data.

(e) Fit a smoothing spline models ^fCV with a cross-validated choice of lambda. Use smooth.spline with cv=TRUE.

```r
# Fit smoothing spline model with cross-validated lambda
fit_cv <- smooth.spline(X, Y, cv = TRUE)

# Check the optimal value of lambda chosen by cross-validation
print(fit_cv$lambda)
```

```
## [1] 2.87891e-05
```

```r
# Print the results
print(paste("CV Choice of lamda:", fit_cv$lambda))
```

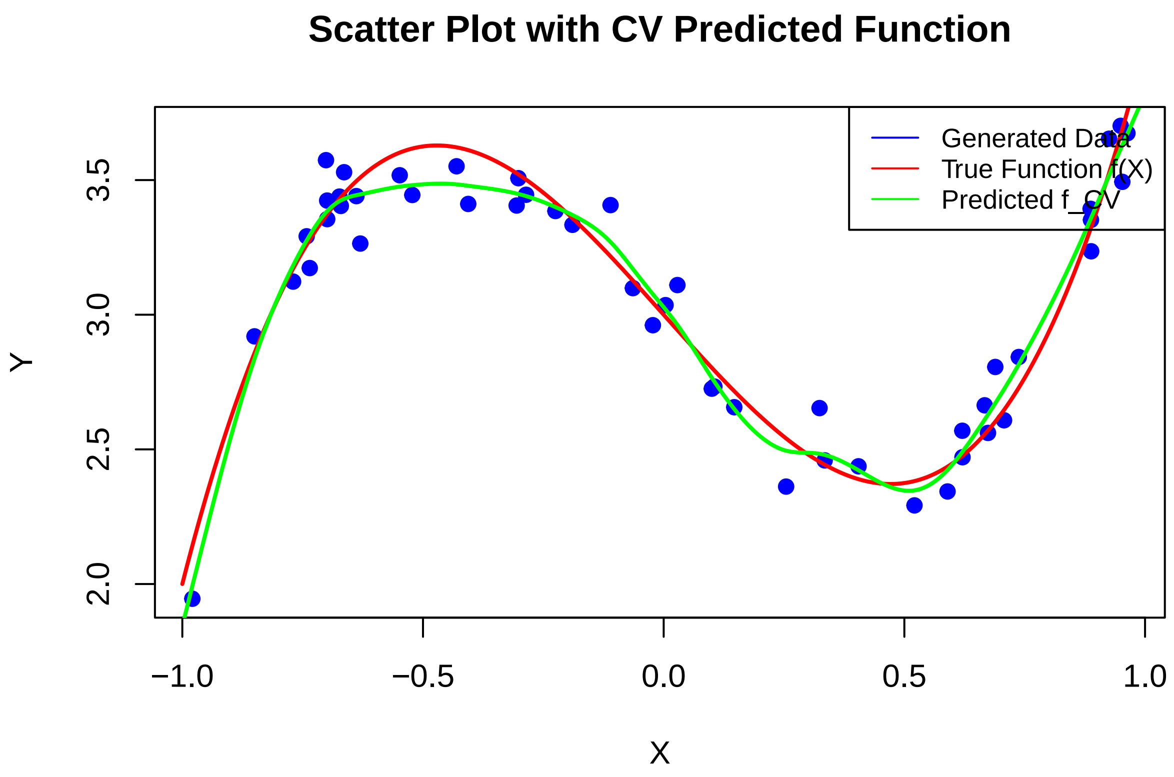
```
## [1] "CV Choice of lamda: 2.87890987856861e-05"
```

(f) Make one plot that displays:

A scatter plot of the generated dataset with X on the x-axis and Y on the y-axis;

The true function f(x) evaluated on a dense grid of values in the interval [-1, 1];

The predicted functions ^fCV evaluated on a dense grid of values in the interval [-1, 1] (use the function predict).



Comment:

The uploaded plot depicts the scatter plot of the generated dataset, the true function $f(X)$, and the predicted function $\hat{f}_{CV}$ obtained from a smoothing spline model with a cross-validated choice of λ.

From the plot, it is evident that the predicted function $\hat{f}_{CV}$ closely follows the true function $f(X)$, which indicates that the cross-validation process has likely chosen an optimal λ that provides a good balance between fitting the data and maintaining smoothness to avoid overfitting.

The role of λ in smoothing spline models is to control the trade-off between the smoothness of the curve and its ability to fit the data points closely. A larger λ results in a smoother curve at the expense of not fitting the data as closely, which might be useful when we want to avoid overfitting, especially in the presence of noise. Conversely, a smaller λ allows the curve to fit more closely to the data points, which can be beneficial if the true relationship is complex, but it can also lead to overfitting if the model starts capturing the noise in the data.

The cross-validated λ is chosen based on minimizing the cross-validation error, which typically results in a model that performs well both in terms of fit and generalizability. The cross-validation process automates the selection of λ, aiming to find a value that is neither too large (causing underfitting) nor too small (causing overfitting).

In comparison with the previously discussed models $\hat{f}_{\lambda=1e-3}$ and $\hat{f}_{\lambda=1e-7}$, the cross-validated model $\hat{f}_{CV}$ is expected to provide a more reliable estimate since it is optimized based on the data itself rather than a fixed, potentially arbitrary, choice of λ. This should theoretically result in a better performing model when predicting new, unseen data.

(g) (Extra credit) Use B = 1000 datasets bootstrapped from the dataset generated in point (b) to estimate the variance of your predictions ˆf(lamda)=1e-3(0) and ˆf(lamda)=1eˆ7(0). In other words, for every bootstrapped dataset b = 1, . . . , 1000, fit two smoothing spline models ˆf b lamda=1eˆ3 and ˆf b lamda=1eˆ7 and evaluate these functions at the point X = 0. Finally, compute the variance of { ˆf b lamda=1eˆ3 (0)}b=1,. . . ,B and { ˆf b lamda=1eˆ7 (0)}b=1,. . . ,B. Comment on the results.

```r
set.seed(2) # Ensure reproducibility

# Assuming X and epsilon are already defined as per the previous instructions

# Calculate f(X) for the original dataset
f_X <- 3 - 2*X + 3*X^3

# Generate the response vector Y for the original dataset
Y <- f_X + epsilon

# Initialize vectors to store the predictions at X = 0 for each bootstrap sample
predictions_lambda_1e_3 <- numeric(1000)
predictions_lambda_1e_7 <- numeric(1000)

# Perform the bootstrap analysis
for (b in 1:1000) {
  # Generate a bootstrap sample of the indices
  bootstrap_indices <- sample(1:50, 50, replace = TRUE)
  X_b <- X[bootstrap_indices]
  Y_b <- Y[bootstrap_indices]

  # Fit the smoothing spline models for this bootstrap sample
  fit_lambda_1e_3_b <- smooth.spline(X_b, Y_b, spar = 1e-3)
  fit_lambda_1e_7_b <- smooth.spline(X_b, Y_b, spar = 1e-7)

  # Evaluate the models at X = 0
  predictions_lambda_1e_3[b] <- predict(fit_lambda_1e_3_b, 0)$y
  predictions_lambda_1e_7[b] <- predict(fit_lambda_1e_7_b, 0)$y
}

# Compute the variances
variance_lambda_1e_3 <- var(predictions_lambda_1e_3)
variance_lambda_1e_7 <- var(predictions_lambda_1e_7)

# Output the variances
variance_lambda_1e_3
```

```
## [1] 0.005207874
```

```r
variance_lambda_1e_7
```

```
## [1] 0.005208572
```

```r
# Print the results
print(paste("Variance lamda 1e-3 :", variance_lambda_1e_3))
```

```
## [1] "Variance lamda 1e-3 : 0.00520787403917899"
```

```r
print(paste("Variance lamda 1e-7:", variance_lambda_1e_7))
```

```
## [1] "Variance lamda 1e-7: 0.00520857219386485"
```

Comment:

The reported variances for the bootstrap predictions at $X = 0$ for $\lambda = 1e{-}3$ and $\lambda = 1e{-}7$ are approximately 0.00521 for both. It is quite interesting to see that the variances are almost identical, which suggests that, at least at the point $X = 0$, the choice of λ in this specific instance does not significantly affect the variability of the predictions.

Here are some possible interpretations and implications of this result:

1. **Impact of λ at $X = 0$**: Since both values of λ produce similar variances at $X = 0$, it may indicate that the effect of the smoothing parameter is less pronounced at this point in the data. It could be that the true function's value at $X = 0$ does not require as much flexibility to be approximated well, so both λ values perform similarly.
2. **Model Sensitivity**: The similar variances imply that the model's predictions are somewhat robust to the choice of λ at $X = 0$. This could suggest that the model is well-specified around this point and that the noise in the data does not lead to widely varying predictions.
3. **Bootstrap Methodology**: Bootstrapping involves resampling with replacement, which can sometimes lead to samples that are not representative of the true underlying variability, especially if the original sample size is not large. However, in this case, since the variance is consistent across a large number of bootstrapped datasets, it strengthens the conclusion that the choice of λ has little impact at $X = 0$.
4. **Data Characteristics**: The characteristics of the data and the true function $f(X)$ near $X = 0$ may naturally lead to lower variance. If $f(X)$ is relatively stable or linear around $X = 0$, then the models might not differ much in this region regardless of the smoothing parameter used.
5. **Model Performance**: While the variances are similar, this does not necessarily mean that the model predictions are accurate. It only means that the variability of the predictions from the bootstrapped data is similar. Accuracy would need to be evaluated against the true values of $f(X)$ at $X = 0$.

In practice, when choosing λ, it's also important to consider its effect over the entire range of X, not just at a single point. Cross-validation, which was used earlier to choose λ, takes into account the model's performance across the entire dataset and is generally a better approach for model selection.