STA 561 HW2 (Decision Trees)

Daniel Truver

1/29/2018

(1) Classifiers for Basketball Courts

```
perceptron = function(X, y, I){
  iteration = 1
  w = rep(0, ncol(X))
  for (j in 1:I){
    for (i in seq_along(y)){
      if (y[i]*(w %*% X[i,]) <= 0){
        w = w + y[i]*X[i,] # update step
        iteration = iteration + 1
      }
    }
    accuracy = sum((X %*% w) * y > 0)/length(y) # calculate propotion of correctly classified point
    if (accuracy == 1){
      break # no need to continue if we have perfect separation
    }
  }
  return(list("iteration" = iteration, "w" = w, "accuracy" = accuracy))
}
```

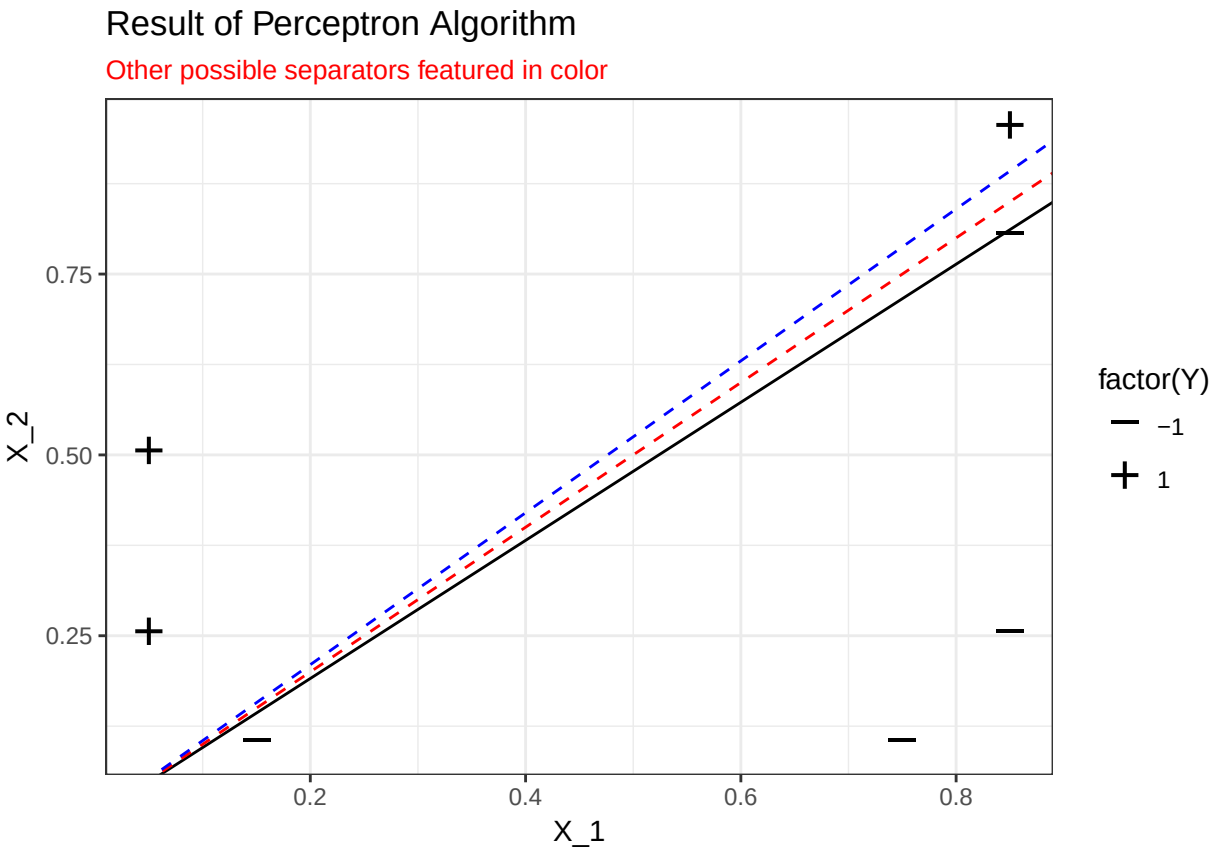
(a) Let's run the perceptron because it's fun.

```
X_1 = c(.75, .85, .85, .15,.05,.05,.85)
X_2 = c(.1,.8,.95,.1,.25,.5,.25)
Y = c(-1, -1, 1, -1, 1, 1, -1)
X_b = cbind(X_1, X_2)
```

```
res = perceptron(X_b, Y, I = 100)
```

The preceptron made 8 mistakes before converging with accuracy = 1 (error = 0). See the decision boundary below.

```
suppressMessages(library(ggplot2))
ggplot(data = data.frame(X_1, X_2, Y), aes(x = X_1, y = X_2)) +
  geom_point(aes(pch = factor(Y)), size = 7) +
  scale_shape_manual(values = c("-", "+")) +
  geom_abline(slope = -res$w[["X_1"]]/res$w[["X_2"]], intercept = 0) +
  ggtitle("Result of Perceptron Algorithm",
          subtitle = "Other possible separators featured in color") +
  geom_abline(intercept = 0, slope = 1, color = "red", lty = "dashed") +
  geom_abline(intercept = 0, slope = 1.05, color = "blue", lty = "dashed") +
  theme_bw() +
  theme(plot.subtitle = element_text(color = "red"))
```



(b) Growing the decision tree.

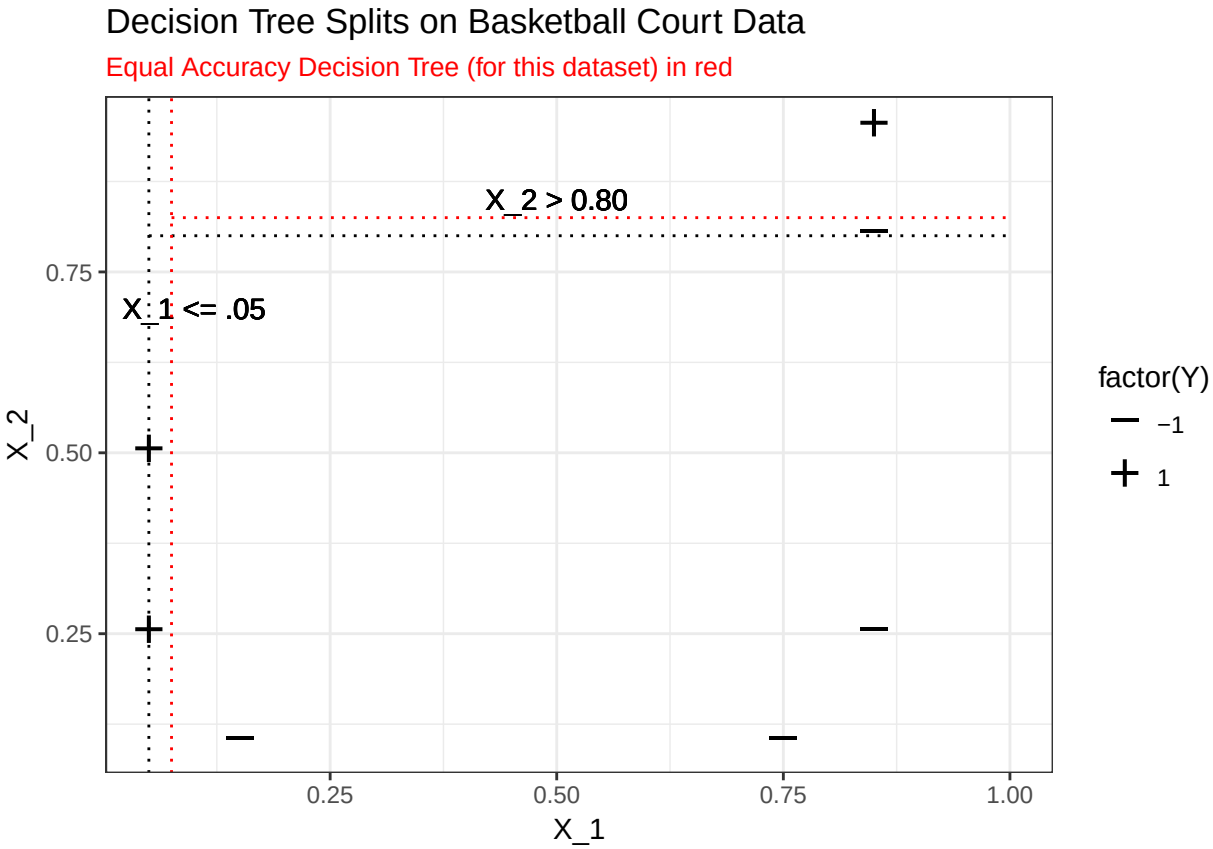
```
gini.index = function(node){ # function to calculate gini Index
  p = sum(node > 0)/length(node)
  I = 2*p*(1-p)
  return(I)
}
X = X_b
Y = Y
grow.tree = function(X, Y, current_node = NULL,
                     epsilon = 10^-6, threshold = 0.20){ # split node based on gini index
  X = data.frame(X)
  Y = data.frame(Y = Y)
  data.df = cbind(X,Y)
  if (is.null(current_node)){
    current_node = data.df
  }
  # variables to be filled later
  delta.I.record = c()
  leaf1.record = list()
  leaf2.record = list()
  split.record = list()
  c = 1
  for ( i in 1:(ncol(current_node)-1) ){
    # get possible split values, adjust min, max up, down to make inclusive inequalities
    # run without trouble
```

```
    split.sequence = sort(unique(current_node[,i]))
    split.sequence[1] = split.sequence[1] + epsilon
    split.sequence[length(split.sequence)] = split.sequence[length(split.sequence)] -
      epsilon
    for(x in split.sequence){ # this loop calculates each possible split and reduction in gini index
      split.record = append(split.record, list(c(i, x)))
      node_1 = current_node[which(current_node[,i] <= x),]
      leaf1.record[[c]] = node_1
      I_1 = gini.index(node = node_1$Y)
      node_2 = current_node[which(current_node[,i] > x),]
      leaf2.record[[c]] = node_2
      I_2 = gini.index(node = node_2$Y)
      delta.I = gini.index(current_node$Y) - sum(nrow(node_1)/nrow(current_node)*I_1,
                                                 nrow(node_2)/nrow(current_node)*I_2)
      delta.I.record = c(delta.I.record, delta.I)
      c = c + 1
    }
  }
  optimal = which.max(delta.I.record) # find split with greatest gini reduction
  new.nodes = list(leaf1.record[[optimal]], leaf2.record[[optimal]])
  Split = split.record[[optimal]]
  return(list("opt" = delta.I.record[optimal], "leaves" = new.nodes, "split" = Split,
              "threshold" = delta.I.record[optimal] > threshold))
}
```

The function defined above splits a single node based on the gini index. We implement it below, step by step, until we have grown a full tree.

```
tree_1 = grow.tree(X,Y)
pure_1 = length(unique(tree_1$leaves[[1]]$y)) == 1
pure_2 = length(unique(tree_1$leaves[[2]]$y)) == 1
split_1 = tree_1$split
# we see that the second leaf is not pure, so we attempt to split it again
tree_2 = grow.tree(X, Y, current_node = tree_1$leaves[[2]])
pure_1 = length(unique(tree_2$leaves[[1]]$y)) == 1
pure_2 = length(unique(tree_2$leaves[[2]]$y)) == 1
split_2 = tree_2$split
# all nodes are now pure; we should stop
```

We can see the splits in this lovely graphic.



The error on this tree is 0. We should note that if we set the ΔI threshold at or above 0.27, then there are not splits in the tree; we get only one class.

(c) The optimal linear classfier

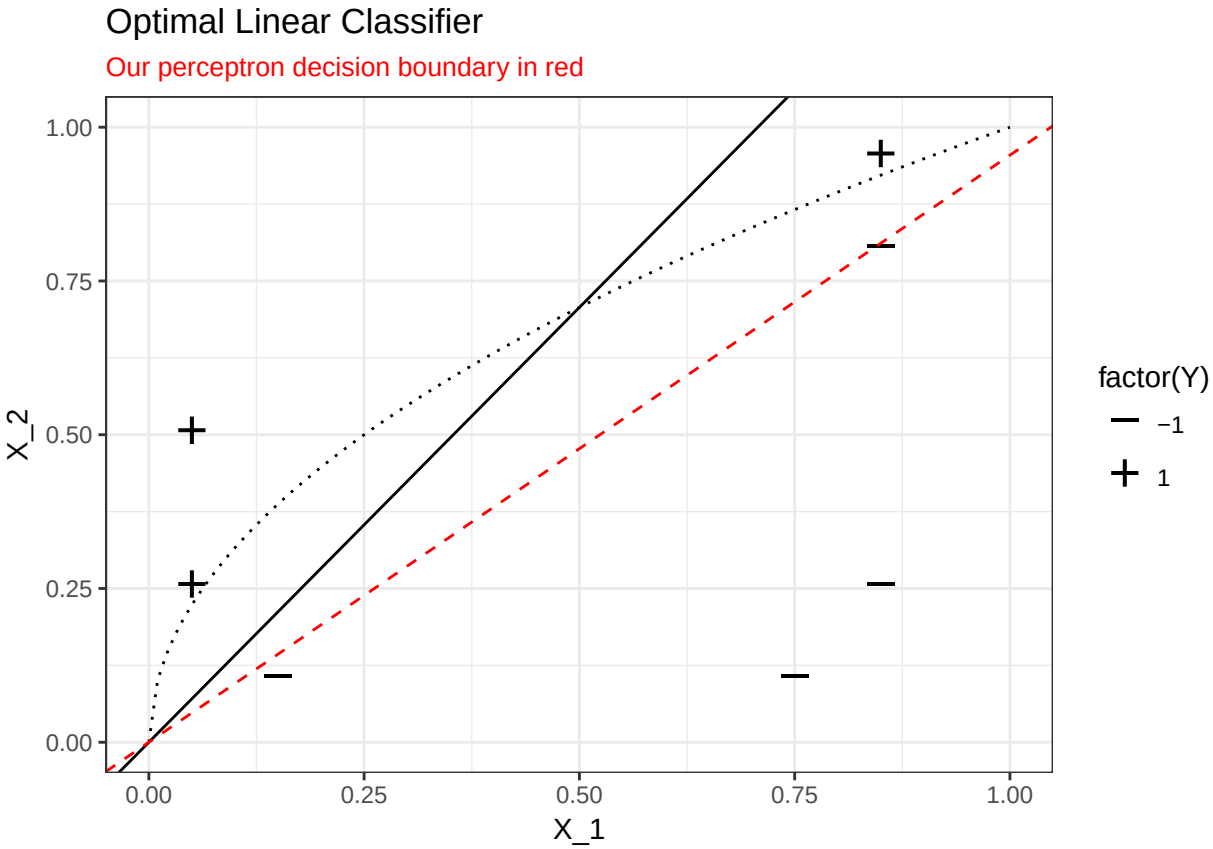
If we want to minimize the expected loss, we need to minimize the area between the true boundary, $\sqrt{x_1}$ and the decision boundary $w \cdot x_1$. Therefore, we compute the following integral.

$$\begin{aligned} h(w) &= \int_0^1 \min(|\sqrt{x_1} - wx|, 1 - \sqrt{x_1}) dx_1 \\ &= \int_0^{1/w^2} (\sqrt{x_1} - wx_1) dx_1 + \int_{1/w^2}^{1/w} (wx_1 - \sqrt{x_1}) dx_1 + \int_{1/w}^1 (1 - \sqrt{x_1}) dx_1 \\ &= \frac{1}{3} w^{-3} - \frac{1}{2} w^{-1} + \frac{1}{3} \end{aligned}$$

To get the minimum, we take the derivative, set that baby equal to zero, and crank the algebra machine until we get:

$$\begin{aligned} 0 = h'(w) &= -w^{-4} + \frac{1}{2} w^{-2} \\ \implies w &= 2^{1/2} = 1.414 \end{aligned}$$

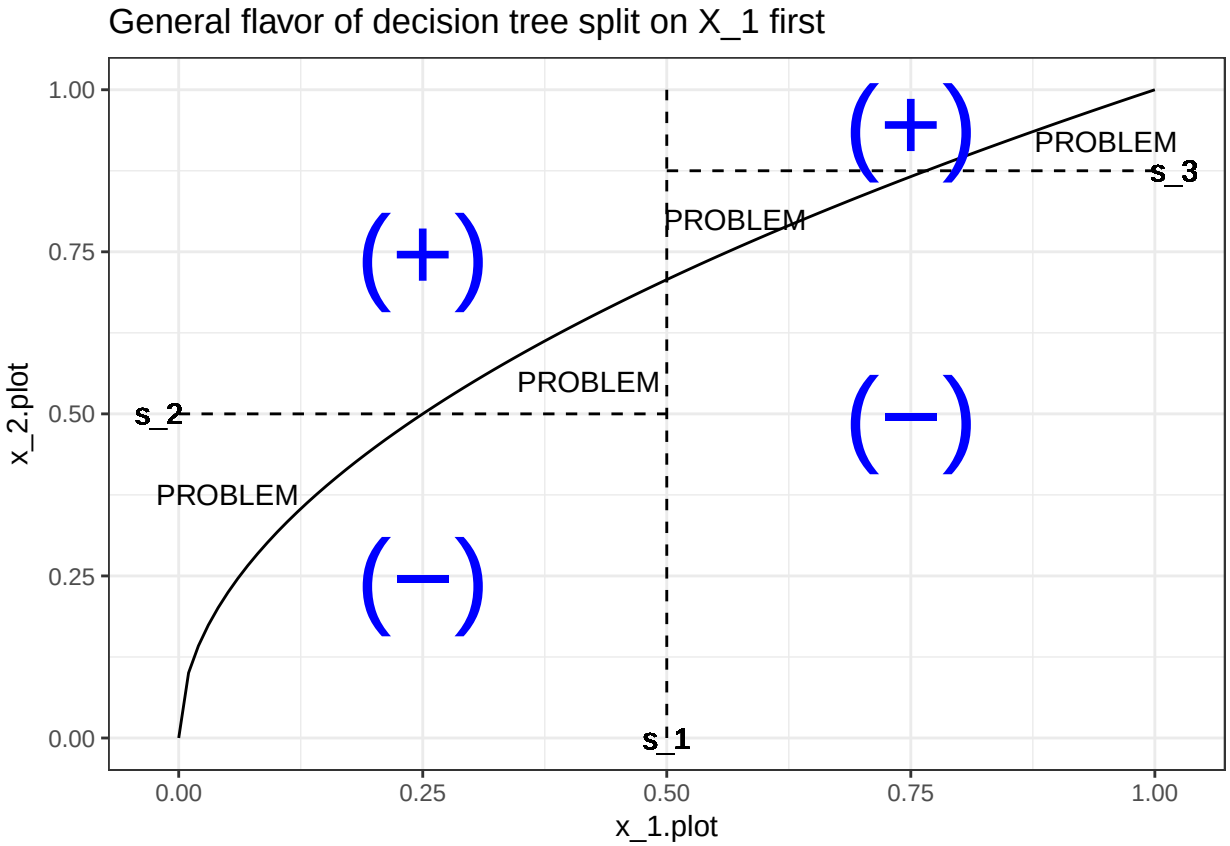
We plot the optimal decision boundary here.



We note that this classifier has lower accuracy in than the perceptron separator we found in a previous part. This is not surprising; the optimal classifier is the one that should perform best on average over a set of datasets, whereas our perceptron trained on one dataset specifically.

(d) The optimal decision tree

The general idea behind our approach here is the same as in optimizing the linear classifier. Since we are assuming the location of shots to be uniformly distributed, we want to minimize the misclassified area of the court. We begin with a visulalization. A general decision tree that splits on X_1 and then X_2 looks something like this.



Here, s_1 is the x_1 intercept of the vertical split, and s_2, s_3 are the x_2 intercepts of the left and right horizontal splits, respectively. Classification of each region is given by the obnoxiously large positive and negative signs. The ares that would be misclassified have a large 'PROBLEM' label.

We can calculate this area as follows:

$$\begin{aligned} h(s_1, s_2, s_3) &= \int_0^{s_1} |\sqrt{x_1} - s_2| dx_1 + \int_{s_1}^1 |\sqrt{x_1} - s_3| dx_1 \\ &= \int_0^{s_2^2} (s_2 - \sqrt{x_1}) dx_1 + \int_{s_2^2}^{s_1} (\sqrt{x_1} - s_2) dx_1 \\ &\quad + \int_{s_1}^{s_3^2} (s_3 - \sqrt{x_1}) dx_1 + \int_{s_3^2}^1 (\sqrt{x_1} - s_3) dx_1 \\ &\text{...some algebra later...} \\ &= \frac{2}{3}s_2^3 + \frac{2}{3}s_3^3 + \frac{4}{3}s_1^{3/2} - s_2 s_1 - s_3 s_1 - s_3 + \frac{2}{3} \end{aligned}$$

Now, we could take some partial derivatives, set things equal to zero, do more algebra, pray for death, and get an honest to (your preferred deity here) expression for these variables. Or, in less than one second, we run the following code chunk and get an approximation.

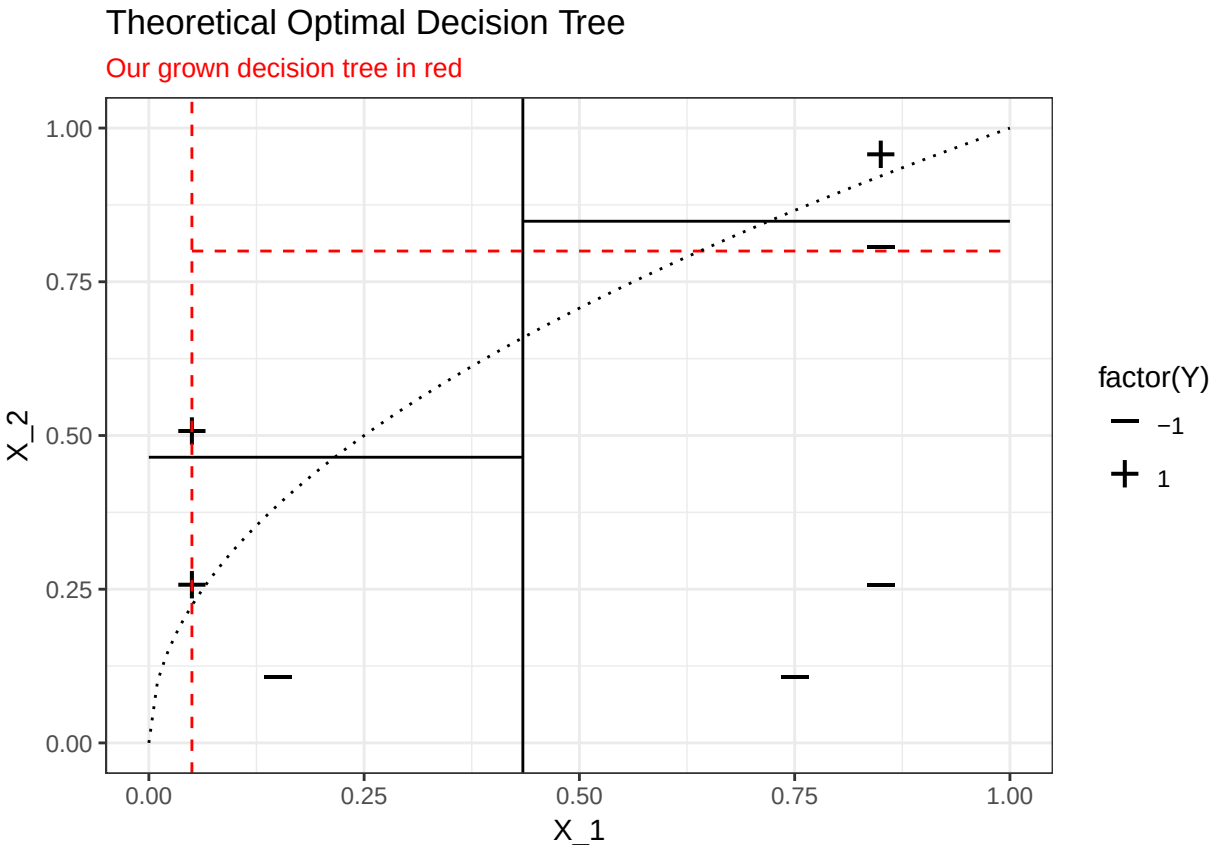
```
s1 = seq(0,1, length.out = 100)
s2 = seq(0,1, length.out = 100)
s3 = seq(0,1, length.out = 100)
split.exhaustive = matrix(NA, nrow = 100^3, ncol = 4)
i = 1
for (x in s1){
```

```
  for (y in s2){
    for (z in s3){
      split.exhaustive[i, 1] = x
      split.exhaustive[i, 2] = y
      split.exhaustive[i, 3] = z
      split.exhaustive[i, 4] = (2/3)*y^3 + (2/3)*z^3 + (4/3)*x^(3/2) -y*x -z*x - z + 2/3
      i = i + 1
    }
  }
}
minimum = which.min(split.exhaustive[,4])
s.opt = split.exhaustive[minimum, 1:3]
```

Table 1: Optimal Splitting Values

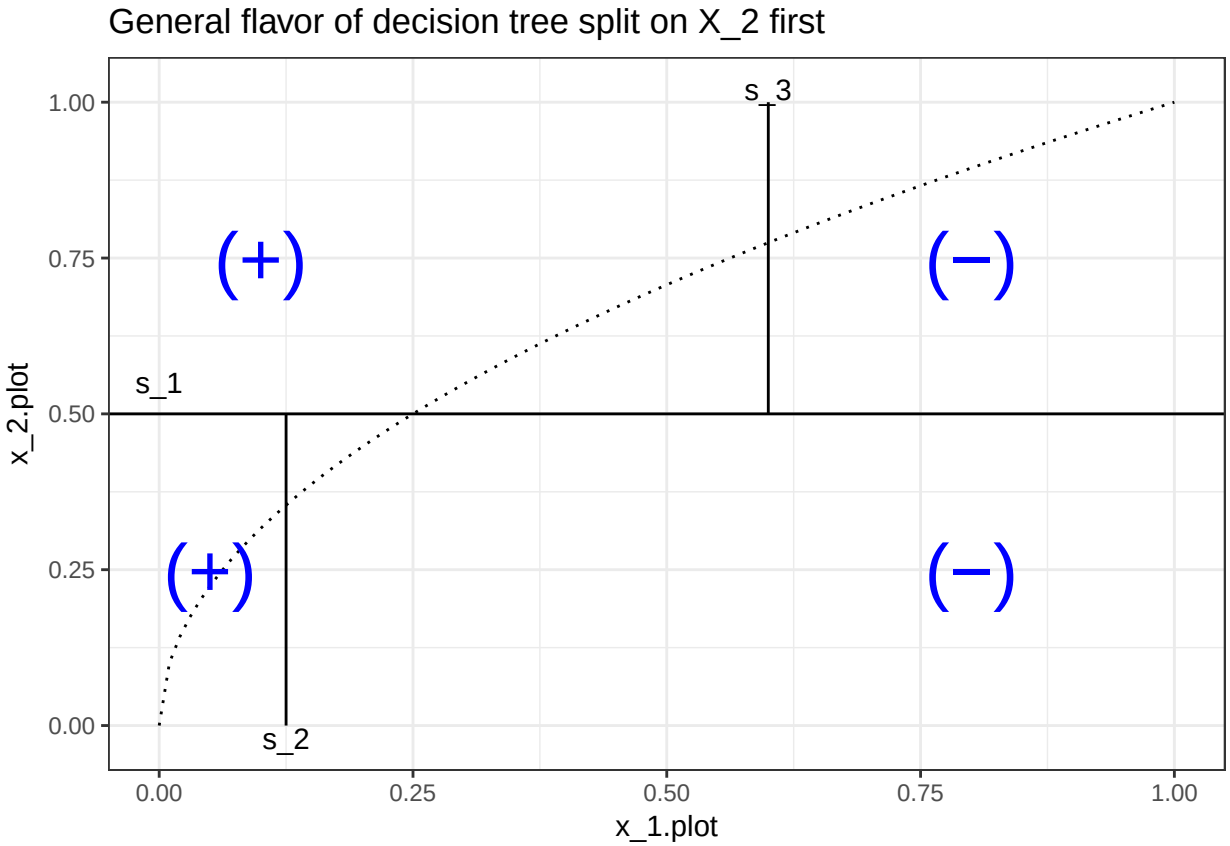
s_1	s_2	s_3
0.43	0.46	0.85

Please enjoy the following graphic. Signed copies available upon request.



Once again, we see that the accuracy is higher on our grown tree for this particular dataset. The theoretically optimal tree only has lower expected loss on average.

Now, we move on to the case of splitting on X_2 first. The general case looks something like this:



We will go through the same process as before to obtain an expression for the expected loss.

$$h(s_1, s_2, s_3) = \int_0^{s_2} \sqrt{x_1} dx_1 + \int_{s_2}^{s_1^2} (s_1 - \sqrt{x_1}) dx_1 + \int_{s_1}^{s_3} (\sqrt{x_1} - s_1) dx_1 + \int_{s_3}^{1} (1 - \sqrt{x_1}) dx_1$$

...some algegra later...

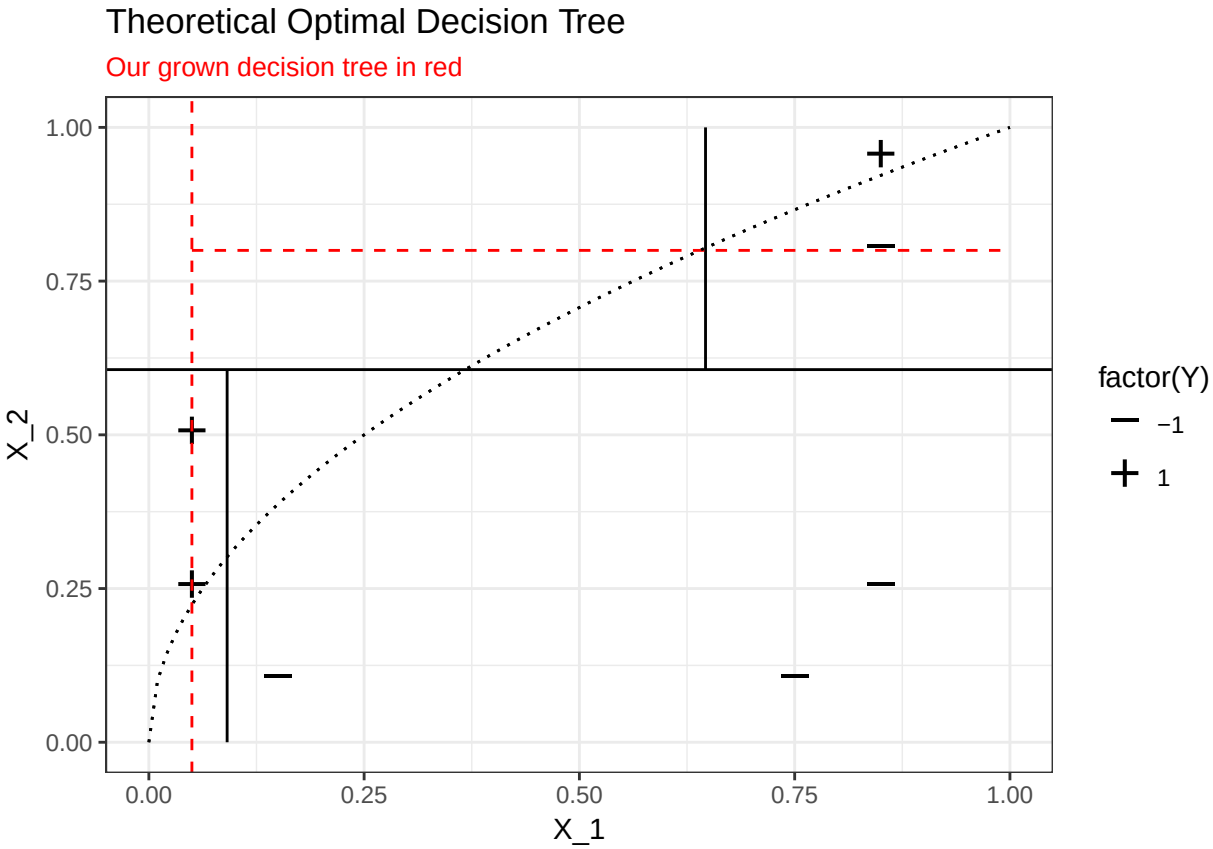
$$= \frac{4}{3} s_2^{3/2} + \frac{4}{3} s_3^{3/2} + \frac{1}{3} s_1^3 - s_1 s_2 - s_1 s_3 - s_3 + \frac{1}{3}$$

Again, we will compute the minimum of this function with this lovely computing machine exactly as above to obtain the following values for s_1, s_2, s_3.

Table 2: Optimal values for tree split on X_2 first

s_1	s_2	s_3
0.61	0.09	0.65

Visually, it looks something like this.

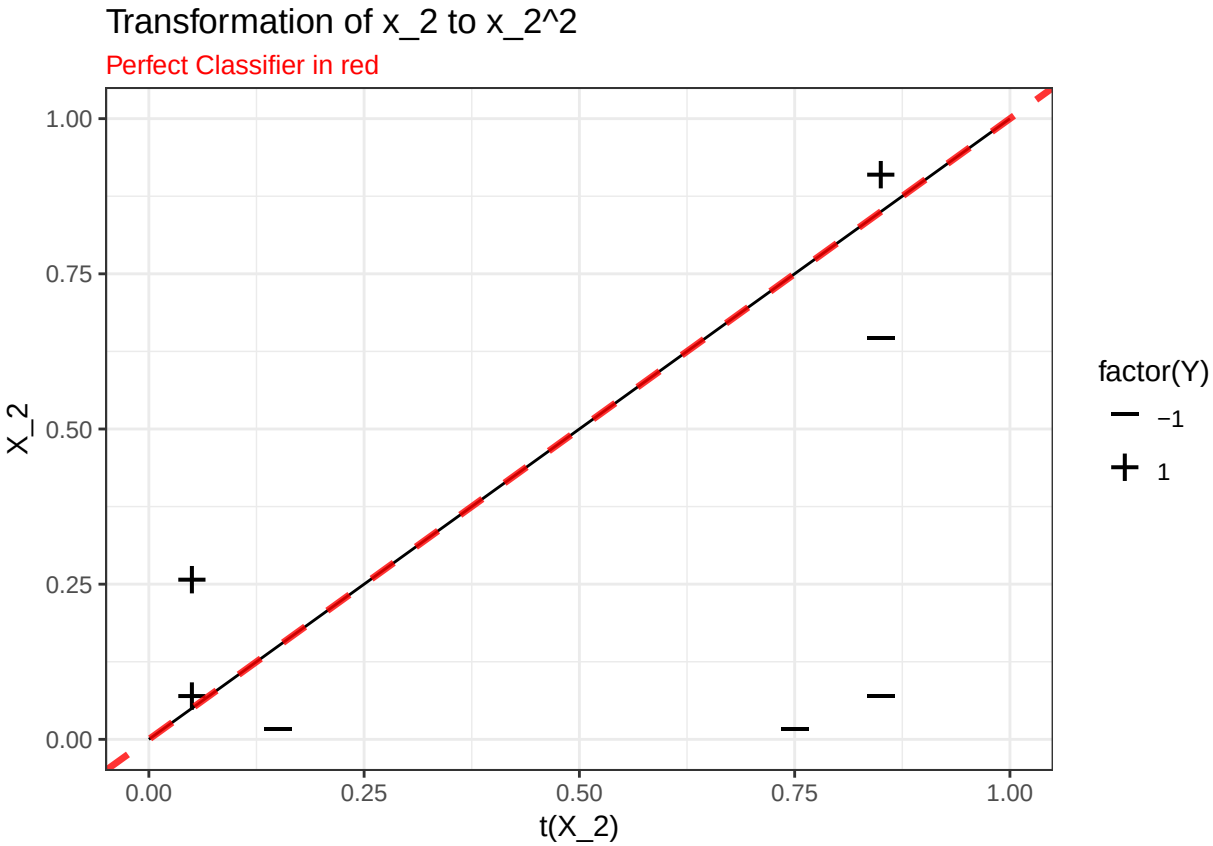


Once again, the accuracy is lower for the optimal loss tree on this specific example. We are still not troubled by this for the reasons explained above.

NOTE: In the above integrals, we have assumed each time that $s_2 < s_3$. We are comfortable making this decision since moving the s_2 and s_3 lines past each other always results in more misclassified area. Namely, an arrangement such as this will have the s_2 or s_3 line intersect the s_1 line without intersecting the true decision boundary. This will yield an additional rectangle of misclassified area that need not exist.

(e) Transformations of Covariates

Well, this one seems fairly tame after the algebra monstrosity of part (d). We know the true decision boundary to be $x_2 = \sqrt{x_1}$, so $x_2^2 = x_1$, and we consider the transformation, $t : x_2 \to x_2^2$.



Using the recognizing straight lines property, we deduce that the optimal classifier is the line of slope 1 passing through the origin under the transformation $t : x_2 \to x_2^2$. This classifier will always have zero error (perfect accuracy) because it is the same as the decision boundary.

(f) Decision tree under transformation

We could not obtain the same error under this transformation, especially not a depth 2 decision tree. A tree will produce a finite number of rectangular partitions of the space which are unable to capture a diagonal line. One could attempt an argument for 0 expected loss as the number of terminal nodes $c \to \infty$, but this is unrealistic.

(h) Classifying the paint (linear)

We will use the same integration technique as above. We consider below the line as inside the paint and above the line as outside the paint. We want to minimize the area

$$h(w) = \int_0^{.5} wx_1 dx_1 + \int_{.5}^{.25/w} (.25 - wx_1) dx_1 + \int_{.25/w}^{1} (wx_1 - .25) dx_1$$

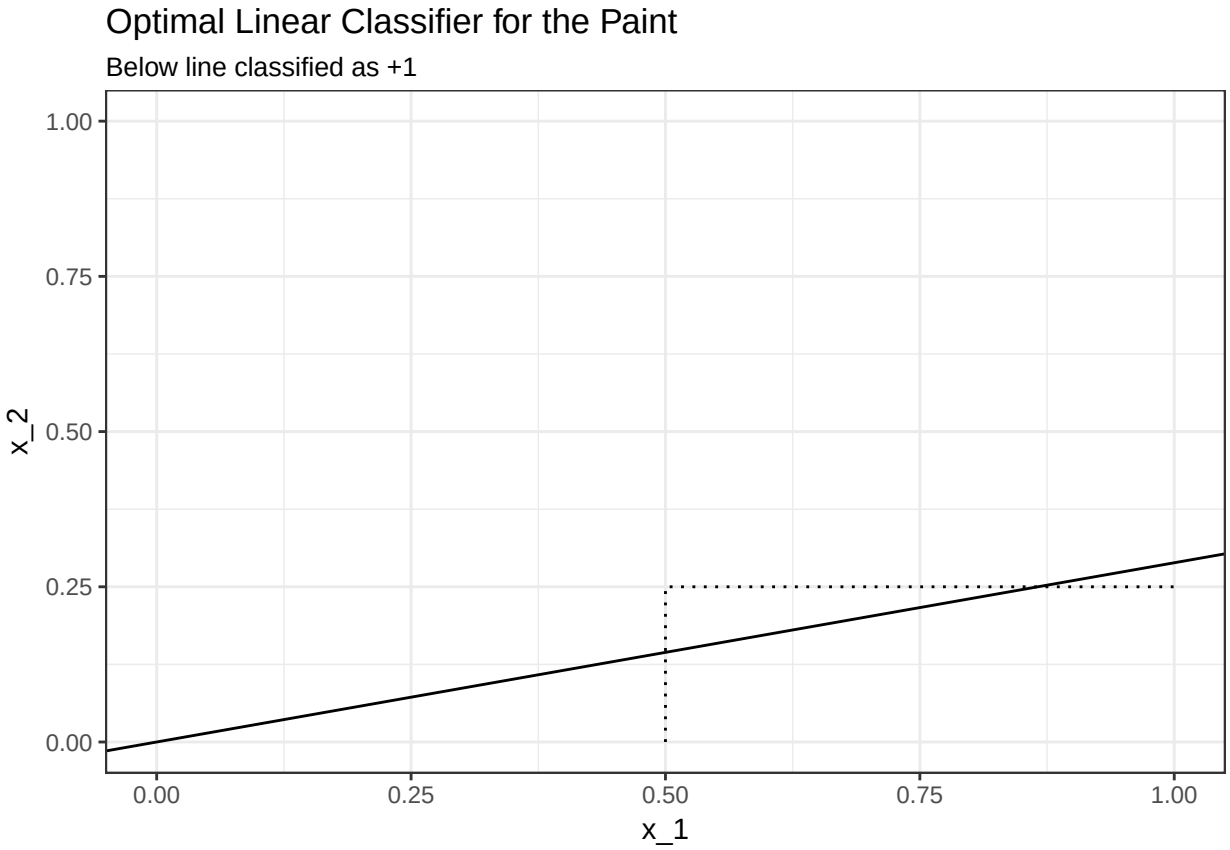
...some algebra later...

$$= \frac{1}{16} w^{-1} + \frac{1}{4} w + \frac{3}{8}$$

Setting $h'(w) = 0$, we get

$$w = \sqrt{\frac{1}{12}}$$

Which looks like



The 'true risk' of this classfier is 0.06.

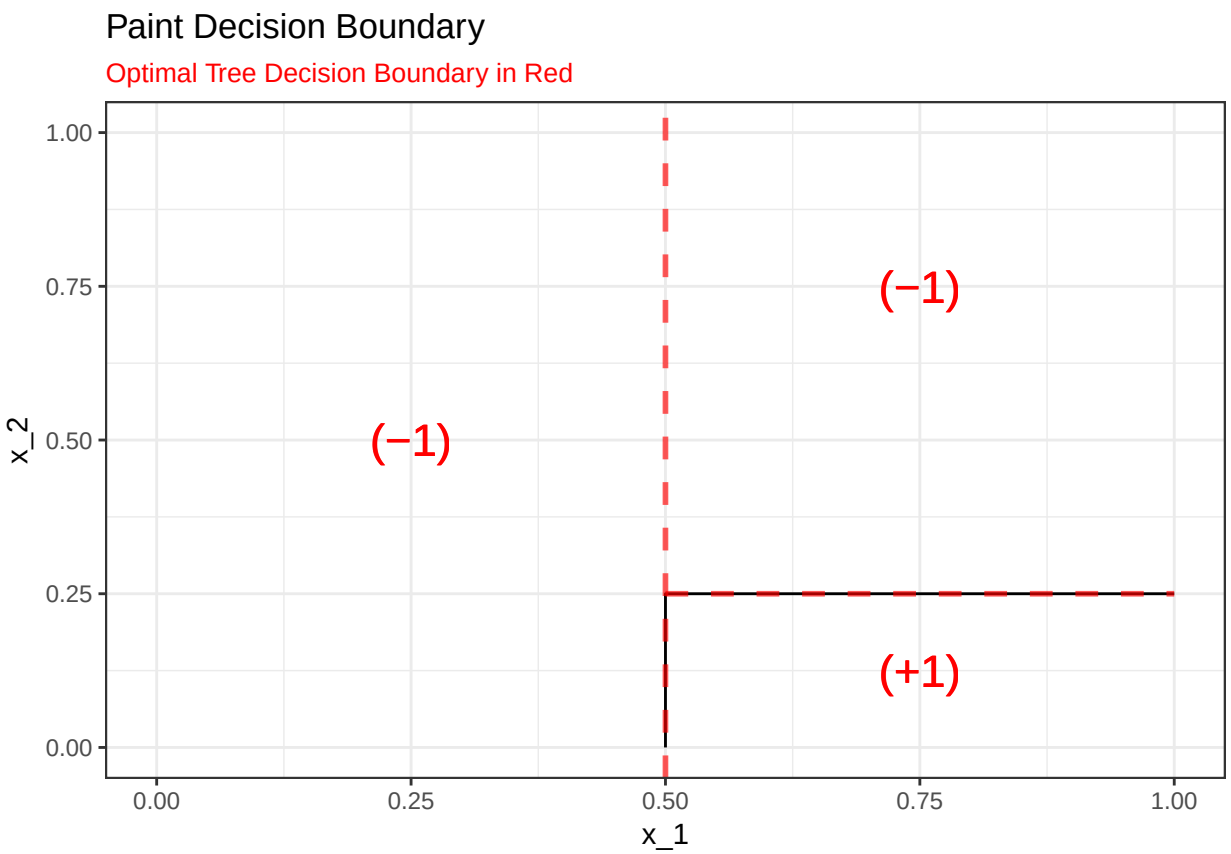
(i) Classifying the paint (tree)

Fortunately, the structure of the decision tree decision boundary makes this obvious.

$$f(x_1, x_2) = \begin{cases} -1 & x_1 \leq 0.5 \\ -1 & (x_1, x_2) \in (0.5, 1] \times (0.25, 1] \\ 1 & (x_1, x_2) \in (0.5, 1] \times [0, 0.25] \end{cases}$$

The decision boundary is as follows.

```
ggplot(data = data.frame(x = c(.5,.5,1), y =c(0,.25,.25) ), aes(x,y)) +
  geom_line() +
  xlim(0,1) +
  ylim(0,1) +
  ggtitle("Paint Decision Boundary",
          "Optimal Tree Decision Boundary in Red") +
  geom_vline(xintercept = 0.5, color = "red", lty = "dashed", size = 1, alpha = .65) +
  geom_line(data = data.frame(x_1 = c(0.5, 1), x_2 = .25), aes(x_1, x_2),
            color = "red", lty = "dashed", size = 1, alpha = .65) +
  theme_bw() +
  theme(plot.subtitle = element_text(color = "red")) +
  geom_text(aes(.75, .125, label = "(+1)"), size = 6, color = "red") +
  geom_text(aes(.75, .75, label = "(-1)"), size = 6, color = "red") +
  geom_text(aes(.25, .5, label = "(-1)"), size = 6, color = "red") +
  ylab("x_2") + xlab("x_1")
```



The error is clearly 0.

---

(2) Variable Importance for Trees and Forests

Hurrah! CSV files!

```
train = read.csv("train.csv")
test = read.csv("test.csv")
```

(a) A Single Stump

Fine, I'll write my own. It's going to be specific to this prblem though. I'll work on the generality later. The Gini Index function here will be different that question (1). It will accept data frames rather than a vector for the sake of avoiding syntax problems and maintaining sanity.

```
gini.index = function(node.df){ # response should be called Y, always {0,1}
  p = sum(node.df$Y)/nrow(node.df)
  I = 2*p*(1-p)
  return(I)
}
grow.stump = function(df, surrogate = TRUE){ # df = data frame of response AND covariates
  N = nrow(df)
  I_df = gini.index(df) # get the gini index value of the full data
  predictors = names(df)[names(df) != "Y"]
  c = 1
```

```
  leaf1.record = NULL # new strategy for storing results
  leaf2.record = NULL
  delta.I.record = -Inf
  for (j in predictors){ # no need for inner loop since all covariates are binary
    predictor.is.1 = which(df[,j] == 1)
    predictor.is.0 = -predictor.is.1
    leaf1 = df[predictor.is.0,]
    leaf2 = df[predictor.is.1,]
    class1 = 1*(sum(leaf1$Y > 0) > sum(leaf1$Y <= 0))
    class2 = 1*(sum(leaf2$Y > 0) > sum(leaf2$Y <= 0))
    I_1 = gini.index(leaf1)
    I_2 = gini.index(leaf2)
    N_1 = nrow(leaf1)
    N_2 = nrow(leaf2)
    delta.I = I_df - sum( (N_1/N)*I_1, (N_2/N)*I_2 )
    if (delta.I > delta.I.record){
      delta.I.record = delta.I
      leaf1.record = list("leaf" = leaf1, "split" = j, "class" = class1)
      leaf2.record = list("leaf" = leaf2, "split" = j, "class" = class2)
    }
  }
  surrogateStump = NULL
  if (surrogate){
    j = actualBest = leaf1.record$split
    surrogateOptions = predictors[!(predictors %in% actualBest)]
    lambda = rep(NA, length(surrogateOptions))
    l = 1
    for (j.tilde in surrogateOptions){
      p_L = nrow(leaf1.record$leaf)/N
      p_R = nrow(leaf1.record$leaf)/N
      p_min = min(p_L, p_R)
      p_LL = sum(df[,j] == 0 & df[,j.tilde] == 0)/N
      p_RR = sum(df[,j] == 1 & df[,j.tilde] == 1)/N
      p_minmin = 1 - p_LL - p_RR
      lambda[l] = (p_min - p_minmin)/p_min
      l = l+1
    }
    bestSurrogate = surrogateOptions[which.max(lambda)]
    surrogateSplit.1 = which(df[,bestSurrogate] == 1)
    surrogateSplit.0 = -surrogateSplit.1
    surrogateLeaf1 = df[surrogateSplit.0,]
    surrogateLeaf2 = df[surrogateSplit.1,]
    class1 = 1*(sum(surrogateLeaf1$Y > 0) > sum(surrogateLeaf1$Y <= 0))
    class2 = 1*(sum(surrogateLeaf2$Y > 0) > sum(surrogateLeaf2$Y <= 0))
    deltaSurrogate = I_df - sum(nrow(surrogateLeaf1)/N*gini.index(surrogateLeaf1),
                                nrow(surrogateLeaf2)/N*gini.index(surrogateLeaf2))
    surrogateStump = list("leaf1" = surrogateLeaf1,
                          "leaf2" = surrogateLeaf2,
                          "split" = bestSurrogate,
                          "class1" = class1, "class2" = class2,
                          "delta" = deltaSurrogate)
  }
  return(list("leaf1" = leaf1.record, "leaf2" = leaf2.record, "deltaI" = delta.I.record,
```

```
              "surrogate" = surrogateStump))
}
```

(a)(i) Best and Surrogate Splits

```
stump1 = grow.stump(train)
cat("Best split:", stump1$leaf1$split,
    "\n with '0' leaf predicting:", stump1$leaf1$class,
    "\n and '1' leaf predicting:", stump1$leaf2$class)
```

```
## Best split: X1
##  with '0' leaf predicting: 0
##  and '1' leaf predicting: 1
```

```
cat("\nSurrogate Split:", stump1$surrogate$split,
    "\n with '0' leaf predicting:", stump1$surrogate$class1,
    "\n and '1' leaf predicting:", stump1$surrogate$class2)
```

```
##
## Surrogate Split: X2
##  with '0' leaf predicting: 0
##  and '1' leaf predicting: 1
```

For predictors $(X_1, \ldots, X_5)$, let f denote the stump resulting from the best split and $\tilde{f}$ denote the stump resulting from the surrogate split.

$$f(X_1, \ldots, X_5) = \begin{cases} 0 & X_1 = 0 \\ 1 & X_1 = 1 \end{cases} \tilde{f}(X_1, \ldots, X_5) = \begin{cases} 0 & X_2 = 0 \\ 1 & X_2 = 1 \end{cases}$$

(a)(ii) Variable Importance

Time to plug and chug. The best split is on X_1, and there is only one split in our stump, so we have only one value to compute, namely the importance for X_1.

$$\begin{aligned} \text{Imp}^T(X_1) &= \Delta I(0, 1, 1) \\ &= \text{reduction in Gini Index at split} \\ &= 0.270 \end{aligned}$$

To be honest, I don't know how to interpret this number. I'm not sure what it's range is or what is considered a large value. Relative to the surrogate split, which has importance

$$\text{Imp}^T(X_2) = 0.106,$$

it seems that X_1 is more important to the classification. How to compare these values is not immediately obvious to me. One way I have seen to compare these values is to divide them all by the magnitude of the largest, in which case we would get a 'relative' importance like so.

$$\hat{Imp}^T(X_1) = 1, \quad \hat{Imp}^T(X_2) = 0.393$$

With this measure, I feel more comfortable asserting that, yes, X_1 is more important than X_2, by a factor of about 2.5.

(a)(iii) MSE

```
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
```

```
test.MSE1 = test %>%
  mutate(yhat1 = ifelse(X1 == 0, 0, 1)) %>%
  mutate(error1 = (yhat1 - Y)^2) %>%
  mutate(yhat2 = ifelse(X2 == 0, 0, 1)) %>%
  mutate(error2 = (yhat2 - Y)^2)
MSE1 = mean(test.MSE1$error1)
MSE2 = mean(test.MSE1$error2)
knitr::kable(data.frame(MSE1, MSE2), col.names = c("MSE for X_1 split", "MSE for X_2 split"),
             caption = "Test MSEfor Best (X_1) and Surrogate (X_2) Split")
```

Table 3: Test MSEfor Best (X_1) and Surrogate (X_2) Split

MSE for X_1 split	MSE for X_2 split
0.1	0.27

(b) Growing a Forest (of stumps)

```
n = nrow(train)
B = 0.8 * n
M = 1000
K = 5
res = data.frame(leaf0 = rep(NA, M), split = rep(NA, M), leaf1 = rep(NA, M), surrogate = rep(NA, M),
                 delta = rep(NA, M), OOBerror = rep(NA, M), OOBperm = rep(NA, M))
forest = list(res,
              res,
              res,
              res,
              res)
for (k in 1:K){
  for (m in 1:M){
    bootstrap = sample(1:n, B, replace = TRUE)
    df.boot = train[bootstrap,]
    df.oob = train[-unique(bootstrap),]
    predictors = names(df.boot)[names(df.boot) != "Y"]
    df.boot = df.boot %>%
      select(sample(predictors, k), Y)
    stump = grow.stump(df = df.boot, surrogate = ifelse(k==1, FALSE, TRUE))
    forest[[k]]$leaf0[m] = stump$leaf1$class
    forest[[k]]$split[m] = stump$leaf1$split
    forest[[k]]$leaf1[m] = stump$leaf2$class
```

```
    if (k != 1){
      forest[[k]]$surrogate[m] = stump$surrogate$split
    }
    forest[[k]]$delta[m] = stump$deltaI
    df.oob = df.oob %>%
      mutate(yhat = ifelse(.[,stump$leaf1$split] == 0,
                           stump$leaf1$class,
                           stump$leaf2$class))
    forest[[k]]$OOBerror[m] = sum(df.oob$yhat != df.oob$Y)/nrow(df.oob)
    df.oob[,stump$leaf1$split] = sample(df.oob[,stump$leaf1$split])
    df.oob = df.oob %>%
      mutate(yhat = ifelse(.[,stump$leaf1$split] == 0,
                           stump$leaf1$class,
                           stump$leaf2$class))
    forest[[k]]$OOBperm[m] = sum(df.oob$yhat != df.oob$Y)/nrow(df.oob)
  }
}
```

Yes, the above would run faster if we did it in parallel, one core for each `k`, but it would take longer to write and debug that code than it does to run the code in its current state. User efficiency beats machine efficiency this time.

(b)(i) Basic Summary of Forest

Asking 'how many times was each variable the best split?' is not relevant when there are only $k = 1$ choices for variables, so we omit that case here. A same triviality with with the surrogate splits for $k = 2$, but we do not omit this case.

Table 4: Occurences of Each Variable as Best or Surrogate Split

	X1 best	X2 best	X3 best	X4 best	X5 best	X1 surrogate	X2 surrogate	X3 surrogate	X4 surrogate	X
k=2	384	338	111	96	71	0	97	309	304	
k=3	591	307	37	40	25	0	283	244	183	
k=4	787	213	0	0	0	0	566	153	76	
k=5	1000	0	0	0	0	0	1000	0	0	

We see from the table that X_1 is overwhelmingly chosen to be the best variable for splitting; when all variables are available for splitting, X_1 appears in 100% of cases with X_2 always as its surrogate. This fact is significant considering that each tree is grown from a different bootstrap sample. The numbers we see here suggest that X_1 is the most important variable, followed by X_2.

The effect of K is obvious here. The same stump growing algorithm makes every stump, so it is likely to choose the same variable to split each time. Only when that variable is unavailable does it explore the other options fully.

(b)(ii) Variable Importance by OOB Error

Table 5: Variable Importance by Sum of Importance Across Trees

	X1	X2	X3	X4	X5
k=1	0.0537	0.0216	3e-04	3e-04	3e-04
k=2	0.1030	0.0359	3e-04	3e-04	2e-04

	X1	X2	X3	X4	X5
k=3	0.1585	0.0322	2e-04	2e-04	1e-04
k=4	0.2119	0.0224	0e+00	0e+00	0e+00
k=5	0.2697	0.0000	0e+00	0e+00	0e+00

Table 6: Variable Importance by Out-of-Bag Error

	X1	X2	X3	X4	X5
k=1	0.0735	0.0468	-5e-04	3e-04	-0.0021
k=2	0.1413	0.0767	-8e-04	-7e-04	-0.0010
k=3	0.2168	0.0717	-5e-04	-4e-04	-0.0003
k=4	0.2904	0.0490	0e+00	0e+00	0.0000
k=5	0.3678	0.0000	0e+00	0e+00	0.0000

Based on the tables we see above, X_1 appears to be the most important variable in every case, followed by X_2. This trait is more obvious in stumps that are allowed to split on X_1 more often (as K increases). Since the same algorithm is making the splits each time, and these are only stumps, it stands to reason that an important variable will be chosen for a split when allowed. The bootstrapping, in this case where X_1 is so much more important, does not make a difference.

Intuitively, the impact of masking should be lower in a forest. Each tree is grown on a random subset of predictors, so a dominant predictor cannot mask the others in every tree. It will not even be an option for splitting in some trees. Therefore, the other predictors will be used for splits and their importance appears within the importance measure for the forest as whole.

(b)(iii) MSE in the Forest

```
forest.vote = function(k, x.df){
  forest = forest[[k]]
  yhat = rep(0, nrow(x.df))
  for (i in 1:nrow(forest)){
    for (j in 1:nrow(x.df)){
      var.of.interest = forest$split[i]
      yhat.vote = ifelse(x.df[j,var.of.interest]==0,
                         forest$leaf0[i],
                         forest$leaf1[i])
      yhat[j] = yhat[j] + (yhat.vote - .5)
    }
  }
  final_vote = 1*(yhat > 0)
  return(final_vote)
}
trees.ind.class = function(k, test.df){
  Y = test.df$Y
  forest = forest[[k]]
  loss = rep(NA, nrow(forest))
  for (i in 1:nrow(forest)){
    error = rep(NA, nrow(test.df))
    for (j in 1:nrow(test.df)){
      var.of.interest = forest$split[i]
      yhat = ifelse(test.df[j,var.of.interest]==0,
```

```
                          forest$leaf0[i],
                          forest$leaf1[i])
      error[j] = (Y[j] - yhat)^2
    }
    loss[i] = mean(error)
  }
  return(mean(loss))
}
test.x = test[1:(ncol(test) - 1)]
test.df = test %>%
  mutate(yhat.vote1 = forest.vote(k = 1, test.x)) %>%
  mutate(yhat.vote2 = forest.vote(k = 2, test.x)) %>%
  mutate(yhat.vote3 = forest.vote(k = 3, test.x)) %>%
  mutate(yhat.vote4 = forest.vote(k = 4, test.x)) %>%
  mutate(yhat.vote5 = forest.vote(k = 5, test.x)) %>%
  mutate(error1 = (Y-yhat.vote1)^2) %>%
  mutate(error2 = (Y-yhat.vote2)^2) %>%
  mutate(error3 = (Y-yhat.vote3)^2) %>%
  mutate(error4 = (Y-yhat.vote4)^2) %>%
  mutate(error5 = (Y-yhat.vote5)^2)
MSE.forest = data.frame(vote = rep(NA, 5), ind = rep(NA, 5))
for (i in 1:5){
  MSE.forest[i, "vote"] = mean(test.df[,(i+11)])
  MSE.forest[i, "ind"] = trees.ind.class(k = i, test.df = test)
}
rownames(MSE.forest) = rownames(table.df.imp)
knitr::kable(MSE.forest, col.names = c("Loss by Vote", "The Weird Loss"),
             caption = "Comparison of Loss Measures")
```

Table 7: Comparison of Loss Measures

	Loss by Vote	The Weird Loss
k=1	0.14	0.37571
k=2	0.14	0.26812
k=3	0.10	0.19233
k=4	0.10	0.13621
k=5	0.10	0.10000

The voting method is the proper method for calculating loss since that's how random forests actually make classifications of new data. We want our estimate of loss to come from the process actually used in classification.

It seems that both types of error decrease when K increases. Intuitively, we understand this to mean that if the stumps are allowed to split on the most important variable more often, then they will have lower error. We must remember, however, that this can lead to overfitting, especially when we grow deeper trees.

(c) Another Forest

```
q = seq(.4,.8, by = .1)
n = nrow(train)
B_n = q*n
K = 2
```

```
M = 1000
res = data.frame( split = rep(NA, M), surrogate = rep(NA, M),
                  delta = rep(NA, M), OOBerror = rep(NA, M), OOBperm = rep(NA, M))
forest2 = list(res,
               res,
               res,
               res,
               res)
k = 1
for (B in B_n){
  for (m in 1:M){
    bootstrap = sample(1:n, B, replace = TRUE)
    df.boot = train[bootstrap,]
    df.oob = train[-unique(bootstrap),]
    predictors = names(df.boot)[names(df.boot) != "Y"]
    df.boot = df.boot %>%
      select(sample(predictors, K), Y)
    stump = grow.stump(df = df.boot, surrogate = TRUE)
    # forest2[[k]]$leaf0[m] = stump$leaf1$class
    forest2[[k]]$split[m] = stump$leaf1$split
    # forest2[[k]]$leaf1[m] = stump$leaf2$class
    forest2[[k]]$surrogate[m] = stump$surrogate$split
    forest2[[k]]$delta[m] = stump$deltaI
    df.oob = df.oob %>%
      mutate(yhat = ifelse(.[,stump$leaf1$split] == 0,
                           stump$leaf1$class,
                           stump$leaf2$class))
    forest2[[k]]$OOBerror[m] = sum(df.oob$yhat != df.oob$Y)/nrow(df.oob)
    df.oob[,stump$leaf1$split] = sample(df.oob[,stump$leaf1$split])
    df.oob = df.oob %>%
      mutate(yhat = ifelse(.[,stump$leaf1$split] == 0,
                           stump$leaf1$class,
                           stump$leaf2$class))
    forest2[[k]]$OOBperm[m] = sum(df.oob$yhat != df.oob$Y)/nrow(df.oob)
  }
  k = k + 1
}
```

(c)(i) Importance, Different Bootstrap Sizes

Table 8: Variable Importance by Sum of Importance Across Trees

	X1	X2	X3	X4	X5
B=0.4n	0.1042	0.0340	5e-04	5e-04	5e-04
B=0.5n	0.1075	0.0344	5e-04	4e-04	3e-04
B=0.6n	0.1022	0.0317	4e-04	4e-04	3e-04
B=0.7n	0.1022	0.0331	3e-04	3e-04	2e-04
B=0.8n	0.1119	0.0312	3e-04	3e-04	2e-04

Table 9: Variable Importance by Out-of-Bag Error

	X1	X2	X3	X4	X5
B=0.4n	0.1419	0.0719	-6e-04	-7e-04	-0.0013
B=0.5n	0.1428	0.0705	-6e-04	-4e-04	-0.0007
B=0.6n	0.1401	0.0676	-7e-04	-5e-04	-0.0011
B=0.7n	0.1395	0.0705	1e-04	-6e-04	-0.0015
B=0.8n	0.1521	0.0685	-4e-04	-5e-04	-0.0012

We see here that as B increases, X_1 remains the most important, but the magnitude of its out-of-bag 'raw importance' decreases. The other variables tend to fluctuate and many are so small in magnitude that we cannot say much about changes in their raw importance. As to why the most important variable shows this inverse correlation with bootstrap sample size, I can only speculate. Please see below for speculation.

For smaller values of B, there is a lower probability of repeated sampling of any data point; it is closer to a sample without replacement. Essentially, the algorithm is training on a subset that is closer to the distribution of the full set, leading to lower values of $error_{OOB}(T_i, x_j)$ and therefore a larger value of $Imp_{OOB}(x_j)$ overall.

But, I look forward to hearing a correct explanation at some point.

(c)(ii) Standard Deviation of Importance

This feels like a strange thing to calculate.

Table 10: Standard Deviation of Variable Importance Across Trees

	X1	X2	X3	X4	X5
B=0.4n	0.0358	0.0279	0.0057	0.0042	0.0054
B=0.5n	0.0332	0.0279	0.0057	0.0047	0.0037
B=0.6n	0.0301	0.0227	0.0034	0.0039	0.0030
B=0.7n	0.0261	0.0212	0.0023	0.0029	0.0024
B=0.8n	0.0245	0.0194	0.0027	0.0026	0.0021

Table 11: Standard Deviation of Variable Importance by Out-of-Bag Error

	X1	X2	X3	X4	X5
B=0.4n	0.0288	0.0311	0.0328	0.0334	0.0262
B=0.5n	0.0315	0.0349	0.0295	0.0290	0.0326
B=0.6n	0.0339	0.0359	0.0312	0.0293	0.0296
B=0.7n	0.0354	0.0382	0.0266	0.0370	0.0340
B=0.8n	0.0375	0.0387	0.0345	0.0344	0.0330

Well, this time the decreasing trend appears within the variable imprtance, rather than the OOB importance. Perhaps with more data points on which to train (B larger) there is less uncertainty about which variable is important, with regards to impurity measure that is. As for while the OOB importance shows an increase in variability as B increases, well, the number of data points which fall 'out-of-bag' will be lower. With fewer data points, it makes intuitive sense that our uncertainty about the error would be higher. This fails to explain the decrease we see in X_1 importance standard deviation for $B = 0.8n$, but it's the best I've got right now.