STAT 471: Homework 4

Ashley Clarke

Due: November 17, 2021 at 11:59pm

Contents

Instructions

Setup

Pull the latest version of this assignment from Github and set your working directory to `stat-471-fall-2021/homework/homework-4`. Consult the getting started guide if you need to brush up on `R` or `Git`.

Collaboration

The collaboration policy is as stated on the Syllabus:

> "Students are permitted to work together on homework assignments, but solutions must be written up and submitted individually. Students must disclose any sources of assistance they received; furthermore, they are prohibited from verbatim copying from any source and from consulting solutions to problems that may be available online and/or from past iterations of the course."

In accordance with this policy,

Please list anyone you discussed this homework with: - Zach Bradlow, Sarah Hu, and Paul Heysch de la Borde

Please list what external references you consulted (e.g. articles, books, or websites): - https://towardsdatascience.com/understanding-decision-trees-once-and-for-all-2d891b1be579 - https://www.analyticsvidhya.com/blog/2020/06/4-ways-split-decision-tree/#related-articles

Writeup

Use this document as a starting point for your writeup, adding your solutions after "**Solution**". Add your R code using code chunks and add your text answers using **bold text**. Consult the preparing reports guide for guidance on compilation, creation of figures and tables, and presentation quality.

Programming

The `tidyverse` paradigm for data wrangling, manipulation, and visualization is strongly encouraged, but points will not be deducted for using base `R`.

Grading

The point value for each problem sub-part is indicated. Additionally, the presentation quality of the solution for each problem (as exemplified by the guidelines in Section 3 of the preparing reports guide will be evaluated on a per-problem basis (e.g. in this homework, there are three problems). There are 100 points possible on this homework, 83 of which are for correctness and 17 of which are for presentation.

Submission

Compile your writeup to PDF and submit to Gradescope.

Case Study: Spam Filtering

In this homework, we will be looking at data on spam filtering. Each observation corresponds to an email to George Forman, an employee at Hewlett Packard (HP) who helped compile the data in 1999. The response `spam` is 1 or 0 according to whether that email is spam or not, respectively. The 57 features are extracted from the text of the emails, and are described in the documentation for this data. Quoting from this documentation:

> There are 48 continuous real [0,100] attributes of type word_freq_WORD = percentage of words in the e-mail that match WORD, i.e. 100 * (number of times the WORD appears in the e-mail) / total number of words in e-mail. A "word" in this case is any string of alphanumeric characters bounded by non-alphanumeric characters or end-of-string.

There are 6 continuous real [0,100] attributes of type char_freq_CHAR = percentage of characters in the e-mail that match CHAR, i.e. 100 * (number of CHAR occurences) / total characters in e-mail.

There is 1 continuous real [1,...] attribute of type capital_run_length_average = average length of uninterrupted sequences of capital letters.

There is 1 continuous integer [1,...] attribute of type capital_run_length_longest = length of longest uninterrupted sequence of capital letters.

There is 1 continuous integer [1,...] attribute of type capital_run_length_total = sum of length of uninterrupted sequences of capital letters = total number of capital letters in the e-mail.

The goal is to build a spam filter, i.e. to classify whether an email is spam based on its text.

First, let's load a few libraries:

```
library(rpart)          # to train decision trees
library(rpart.plot)     # to plot decision trees
library(randomForest)   # random forests
library(gbm)            # boosting
library(tidyverse)      # tidyverse
library(dplyr)
library(kableExtra)
```

Next, let's load the data (first make sure spam_data.tsv is in your working directory):

```
spam_data = read_tsv("../../data/spam_data.tsv")
```

The data contain a test set indicator, which we filter on to create a train-test split.

```
# extract training data
spam_train = spam_data %>%
  filter(test == 0) %>%
  select(-test)

# extract test data
spam_test = spam_data %>%
  filter(test == 1) %>%
  select(-test)
```

1 Exploratory Data Analysis (18 points for correctness; 2 points for presentation)

First, let's explore the training data.

1.1 Class proportions (3 points)

A good first step when tackling a classification problem is to look at the class proportions.

i. (1 points) What fraction of the training observations are spam?

```
#calc fraction that are spam
frac_spam = spam_train %>%
  summarise(mean(spam == 1))
```

39.7% of training observations are spam

ii. (2 points) Assuming the test data contain the same class proportions, what would be the misclassification error of a naive classifier that always predicts the majority class? **The misclassification error will be 39.7% because the naive classifier will always predict no spam, when 39.7% of emails are actually spam since no spam is the majority class**

1.2 Exploring word frequencies (15 points)

There are 48 features based on word frequencies. In this sub-problem we will explore the variation in these word frequencies, look at most frequent words, as well as the differences between word frequencies in spam versus non-spam emails.

1.2.1 Overall word frequencies (8 points)

Let's first take a look at the average word frequencies across all emails. This will require some `dplyr` manipulations, which the following two sub-parts will guide you through.

i. (3 points) Produce a tibble called `avg_word_freq` containing the average frequencies of each word by calling `summarise_at` on `spam_train`. Print this tibble (no need to use kable). (Hint: Check out the documentation for `summarise_at` by typing `?summarise_at`. Specify all columns starting with "word_freq_" via `vars(starts_with("word_freq"))`).

```
avg_word_freq = spam_train %>%
  summarise_at(vars(starts_with("word_freq")), funs(mean))
```

```
## Warning: `funs()` was deprecated in dplyr 0.8.0.
## Please use a list of either functions or lambdas:
##
##   # Simple named list:
##   list(mean = mean, median = median)
##
##   # Auto named with `tibble::lst()`:
##   tibble::lst(mean, median)
##
##   # Using lambdas
##   list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

```
print(avg_word_freq)
```

```
## # A tibble: 1 x 48
##   word_freq_make word_freq_address word_freq_all word_freq_3d word_freq_our
##            <dbl>             <dbl>         <dbl>        <dbl>         <dbl>
## 1          0.111             0.228         0.274       0.0630         0.318
## # ... with 43 more variables: word_freq_over <dbl>, word_freq_remove <dbl>,
## #   word_freq_internet <dbl>, word_freq_order <dbl>, word_freq_mail <dbl>,
## #   word_freq_receive <dbl>, word_freq_will <dbl>, word_freq_people <dbl>,
## #   word_freq_report <dbl>, word_freq_addresses <dbl>, word_freq_free <dbl>,
## #   word_freq_business <dbl>, word_freq_email <dbl>, word_freq_you <dbl>,
## #   word_freq_credit <dbl>, word_freq_your <dbl>, word_freq_font <dbl>,
## #   word_freq_000 <dbl>, word_freq_money <dbl>, word_freq_hp <dbl>, ...
```

ii. (3 points) Create a tibble called `avg_word_freq_long` by calling `pivot_longer` on `avg_word_freq`. The result should have 48 rows and two columns called `word` and `avg_freq`, the former containing each word and the latter containing its average frequency. Print this tibble (no need to use kable). [Hint: Use `cols = everything()` to pivot on all columns and `names_prefix = "word_freq_"` to remove this prefix.]

```
avg_word_freq_long = avg_word_freq %>%
  #assign each word as a column name and frequency as values
   pivot_longer(cols = everything(), names_prefix = "word_freq_",
                names_to = 'word', values_to = 'avg_freq')
print(avg_word_freq_long)
```

```
## # A tibble: 48 x 2
##    word     avg_freq
##    <chr>       <dbl>
##  1 make       0.111
##  2 address    0.228
##  3 all        0.274
##  4 3d         0.0630
##  5 our        0.318
##  6 over       0.0958
##  7 remove     0.114
##  8 internet   0.107
##  9 order      0.0889
## 10 mail       0.242
## # ... with 38 more rows
```

iii. (2 points) Produce a histogram of the word frequencies. What are the top three most frequent words? How can it be that a word has a frequency of more than 1?

```
# plot histogram of word frequency
avg_word_freq_long %>%
  ggplot(aes(x = avg_freq)) +
  geom_histogram(bins = 15, fill = "grey", col = "black") +
  labs(x = "Word Frequency",
       y = "Number of Words") +
  theme_bw()
```

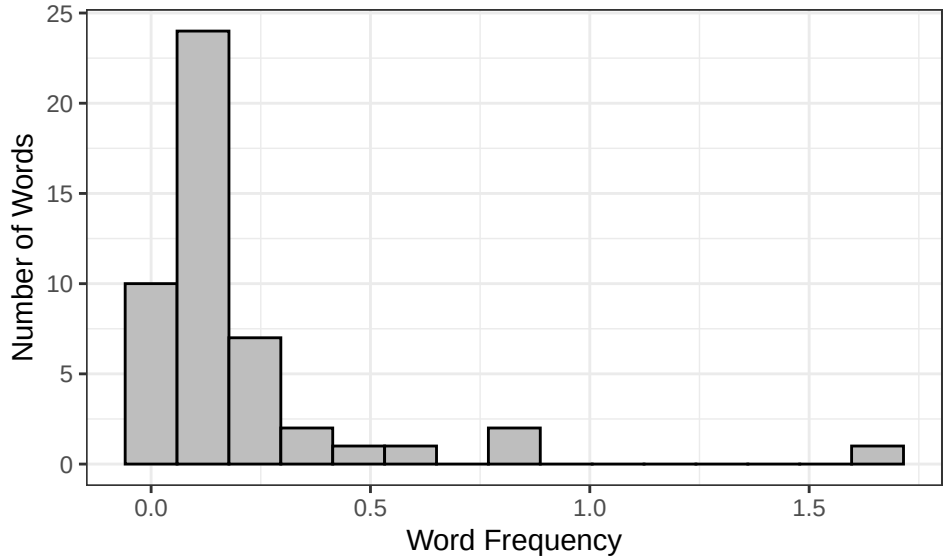

Figure 1: Distribution of word frequency

Table 1: Top 3 Most Frequent Words

word	avg_freq
you	1.661
your	0.820
george	0.772

```
# find top 3 frequency
top_3_freq = avg_word_freq_long %>%
  arrange(desc(avg_freq)) %>%
  head(3)
#print top 3 words
top_3_freq %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 3,
        caption = "Top 3 Most Frequent Words") %>%
 kable_styling(position = "center")
```

The top three most frequent words are "you", "your", and "george". The frequencies are percentages, so their maximum value is 100 rather than 1.

1.2.2 Differencies in word frequencies between spam and non-spam (7 points)

Perhaps even more important than overall average word frequencies are the *differences* in average word frequencies between spam and non-spam emails.

iv. (4 points) For each word, compute the difference between its average frequency among spam and non-spam emails (i.e. average frequency in spam emails minus average frequency in non-spam emails). Store these differences in a tibble called `diff_avg_word_freq`, with columns `word` and `diff_avg_freq`. Print this tibble (no need to use kable).

[Full credit will be given for any logically correct method of doing this. Three extra credit points will be given for a correct solution that employs one continuous sequence of pipes.]

```
diff_avg_word_freq = spam_train %>%
  group_by(spam) %>%
  summarise_at(vars(starts_with("word_freq")), funs(mean)) %>%
  pivot_longer(cols = -c(spam), names_prefix = "word_freq_", names_to =
                 'word', values_to = 'avg_freq') %>%
  pivot_wider(names_from = spam, values_from = avg_freq) %>%
  rename(not_spam = '0', spam = '1') %>%
  #calc difference between frequency in spam and not spam
  summarise(word, diff_avg_freq = spam - not_spam)

print(diff_avg_word_freq)
```

```
## # A tibble: 48 x 2
##    word     diff_avg_freq
##    <chr>            <dbl>
##  1 make            0.0765
##  2 address        -0.115
##  3 all             0.213
##  4 3d              0.157
```

```
##  5 our              0.310
##  6 over             0.129
##  7 remove           0.261
##  8 internet         0.177
##  9 order            0.123
## 10 mail             0.185
## # ... with 38 more rows
```

v. (3 points) Plot a histogram of these word frequency differences. Which three words are most overrepresented in spam emails? Which three are most underrepresented in spam emails? Do these make sense?

```
# plot histogram of word frequency
diff_avg_word_freq  %>%
  ggplot(aes(x = diff_avg_freq)) +
  geom_histogram(bins = 15, fill = "grey", col = "black") +
  labs(x = "Difference in Average Word Frequency",
       y = "Number of Words") +
  theme_bw()
```

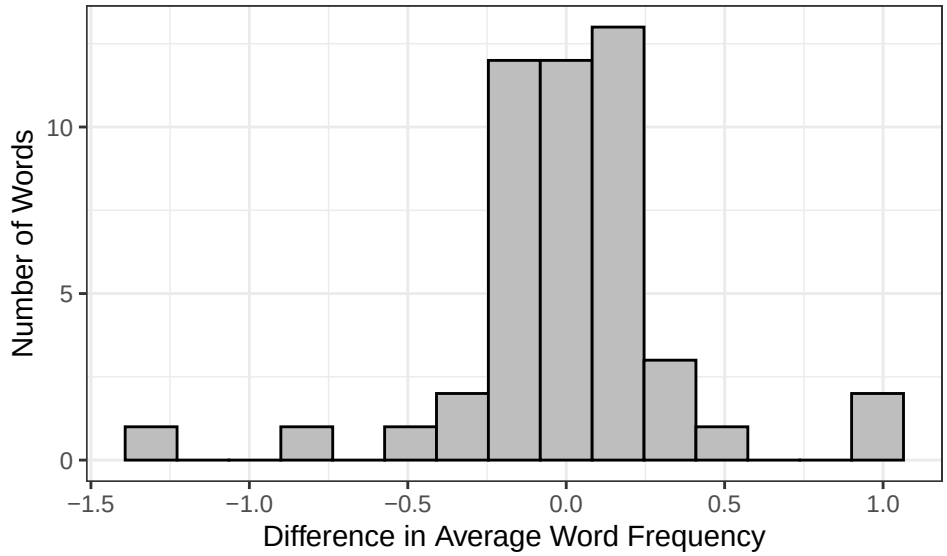

Figure 2: Distribution of word frequency differences between spam and non spam emails.

```
# find top 3 overrepresented
top_3_over = diff_avg_word_freq %>%
  arrange(desc(diff_avg_freq)) %>%
  head(3)

#print top 3 underrepresented words
top_3_over %>%
  kable(format = "latex", row.names = NA,
       booktabs = TRUE,
       digits = 3,
       caption = "Top 3 Most Overrespresented Words in Spam Emails") %>%
 kable_styling(position = "center")
```

Table 2: Top 3 Most Overrespresented Words in Spam Emails

word	diff_avg_freq
you	1.014
your	0.985
free	0.453

Table 3: Top 3 Most Underrespresented Words in Spam Emails

word	diff_avg_freq
george	-1.279
hp	-0.848
hpl	-0.414

```
# find top 3 underrepresented
top_3_under = diff_avg_word_freq %>%
  arrange(diff_avg_freq) %>%
  head(3)

#print top 3 underrepresented words
top_3_under %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 3,
        caption = "Top 3 Most Underrespresented Words in Spam Emails") %>%
 kable_styling(position = "center")
```

The top 3 overrepresented words in spam emails are "you", "your", and "free" while the top 3 underrepresented words in spam emails are "george", "hp", and "hpl". These words make sense because they are more personalized/ descriptive words, whereas the spam emails have more non-descriptive words

2 Classification trees (20 points for correctness; 5 points presentation)

In this problem, we will train classification trees to get some more insight into the relationships between the features and the response.

2.1 Growing the default classification tree (8 points)

i. (1 point) Fit a classification tree with splits based on the Gini index, with default `control` parameters. Plot this tree.

```
tree_fit = rpart(spam ~ .,
                 method = "class",              # classification
                 parms = list(split = "gini"),  # Gini index for splitting
                 data = spam_train)
rpart.plot(tree_fit)
```

ii. (2 points) How many splits are there in this tree? How many terminal nodes does the tree have? **There**

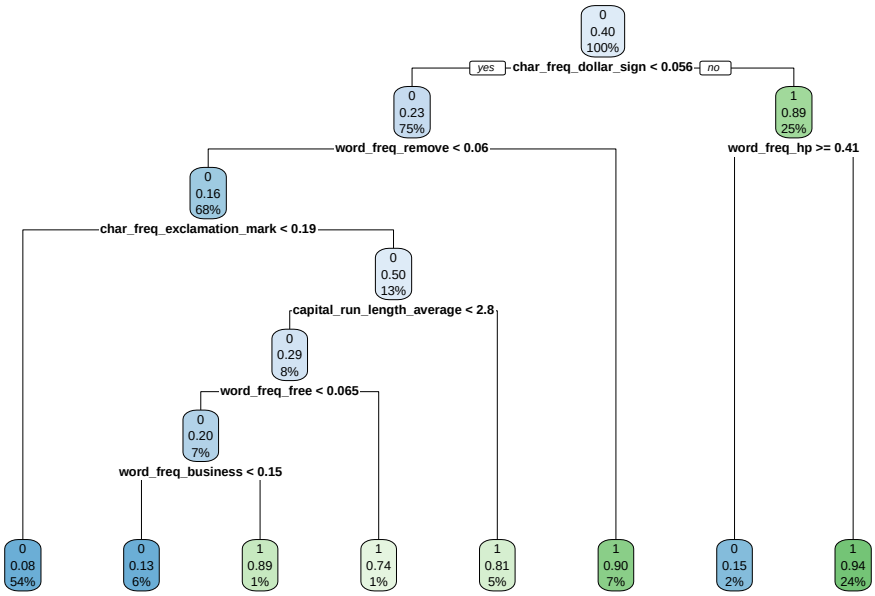

Figure 3: Classification tree for train data that predicts if an email is spam or not spam with splits based on the Gini index

are 7 splits and 8 terminal nodes in the tree

iii. (5 points) What sequence of splits (specify the feature, the split point, and the direction) leads to the terminal node that has the largest fraction of spam observations? Does this sequence of splits make sense as flagging likely spam emails? What fraction of spam observations does this node have? What fraction of the training observations are in this node? **The highest fraction of spam observations occurs when the frequency of the dollar sign is greater than or equal to 0.056 and the the word frequency of hp is less than 0.41. This sequence of splits makes sense because emails with more dollar signs might indicate that someone is trying to attempt a phishing attack. Also, we know that the word hp is underrepresented in spam emails. 24% of the training observations are in this node**

2.2 Finding a tree of optimal size via pruning and cross-validation (12 points)

Now let's find the optimal tree size.

2.2.1 Fitting a large tree T_0 (9 points)

i. (2 points) While we could simply prune back the default tree, there is a possibility the default tree is not large enough. In terms of the bias-variance tradeoff, why would it be a problem if the default tree were not large enough? **It would be a problem if the default tree was not large enough because we might underfit or create an oversimplified tree. We know the more terminal nodes(regions) a tree has, the more flexible the tree is, While a smaller tree might give us lower variance, it will be more biased since it lacks the tools to fully capture trends in the data. The decrease in variance might not overcome the increase in bias, which increases the error of our model. Therefore, pruning a tree that is not large enough may not allow us to pick the tree with the lowest error.**

ii. (2 points) First let us fit the deepest possible tree. In class we talked about the arguments `minsplit`

and `minbucket` to `rpart.control`. What values of these parameters will lead to the deepest possible tree? There is also a third parameter `cp`. Read about this parameter by typing `?rpart.control`. What value of this parameter will lead to the deepest possible tree?

'minsplit' is the minimum number of observations that must exist in a node for a split to be attempted. Thus, when 'minsplit' is set to 2 (because there is no reason to split if there is not at least one observation on either side of the split). Attempting to split when there are 2 observations left gives us the deepest possible tree

'minbucket'is the minimum number of observations in any terminal node. Thus, when "minbucket" is set to 1 (a terminal node cannnot have zero observations), the deepest tree is created

'cp' is the complexity parameter and is used to control the size of the decision tree and to select the optimal tree size. If the cost of adding another variable to the decision tree from the current node is above the value of cp, then tree building does not continue. We could also say that tree construction does not continue unless it would decrease the overall lack of fit by a factor of cp.Therefore, a cp of 0 will lead to the deepest possible tree

iii. (1 point) Fit the deepest possible tree T_0 based on the `minsplit`, `minbucket`, and `cp` parameters from the previous sub-part. Print the CP table for this tree (using kable).

```
set.seed(1) # for reproducibility (DO NOT CHANGE)

tree_fit_deep = rpart(spam ~ .,
                      control = rpart.control(minsplit = 2, minbucket = 1, cp = 0),
                      method = "class",              # classification
                      parms = list(split = "gini"),  # Gini index for splitting
                      data = spam_train)
cp_table_deep = printcp(tree_fit_deep) %>% as_tibble()
```

```
##
## Classification tree:
## rpart(formula = spam ~ ., data = spam_train, method = "class",
##     parms = list(split = "gini"), control = rpart.control(minsplit = 2,
##         minbucket = 1, cp = 0))
##
## Variables actually used in tree construction:
##  [1] capital_run_length_average capital_run_length_longest
##  [3] capital_run_length_total   char_freq_bracket
##  [5] char_freq_dollar_sign      char_freq_exclamation_mark
##  [7] char_freq_parenthesis      word_freq_000
##  [9] word_freq_1999             word_freq_3d
## [11] word_freq_650              word_freq_85
## [13] word_freq_address          word_freq_addresses
## [15] word_freq_all              word_freq_business
## [17] word_freq_conference       word_freq_credit
## [19] word_freq_cs               word_freq_data
## [21] word_freq_edu              word_freq_email
## [23] word_freq_free             word_freq_george
## [25] word_freq_hp               word_freq_hpl
## [27] word_freq_internet         word_freq_lab
## [29] word_freq_mail             word_freq_make
## [31] word_freq_meeting          word_freq_money
## [33] word_freq_order            word_freq_original
## [35] word_freq_our              word_freq_over
## [37] word_freq_people           word_freq_pm
```

Table 4: CP Table for Deepest possible tree

CP	nsplit	rel error	xerror	xstd
0	205	0.001	0.223	0.013

```
## [39] word_freq_re               word_freq_receive
## [41] word_freq_remove           word_freq_report
## [43] word_freq_technology       word_freq_will
## [45] word_freq_you              word_freq_your
##
## Root node error: 1218/3065 = 0.4
##
## n= 3065
##
##       CP nsplit rel error xerror xstd
## 1  5e-01      0     1e+00    1.0 0.02
## 2  1e-01      1     5e-01    0.5 0.02
## 3  4e-02      2     4e-01    0.4 0.02
## 4  3e-02      4     3e-01    0.3 0.01
## 5  2e-02      5     3e-01    0.3 0.01
## 6  1e-02      6     2e-01    0.3 0.01
## 7  8e-03      7     2e-01    0.2 0.01
## 8  6e-03      8     2e-01    0.2 0.01
## 9  5e-03     10     2e-01    0.2 0.01
## 10 4e-03     11     2e-01    0.2 0.01
## 11 4e-03     12     2e-01    0.2 0.01
## 12 3e-03     14     2e-01    0.2 0.01
## 13 2e-03     18     2e-01    0.2 0.01
## 14 2e-03     30     1e-01    0.2 0.01
## 15 1e-03     45     1e-01    0.2 0.01
## 16 1e-03     53     1e-01    0.2 0.01
## 17 1e-03     59     9e-02    0.2 0.01
## 18 1e-03     62     9e-02    0.2 0.01
## 19 8e-04     66     9e-02    0.2 0.01
## 20 5e-04    134     3e-02    0.2 0.01
## 21 5e-04    138     3e-02    0.2 0.01
## 22 4e-04    145     3e-02    0.2 0.01
## 23 0e+00    205     8e-04    0.2 0.01
```

```
cp_table_deep %>%
  tail(1) %>%  #select deepest tree
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 3,
        caption = "CP Table for Deepest possible tree") %>%
 kable_styling(position = "center")
```

iv. (4 points) How many distinct trees are there in the sequence of trees produced in part iii? How many splits does the biggest tree have? How many average observations per terminal node does it have, and why is it not 1? **There are 23 distinct trees. The largest tree has 205 splits.Therefore, there are 206 terminal nodes in this tree and an average of 14.879 observations in each terminal node. The average number of observations per teminal node is not equal to 1 because there are only 57 variables to split on and we cannot have a variable repeat in the tree.**

Random forest won't use an already used feature to split at further levels in any given branch of the tree.

2.2.2 Tree-pruning and cross-validation (3 points)

i. (1 points) Produce the CV plot based on the information in the CP table printed above. For cleaner visualization, plot only trees with `nsplit` at least 2, and put the x-axis on a log scale using `scale_x_log10()`.

```
cp_table_deep %>%
  filter(nsplit >= 2) %>%
  ggplot(aes(x = nsplit+1, y = xerror,
             ymin = xerror - xstd, ymax = xerror + xstd)) +
  geom_point() + geom_line() +
  scale_x_log10() +
  geom_errorbar(width = 0.2) +
  xlab("Number of terminal nodes") + ylab("CV error") +
  geom_hline(aes(yintercept = min(xerror)), linetype = "dashed") +
  theme_bw()
```

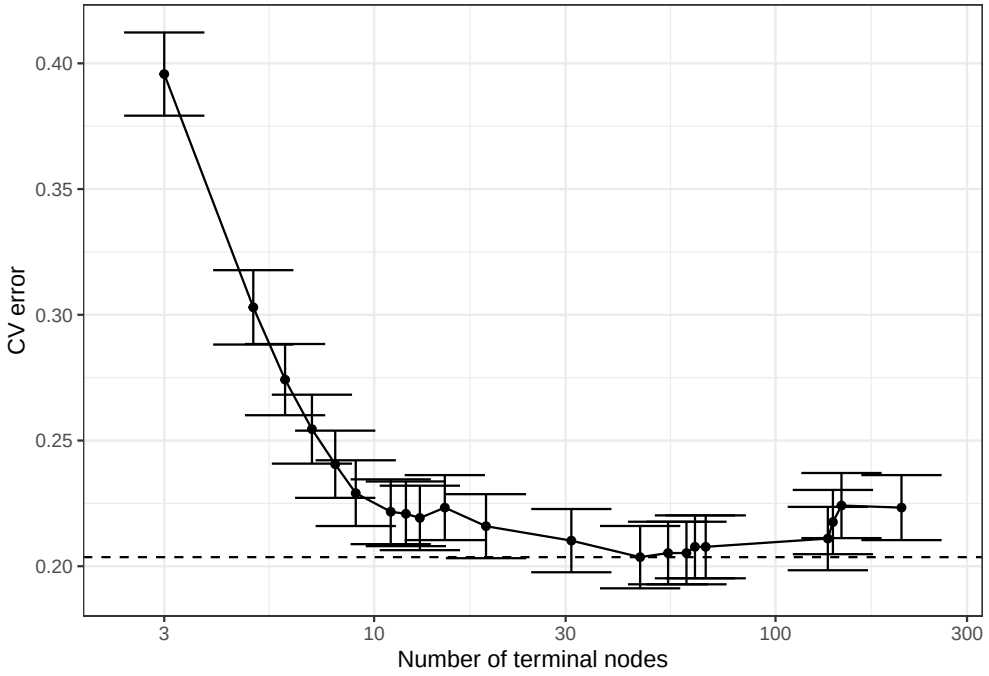

Figure 4: CV plot for deepest tree, where nsplit is grerater than 2

ii. (1 point) Using the one-standard-error rule, how many terminal nodes does the optimal tree have? Is this smaller or larger than the number of terminal nodes in the default tree above?

```
optimal_tree_info = cp_table_deep %>%
  filter(xerror - xstd < min(xerror)) %>%
  arrange(nsplit) %>%
  head(1)
terminal_nodes_optimal = optimal_tree_info %>% select(nsplit) %>% pull() + 1
```

Using the one standard error rule, the tree has 18 splits and 19 terminal nodes, which is higher than than the number of terminal nodes in the default tree above

iii. (1 point) Extract this optimal tree into an object called `optimal_tree` which we can use for prediction on the test set (see the last problem in this homework).

```
optimal_tree = prune(tree = tree_fit_deep, cp = optimal_tree_info$CP)
```

3 Random forests (25 points for correctness; 5 points for presentation)

Note: from this point onward, your code will be somewhat time-consuming. It is recommended that you cache your code chunks using the option `cache = TRUE` in the chunk header. This way, the results of these code chunks will be saved the first time you compile them (or after you change them), making subsequent compilations much faster.

3.1 Running a random forest with default parameters (4 points)

i. (2 points) Train a random forest with default settings on `spam_train`. What value of `mtry` was used?

```
set.seed(1) # for reproducibility (DO NOT CHANGE)
rf_fit = randomForest(factor(spam) ~ ., data = spam_train)
mtry_value = floor(sqrt(ncol(spam_train)-1))
```

The value of 'mtry' is set to the square root of the number of columns. Since there are 57 columns, not including the spam column, 'mtry' is equal to 7

ii. (2 points) Plot the OOB error as a function of the number of trees. Roughly for what number of trees does the OOB error stabilize?

```
tibble(oob_error = rf_fit$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw()
```

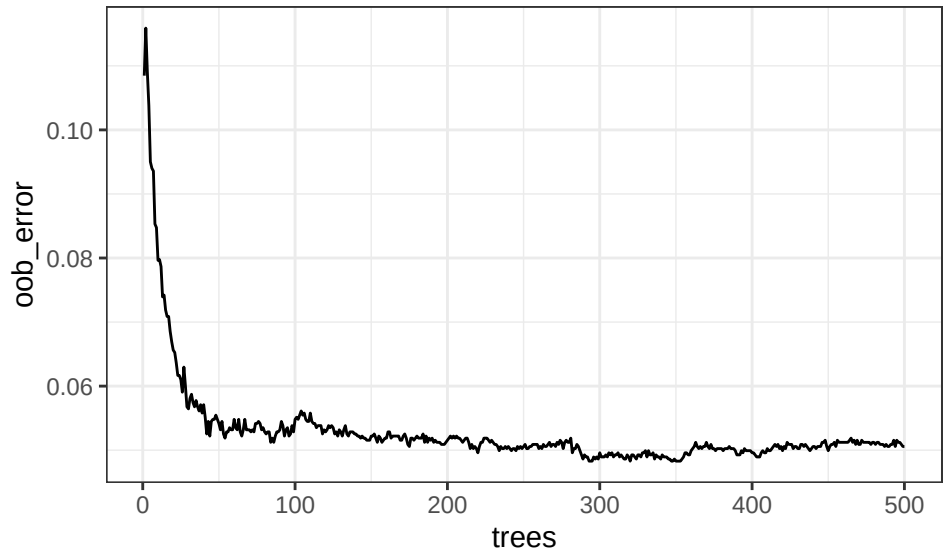

Figure 5: OBB error as a function of the number of trees

OOB error roughly stabilizes at 200 trees in 5

3.2 Computational cost of random forests (7 points)

You may have noticed in the previous part that it took a little time to train the random forest. In this problem, we will empirically explore the computational cost of random forests.

3.2.1 Dependence on whether variable importance is calculated

Recall that the purity-based variable importance is calculated automatically but the OOB-based variable importance measure is only computed if `importance = TRUE` is specified. This is done for computational purposes.

i. (1 point) How long does it take to train the random forest with default parameter settings, with `importance = FALSE`? You can use the command `system.time(randomForest(...))`; see `?system.time` for more details.

```
imp_false = system.time(randomForest(factor(spam) ~ ., data = spam_train,
                                     importance = FALSE))
```

It takes around 4 seconds to train the random forest with default parameter settings

ii. (1 point) How long does it take to train the random forest with default parameter settings except `importance = TRUE`? How many times faster is the computation when `importance = FALSE`?

```
imp_true = system.time(randomForest(factor(spam) ~ ., data = spam_train,
                                    importance = TRUE))
```

It takes around 8 seconds to train the random forest when `importance = TRUE`, which is double the time it took when `importance = FALSE`. Thus, setting computation equal to FALSE is twice as fast as setting it equal to TRUE

3.2.2 Dependence on the number of trees

Another setting influencing the computational cost of running `randomForest` is the number of trees; the default is `ntree = 500`.

i. (3 points) Train five random forests, with `ntree = 100,200,300,400,500` (and `importance = FALSE`). Record the time it takes to train each one, and plot the time against `ntree`. You can programmatically extract the elapsed time by running `system.time(...)["elapsed"]`

```
num_tree = c(100, 200, 300, 400, 500)
times = c(0, 0, 0, 0, 0)
#for loop that calculates and saves run time for dif number trees
for(i in 1:5){
  times[i] = system.time(randomForest(factor(spam) ~ ., data = spam_train,
                                      ntree=num_tree[i]))["elapsed"]
}
#create tibble of num of trees and run time
train_time = tibble(num_tree, times)
train_time %>% ggplot(aes(x= num_tree, y = times)) +
  geom_line()+ theme_bw() +
  labs(x = "Number of Trees", y = "Run Time")
```

ii. (2 points) What relationship between runtime and number of trees do you observe? Does it make sense in the context of the training algorithm for random forests? **As the number of trees increases, the runtime increases. This makes sense because it is more computationally intensive to train on a greater number of trees. This relationship appears to be approximately linear. This makes sense because it takes approxiamtely the same time to train each additional tree in a random forest.**

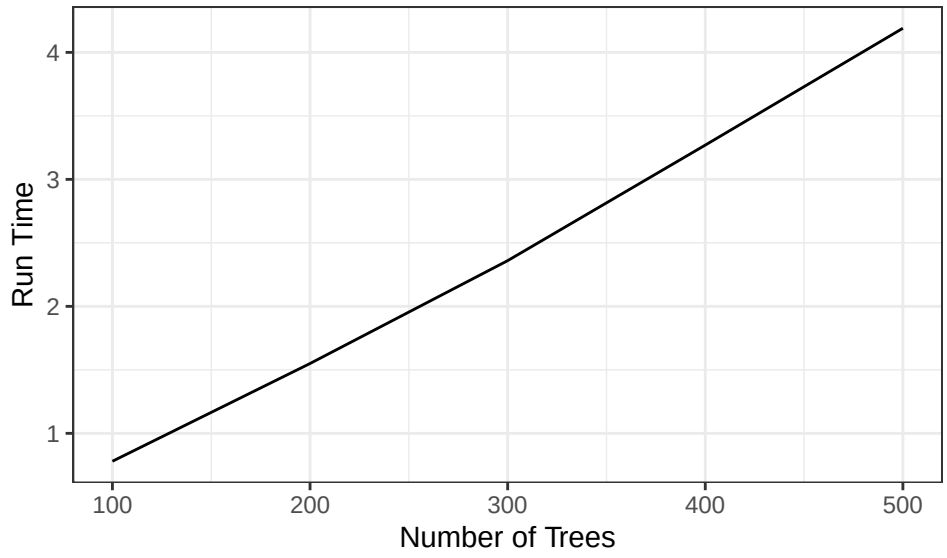

Figure 6: Training time against Number of Trees

3.3 Tuning the random forest (8 points)

i. (2 points) Since tuning the random forest is somewhat time consuming, we want to be careful about tuning it smartly. To this end, does it make sense to tune the random forest with `importance = FALSE` or `importance = TRUE`? Based on OOB error plot from above, what would be a reasonable number of trees to grow without significantly compromising prediction accuracy? **It makes sense to set `importance = FALSE` because doing so cuts the runtime in half. Also, it is not necessary to calculated the OOB-based variable importance when we have already calculated the purity-based variable importance. Based on the OOB error plot from above, it seems reasonable to grow 200 trees because this is when OOB error starts to level off. However, for random forests we do not need to tune B. Instead, we simply need to choose a large enough number. I acknowledge that training on 500 trees will take more time, so I choose to go with 200 trees**

ii. (2 points) About how many minutes would it take to train a random forest with 500 trees for every possible value of m? (For the purposes of this question, you may assume for the sake of simplicity that the choice of m does not impact the training time too much.) Suppose you only have enough patience to wait about 15 seconds to tune your random forest, and you use the reduced number of trees from part i. How many values of m can you afford to try? (The answer will vary based on your computer. Some students will find that there is time for only one or a few values; this is ok.) **Since there are 57 columns in the data (excluding spam), 'mtry' can range from 1 to 57. Thus, there are 57 possible values of m. To train on every value, it would take 57 x 4 seconds or 228 seconds, which is equal to around 4 minutes** **It takes around 1.5 seconds to train based on 200 trees. Therefore I can afford to try 10 values of m because 1.5*10 is equal to 15, which is under the 15 second limit.**

iii. (2 points) Tune the random forest based on the choices in parts i and ii (if on your computer you cannot calculate at least five values of m in 15 seconds, please calculate five values of m, even though it will take longer than 15 seconds). Make a plot of OOB error versus m, and identify the best value of m. How does it compare to the default value of m?

```
#select 10 equally spaced values from 1 to 10
mvalues = round(seq.int(1,57, length.out= 10))
oob_errors = numeric(length(mvalues))
```

```
ntree = 200              #number of trees to test on
for(idx in 1:length(mvalues)){
  set.seed(1)
  m = mvalues[idx]
  rf_fit = randomForest(factor(spam) ~ ., mtry = m, data = spam_train)
  oob_errors[idx] = rf_fit$err.rate[ntree,"OOB"]
}
tibble(m = mvalues, oob_err = oob_errors) %>%
  ggplot(aes(x = m, y = oob_err)) +
  geom_line() + geom_point() +
  scale_x_continuous(breaks = mvalues) +
  theme_bw() +
  labs(x = "Number of Features at Each Split Point",
       y = "Out of Bag Error")
```

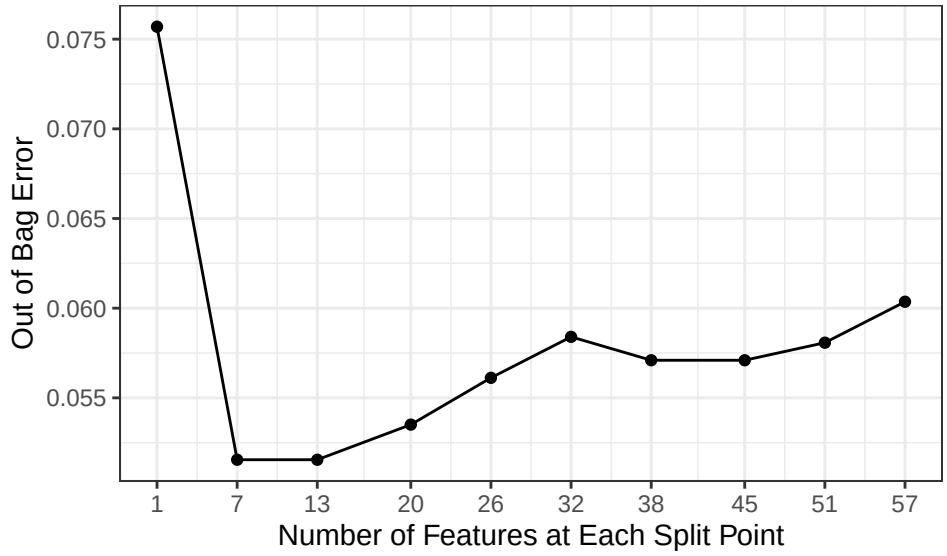

Figure 7: Out of Bag Error against the Number of Features at Each Split Point

```
mtry_optimal = tibble(m = mvalues, oob_err = oob_errors) %>%
  arrange(oob_errors) %>% select(m) %>%
  head(1)
```

The best value of m according to 7 is 7, which is equal to the default value of m.

iv. (2 points) Using the optimal value of m selected above, train a random forest on 500 trees just to make sure the OOB error has flattened out. Also switch to `importance = TRUE` so that we can better interpret the random forest ultimately used to make predictions. Plot the OOB error of this random forest as a function of the number of trees and comment on whether the error has flattened out.

```
set.seed(1) # for reproducibility (DO NOT CHANGE)
rf_fit_tuned = randomForest(factor(spam) ~ ., ntree= 500, mtry = 7, data = spam_train,
                            importance = TRUE)
tibble(oob_error = rf_fit_tuned$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw() +
  labs(x = "Number of Features at Each Split Point",
       y = "Out of Bag Error")
```

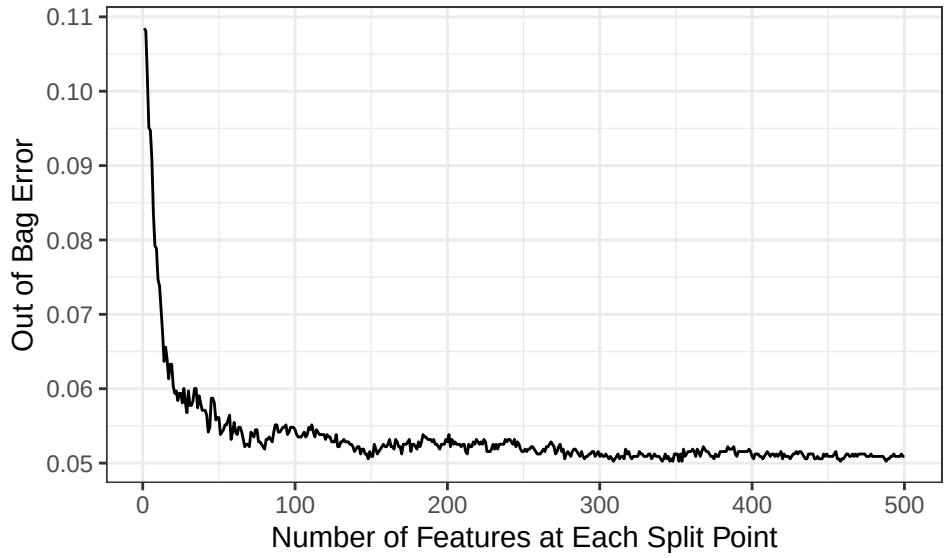

Figure 8: Out of Bag Error against the Number of Trees

The out of bag error appears to have flattened out by the time 200 trees are trained, but trained on 500 trees just to make sure

3.4 Variable importance (6 points)

i. (2 points) Produce the variable importance plot for the random forest trained on the optimal value of m.

We can visualize these importances using the built-in function called `varImpPlot`:

```
varImpPlot(rf_fit_tuned, n.var= 10)
```

ii. (4 points) In order, what are the top three features by each metric? How many features appear in both lists? Choose one of these top features and comment on why you might expect it to be predictive of spam, including whether you would expect an increased frequency of this feature to indicate a greater or lesser probability of spam. **The top 3 features for the accuracy metric are: the exclamation mark, the word "remove", and the dollar sign. The top 3 features for the Gini index are: the exclamation mark, the dollar sign, and the word "remove" Both the exclamation mark and the dollar sign are present in both lists. I except as the number of dollar signsincreases, spam increases because people conducting phishing attacks often use many dollar sign to try and trick users. Also, many spam emails have to do with money.**

4 Boosting (12 points for correctness; 3 points for presentation)

4.1 Model tuning (4 points)

i. (2 points) Fit boosted tree models with interaction depths 1, 2, and 3. For each, use a shrinkage factor of 0.1, 1000 trees, and 5-fold cross-validation.

```
set.seed(1) # for reproducibility (DO NOT CHANGE)
#Fit random forest with interaction depth 1
gbm_fit_1 = gbm(spam ~ .,
                distribution = "bernoulli",
                n.trees = 1000,
```

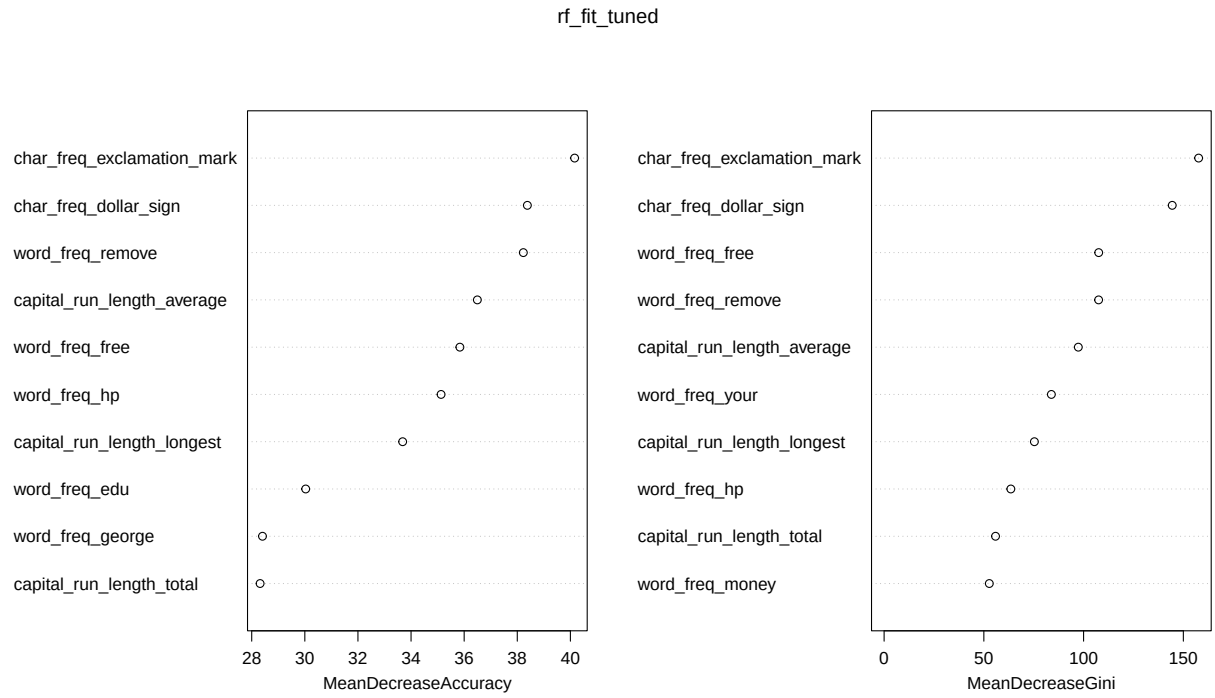

Figure 9: Variable importance plot for random forest model trained on the optimal value of m

```
              interaction.depth = 1,
              shrinkage = 0.1,
              cv.folds = 5,
              data = spam_train)

set.seed(1) # for reproducibility (DO NOT CHANGE)
#Fit random forest with interaction depth 2
gbm_fit_2 = gbm(spam ~ .,
              distribution = "bernoulli",
              n.trees = 1000,
              interaction.depth = 2,
              shrinkage = 0.1,
              cv.folds = 5,
              data = spam_train)

set.seed(1) # for reproducibility (DO NOT CHANGE)
#Fit random forest with interaction depth 3
gbm_fit_3 = gbm(spam ~ .,
              distribution = "bernoulli",
              n.trees = 1000,
              interaction.depth = 3,
              shrinkage = 0.1,
              cv.folds = 5,
              data = spam_train)
```

ii. (2 points) Plot the CV errors against the number of trees for each interaction depth. These three curves should be on the same plot with different colors. Also plot horizontal dashed lines at the minima of

these three curves. What are the optimal interaction depth and number of trees?

```
# extract CV errors
ntrees = 1000
cv_errors = bind_rows(
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_1$cv.error, depth = 1),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_2$cv.error, depth = 2),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_3$cv.error, depth = 3)
)
#calculate min cv_error and min number of trees
best_tree = cv_errors %>%
  filter(cv_err == min(cv_err))


# plot CV errors
cv_errors %>%
  ggplot(aes(x = ntree, y = cv_err, colour = factor(depth))) +
  geom_line() + theme_bw() +
  geom_hline(yintercept = best_tree$cv_err,
             linetype = "dashed") +
  labs(x = "Number of Trees", y = "Cross-Validated Error") +
  scale_y_log10()
```

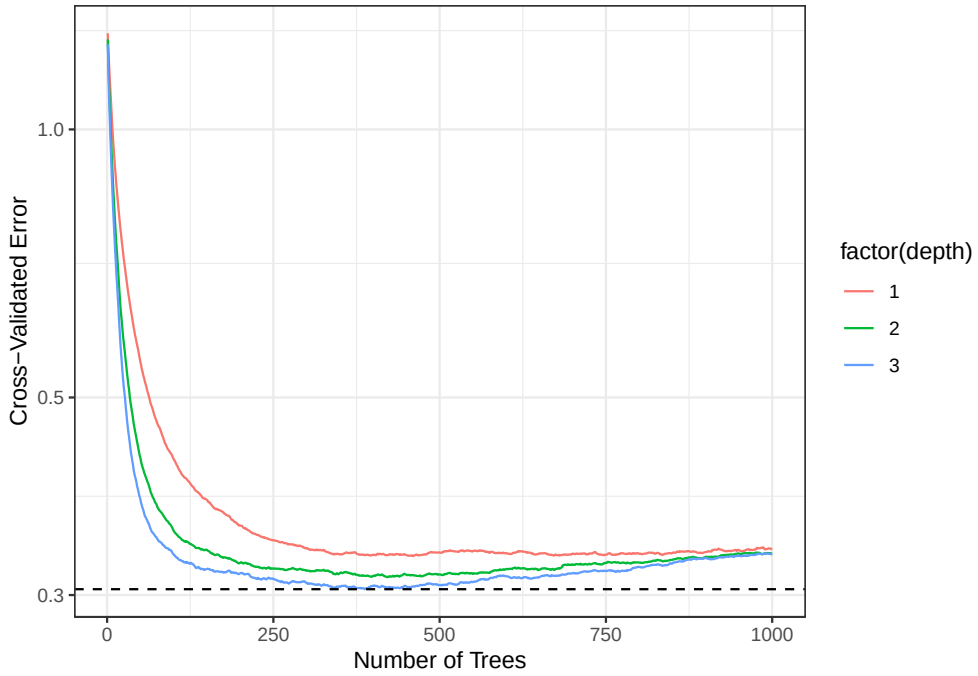

Figure 10: CV error plot for boosted tree model with interaction depths of 1, 2, and 3

The optimal interaction depth is 3 and the optimal number of trees is 485

4.2 Model interpretation (8 points)

i. (4 points) Print the first ten rows of the relative influence table for the optimal boosting model found above (using kable). What are the top three features? To what extent do these align with the top three features of the random forest trained above?

Table 5: Relative Influence Table for Optimal Boosting Model

Variable	Relative Influence
char_freq_exclamation_mark	22.38
char_freq_dollar_sign	18.87
word_freq_remove	11.21
word_freq_free	6.60
word_freq_hp	5.99
word_freq_your	5.83
capital_run_length_longest	5.19
capital_run_length_total	3.58
capital_run_length_average	3.48
word_freq_edu	2.71

```
gbm_fit_tuned = gbm_fit_3
optimal_num_trees = gbm.perf(gbm_fit_3, plot.it = FALSE)
top_10_var = summary(gbm_fit_tuned, n.trees = optimal_num_trees, plotit = FALSE) %>%
  summarise("Variable"= var, "Relative Influence" = rel.inf) %>%
  head(10)
top_10_var_names = top_10_var %>% select(Variable) %>% pull()

top_10_var %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 2,
        caption = "Relative Influence Table for Optimal Boosting Model") %>%
  kable_styling(position = "center")
```

The top 3 features are frequent exclamation marks, frequent dollars signs, and the word remove. These 3 features are the same as the top three feature importance selected by the Gini index in the random forest model.

ii. (4 points) Produce partial dependence plots for the top three features based on relative influence. Comment on the nature of the relationship with the response and whether it makes sense.

```
pdp_1 = plot(gbm_fit_tuned, i.var = top_10_var_names[1], n.trees = optimal_num_trees,
             type = "response")
pdp_2 = plot(gbm_fit_tuned, i.var = top_10_var_names[2], n.trees = optimal_num_trees,
             type = "response")
pdp_3 = plot(gbm_fit_tuned, i.var = top_10_var_names[3], n.trees = optimal_num_trees,
             type = "response")

cowplot::plot_grid(pdp_1, pdp_2, pdp_3, nrow = 1)
```

`char_freq_exclamation_mark` and `char_freq_dollar_sign` exhibit increasing relationships with spam, which makes sense because exclamation marks and dollar signs are typically found in spam emails to get the attention of the viewer. The relationship between `word_freq_remove` and spam seems more complex, with intermediate word frequencies most associated with spam. It's not immediately clear what the connection is between the word "remove" and spam emails because of decrese that is seen in the partial dependence plot.

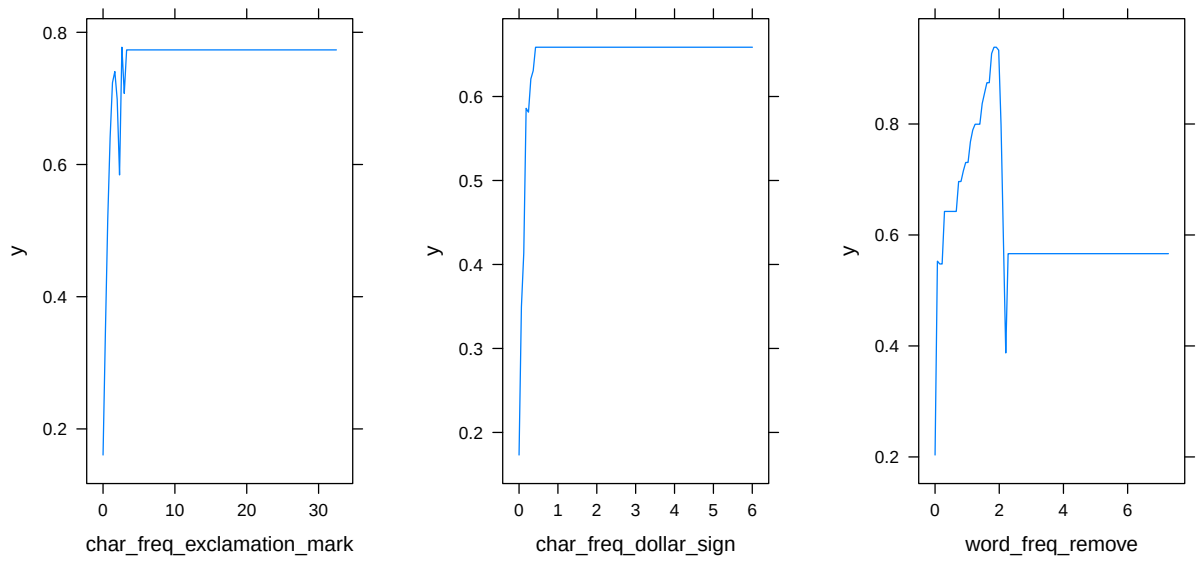

Figure 11: Partial dependence plots for top three features based on relative influence for optimal boosting model

5 Test set evaluation and comparison (8 points for correctness; 2 points for presentation)

i. (2 points) Compute the test misclassification errors of the tuned decision tree, random forest, and boosting classifiers, and print these using kable. Which method performs best?

```
#Decision tree misclassification error
dt_predictions  = predict(optimal_tree, newdata = spam_test, type = "class")
dt_error = mean(dt_predictions != spam_test$spam)

#Random forest misclassification error
rf_predictions  = predict(rf_fit_tuned, newdata = spam_test)
rf_error = mean(rf_predictions != spam_test$spam)

#GBM misclassification error
gbm_probabilities = predict(gbm_fit_tuned, n.trees = optimal_num_trees,
                            type = "response", newdata = spam_test)
gbm_predictions = as.numeric(gbm_probabilities > 0.5)
gbm_error = mean(gbm_predictions != spam_test$spam)

# print nice table
tibble('Decision Tree' = dt_error, 'Random Forest' = rf_error,
       `Boosting Classifier` = gbm_error) %>%
  kable(format = "latex", row.names = NA,
       booktabs = TRUE, digits = 4,
       caption = "Misclassification errors by method") %>%
 kable_styling(position = "center")
```

The random forest performs the best, but it only performs marginally better than the boosting classifier

Table 6: Misclassification errors by method

Decision Tree	Random Forest	Boosting Classifier
0.0833	0.0488	0.0495

ii. (3 points) We might want to see how the test misclassification errors of random forests and boosting vary with the number of trees. The following code chunk is provided to compute these; it assumes that the tuned random forest and boosting classifiers are named `rf_fit_tuned` and `gbm_fit_tuned`, respectively.

```
rf_test_err = apply(
  t(apply(
    predict(rf_fit_tuned,
            newdata = spam_test,
            type = "response",
            predict.all = TRUE)$individual,
    1,
    function(row)(as.numeric(cummean(as.numeric(row)) > 0.5))
    )),
  2,
  function(pred)(mean(pred != spam_test$spam))
  )

gbm_test_err = apply(
  predict(gbm_fit_tuned,
          newdata = spam_test,
          type = "response",
          n.trees = 1:500),
  2,
  function(p)(mean(as.numeric(p > 0.5) != spam_test$spam))
  )
```

Produce a plot showing the misclassification errors of the random forest and boosting classifiers as a function of the number of trees, as well as a horizontal line at the misclassification error of the optimal pruned tree. Put the y axis on a log scale for clearer visualization.

```
ntrees = 500

cv_errors = bind_rows(
  tibble(ntree = 1:ntrees, cv_err = rf_test_err, type = "rf"),
  tibble(ntree = 1:ntrees, cv_err = gbm_test_err, type = "boosting"),
)

cv_errors %>%
  ggplot(aes(x = ntree, y = cv_err, color = type)) +
  scale_y_log10() +
  geom_hline(yintercept = dt_error, linetype = "dashed") +
  geom_line() + theme_bw()
```

iii. (3 points) Between random forests and boosting, which method's misclassification error drops more quickly as a function of the number of trees? Why does this make sense? **Between random forest and boosting, random forest's misclassification error drops faster as a function of the number of trees. This happens because random forests grow deep decision trees in parallel while boosting grows shallow decision trees sequentially.**

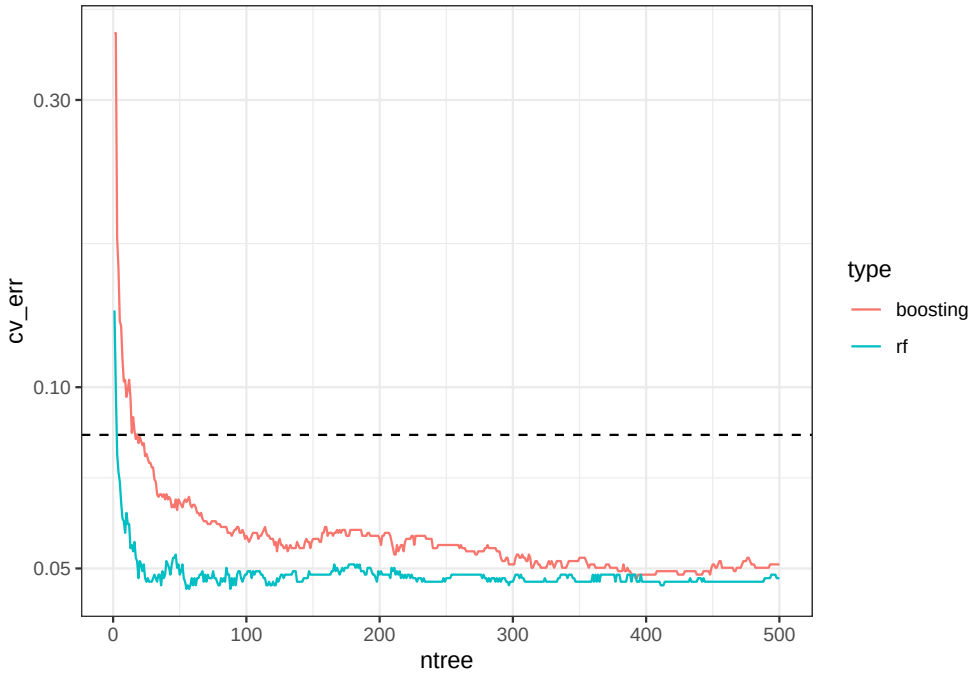

Figure 12: Plot showing the misclassification errors of the random forest and boosting classifiers as a function of the number of trees, as well as a horizontal line at the misclassification error of the optimal pruned tree