STAT 471: Homework 4

Nico Melton

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:

Please list what external references you consulted (e.g. articles, books, or websites):

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(kableExtra)     # kable
```

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?

Solution.

```
# fraction of the training observations that are spam
frac_train_spam = spam_train %>% pull(spam) %>% mean()
```

0.397 of the 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?

Solution.

The misclassification error of a naive classifeir that always predicts the majority class (not spam) would be the fraction of the data that is in the minority class (spam), which is 0.397.

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"))`).

Solution.

```
# summarise word_freq cols
avg_word_freq = spam_train %>%
  summarise_at(vars(starts_with("word_freq")), mean)
avg_word_freq # print
```

```
## # 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.]

Solution.

```
# pivot avg_word_freq longer
avg_word_freq_long = avg_word_freq %>%
  pivot_longer(cols = everything(), values_to = "avg_freq",
               names_to = "word", names_prefix = "word_freq_") %>%
  arrange(avg_freq)
avg_word_freq_long # print
```

```
## # A tibble: 48 x 2
##    word       avg_freq
##    <chr>         <dbl>
##  1 table       0.00529
##  2 parts       0.0124
##  3 conference  0.0314
##  4 cs          0.0463
##  5 original    0.0489
##  6 857         0.0490
##  7 415         0.0496
##  8 report      0.0497
##  9 addresses   0.0507
## 10 receive     0.0581
## # ... 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?

Solution.

```
# graph of word frequencies
avg_word_freq_long %>% ggplot(aes(x = word, y = avg_freq)) +
  geom_col() +
  labs(y = "Average % of words in e-mail") +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
```

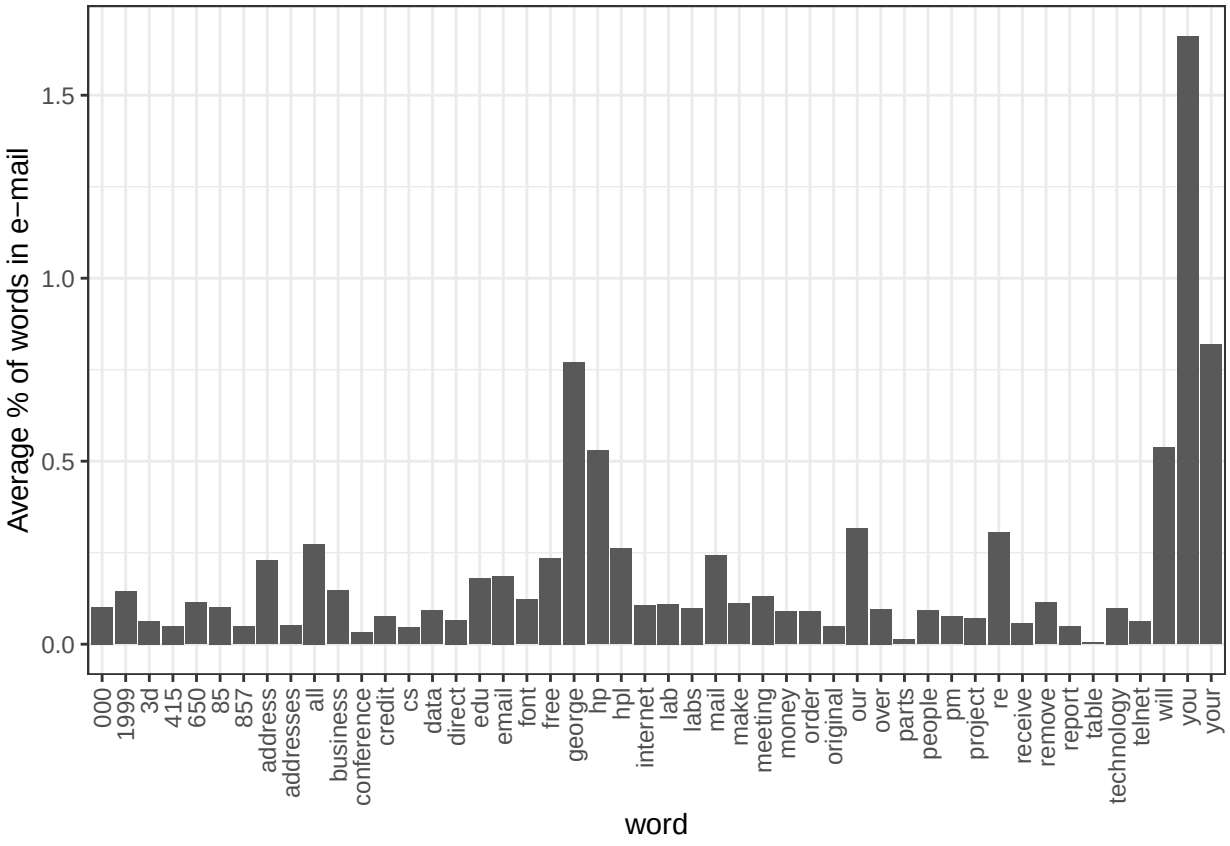

Figure 1: Average word frequencies as a percentage words in the email that match that word.

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

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.]

Solution.

```
# tibble with difference in average frequency of spam and non-spam emails
diff_avg_word_freq = spam_train %>%
  group_by(spam) %>%
  summarise_at(vars(starts_with("word_freq")), mean) %>%
  ungroup() %>%
  pivot_longer(cols = !spam, values_to = "avg_freq",
               names_to = "word", names_prefix = "word_freq_") %>%
  pivot_wider(names_from = "spam", names_prefix = "spam_",
              values_from = "avg_freq") %>%
  mutate(diff_avg_freq = spam_1 - spam_0) %>%
  select(word, diff_avg_freq) %>%
  arrange(diff_avg_freq)
diff_avg_word_freq # print
```

```
## # A tibble: 48 x 2
##    word    diff_avg_freq
##    <chr>           <dbl>
##  1 george         -1.28
##  2 hp             -0.848
##  3 hpl            -0.414
##  4 re             -0.289
##  5 edu            -0.281
##  6 meeting        -0.215
##  7 lab            -0.181
##  8 650            -0.180
##  9 labs           -0.159
## 10 1999           -0.158
## # ... with 38 more rows
```

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

Solution.

```r
# graph of word frequency differences
diff_avg_word_freq %>% ggplot(aes(x = word, y = diff_avg_freq)) +
  geom_col() +
  labs(y = "diff in % of words in email between spam/non-spam") +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
```

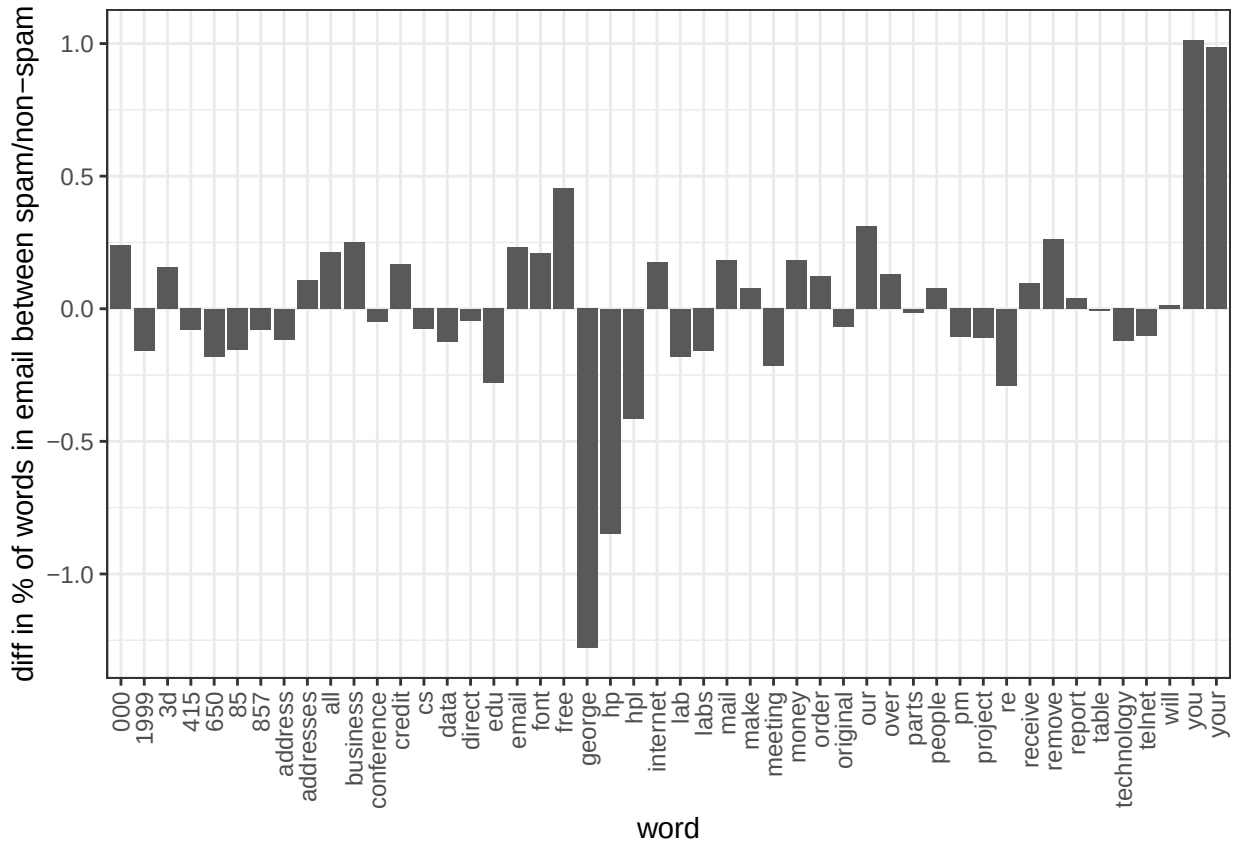

Figure 2: Difference in average word frequencies as a percentage words in the email that match that word.

See Figure 2. The words "you" and "your" are the most overrepresented in the spam emails while the words "george", "hp", and "hpl" are the most underrepresented. This makes sense because "you" and "your" are generic words that could apply to anyone where as the other words are more specific to George and his work.

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.

Solution.

```
# classification tree based on Gini index
class_tree = rpart(spam ~ .,
                   method = "class",
                   parms = list(split = "gini"),
                   data = spam_train)
# plot tree
rpart.plot(class_tree)
```

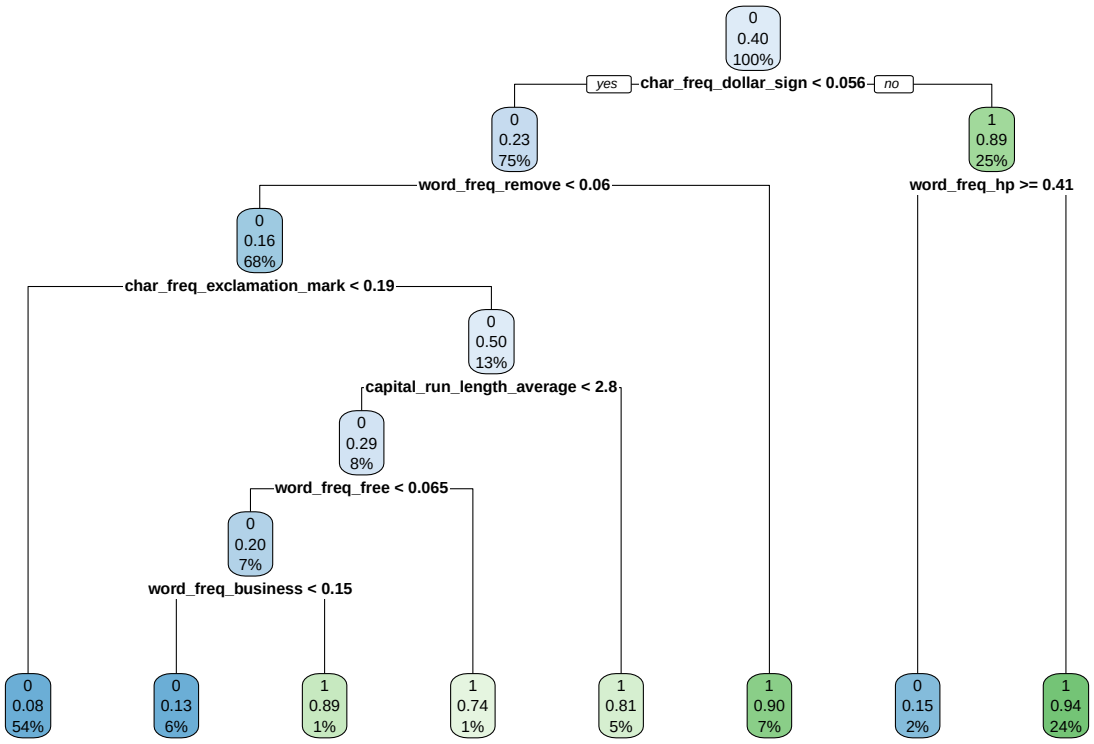

Figure 3: Classification tree of training data based on Gini index

See Figure 3.

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

Solution.

There are 7 splits and 8 terminal nodes.

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?

Solution.

The node with the largest fraction of spam observations has 24% of the training data observations in it and 94% of those are spam observations. The sequence of splits to get to this node is: 1) split on `char_freq_dollar_sign < 0.056` in the NO direction, then 2) split on `word_freq_hp >= 0.41` in the NO direction. This makes sense because as a former employee of HP, George's non-spam emails would probably involve the word "hp" often. Also it makes sense that spam emails may tend to have the "$" sign in them more often because maybe they could be advertising a deal involving money or trying to get the attention of the reader with this symbol.

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?

Solution.

If the default tree is not large enough and we prune it back, we could be increasing the bias by too much (by making the model simpler) while not decreasing the variance enough. Only when we are overfitting the model should we prune the model back in order for the decrease in variance to be more than the increase in bias.

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?

Solution.

The larger `minsplit` and `minbucket` are, the fewer nodes there will be in the tree, therefore, the smaller they are, the more nodes there will be. The deepest possible tree will be one with `minbucket = 1`, `minsplit = 2`, and `cp` equal to some very low value like 0.0001. `minbucket` being 1 means that a terminal leaf node with only 1 observation will not be merged with another terminal leaf node. `minsplit` being 2 means that if a node has 2 or more obervations then can be split.

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).

Solution.

```r
set.seed(1) # for reproducibility (DO NOT CHANGE)
# deepest tree with rpart.control
deepest_tree = rpart(spam ~ .,
                     method = "class",
                     parms = list(split = "gini"),
                     control = rpart.control(minsplit = 2,
                                             minbucket = 1,
                                             cp = 0.0001),
                     data = spam_train)
# display table
deepest_tree$cptable %>%
  kable(format = "latex", longtable = T,
        row.names = NA, booktabs = TRUE,
        col.names = NA,
        digits = 3,
        caption = "CP table of the deepest tree") %>%
  kable_styling(latex_options = c("repeat_header"))
```

Table 1: CP table of the deepest tree

CP	nsplit	rel error	xerror	xstd
0.493	0	1.000	1.000	0.022
0.144	1	0.507	0.530	0.019
0.042	2	0.362	0.396	0.017
0.028	4	0.278	0.303	0.015
0.017	5	0.250	0.274	0.014
0.011	6	0.233	0.255	0.014
0.008	7	0.222	0.241	0.013
0.006	8	0.213	0.229	0.013
0.005	10	0.202	0.222	0.013
0.004	11	0.197	0.221	0.013
0.004	12	0.193	0.219	0.013
0.003	14	0.186	0.223	0.013
0.002	18	0.172	0.216	0.013
0.002	30	0.143	0.210	0.013
0.001	45	0.118	0.204	0.012
0.001	53	0.104	0.205	0.012
0.001	59	0.094	0.205	0.012
0.001	62	0.091	0.208	0.013
0.001	66	0.087	0.208	0.013
0.001	134	0.031	0.211	0.013
0.000	138	0.029	0.218	0.013
0.000	145	0.025	0.224	0.013
0.000	205	0.001	0.223	0.013

See Table 1.

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?

Solution.

There are 23 distinct trees. The biggest tree has 205 splits. The average observation per terminal node in the biggest tree is 14.879. The average is not 1 because some terminal nodes aren't worth splitting any further because they contain all of the same type (all spam or all non-spam).

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()`.

Solution.

```
# make tibble and filter
deep_cp_table = deepest_tree$cptable %>%
  as_tibble() %>%
  filter(nsplit >= 2)
# CV plot
deep_cp_table %>%
  ggplot(aes(x = nsplit+1, y = xerror,
             ymin = xerror - xstd, ymax = xerror + xstd)) +
  geom_point() + geom_line() +
  geom_errorbar(width = 0.02) +
  scale_x_log10() +
  xlab("Number of terminal nodes") + ylab("CV error") +
  geom_hline(aes(yintercept = min(xerror)), linetype = "dashed") +
  theme_bw()
```

See Figure 4.

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?

Solution.

```
optimal_tree_info = deep_cp_table %>%
  filter(xerror - xstd < min(xerror)) %>%
  arrange(nsplit) %>%
  head(1)
```

The optimal tree has 18 terminal nodes. This is smaller than the number of terminal nodes in the default tree.

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).

Solution.

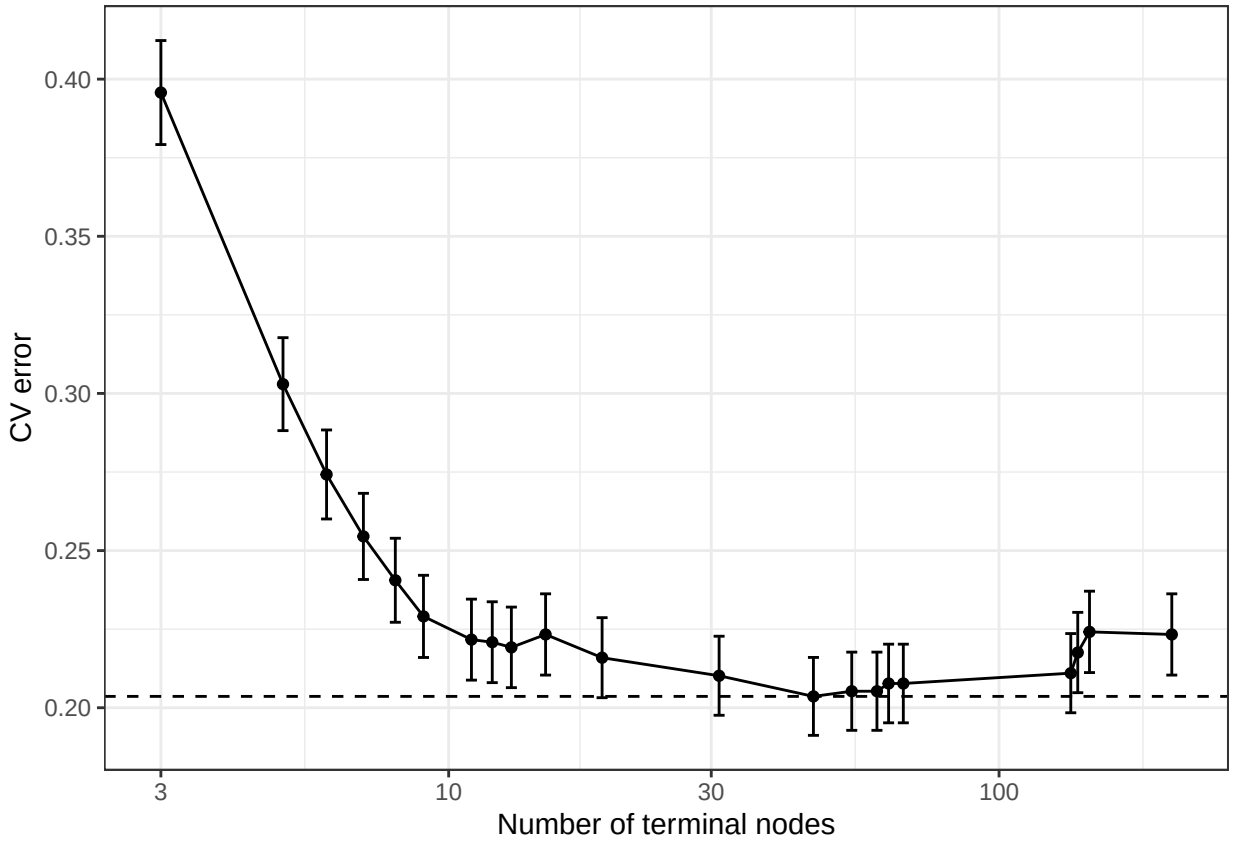

Figure 4: CV plot.

```r
# optimal tree
optimal_tree = optimal_tree = prune(deepest_tree, 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?

Solution.

```r
set.seed(1) # for reproducibility (DO NOT CHANGE)
# random forest with spam_train
rf_fit = randomForest(factor(spam) ~ ., data = spam_train)
```

The value of `mtry` is 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?

Solution.

```r
# plot OOB error as a function of the number of tree
tibble(oob_error = rf_fit$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw()
```

See Figure 5.Around 300 trees is when the OOB stabilizes.

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.

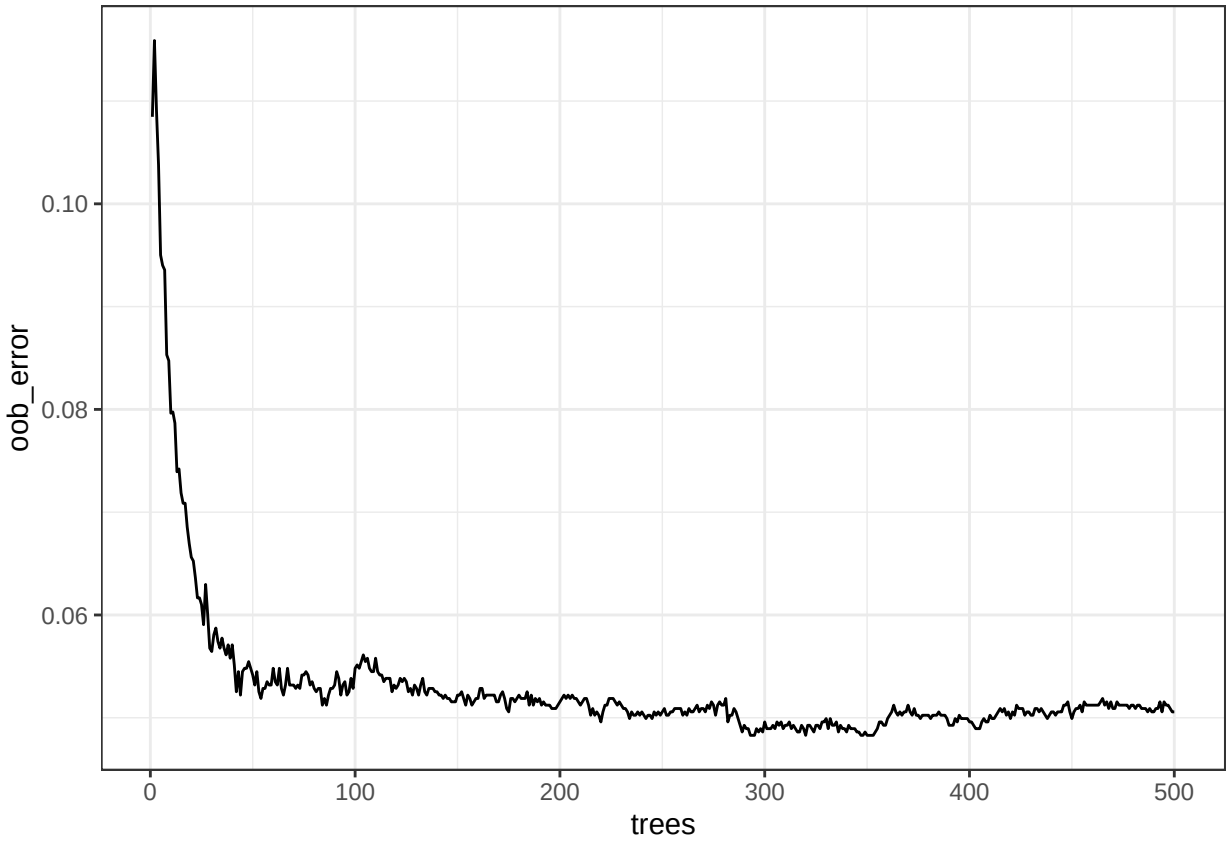

Figure 5: OOB error as a function of the number of trees.

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.

Solution.

```
# time of randomForest with importance = F
time_false = system.time(randomForest(factor(spam) ~ .,
                                      importance = FALSE,
                                      data = spam_train))
```

The time it takes with `importance = FALSE` is 5.578 seconds.

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`?

Solution.

```
# time of randomForest with importance = T
time_true = system.time(randomForest(factor(spam) ~ .,
                                      importance = TRUE,
                                      data = spam_train))
```

The time it takes with `importance = TRUE` is 13.346 seconds..

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"]`

Solution.

```
# vector with number of trees
ntrees = c(100,200,300,400,500)
# create tibble to store times
rf_times = tibble(ntree = ntrees, time = NA)
# for loop to train five random forests and record the time
for(i in ntrees) {
  rf_times[rf_times$ntree == i,]$time =
    system.time(
      randomForest(factor(spam) ~ .,
                   importance = FALSE,
                   data = spam_train)
    )["elapsed"]
}
# plot the times
rf_times %>%
```

```
ggplot(aes(x = ntree, y = time)) +
geom_point() +
geom_line() +
labs(x = "number of trees", y = "runtime (seconds)") +
theme_bw()
```

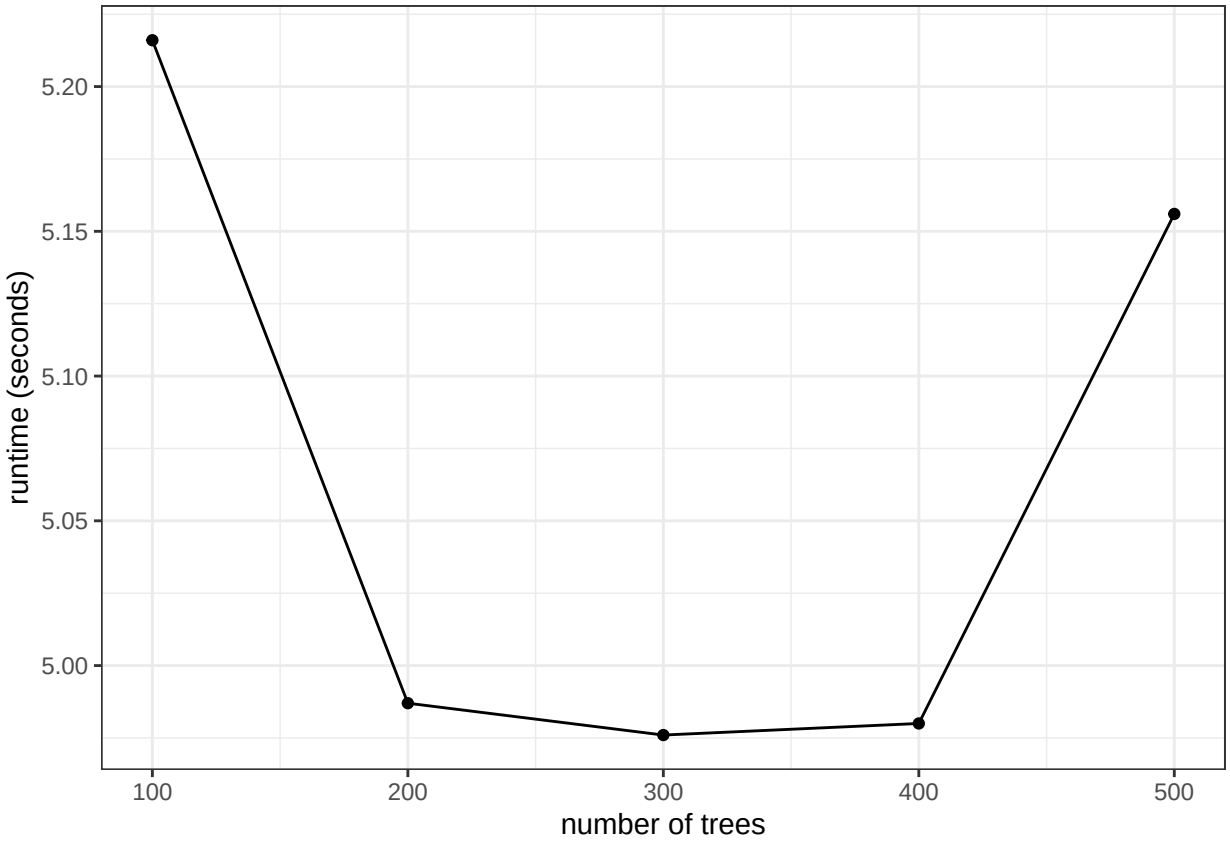

Figure 6: Runtimes of training five random forests.

See Figure 6.

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?

Solution.

There isn't a clear relationship between runtime and the number of trees. We would expect there to be positive relationship - as the number of trees increases so does the runtime - however all the times are very similar and trend is not clear.

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?

Solution.

It makes sense to tune the random forest with `importance = FALSE` since the runtime is over twice a long when `importance = TRUE` and tuning runs many random forests. Based on the OOB error plot, a reasonable number of trees to grow is 300.

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.)

Solution.

It would take about 5.2 seconds per random forest $\times$ 57 values of m $\approx$ 5 minutes **to train a random forest with 500 trees for every possible value of m. Reducing the number of trees won't really affect the runtime of each random forest. With only about 15 seconds, I could only afford to try about 3 values of `m`.**

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?

Solution

```
# roughly evenly spaced values between 1 and 57
m_vals = c(1, 15, 29, 43, 57)
# store oob_errors
oob_errors = numeric(length(m_vals))
# number of trees
ntree = 300
# for loop to run randomForests
for(idx in 1:length(m_vals)){
  m = m_vals[idx]
  rf_fit_temp = randomForest(factor(spam) ~ ., mtry = m, data = spam_train)
  oob_errors[idx] = rf_fit_temp$err.rate[ntree,1]
}
# create tibble and plot
tibble(m = m_vals, oob_err = oob_errors) %>%
  ggplot(aes(x = m, y = oob_err)) +
  geom_line() + geom_point() +
  scale_x_continuous(breaks = m_vals) +
  labs(y = "OOB error") +
  theme_bw()
```

See Figure 7. The best value of `m` that I tested is `15`. This is more than the default value of `m` of 7, however it's unclear whether the default value of `m` is better than this "optimal" value.

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.

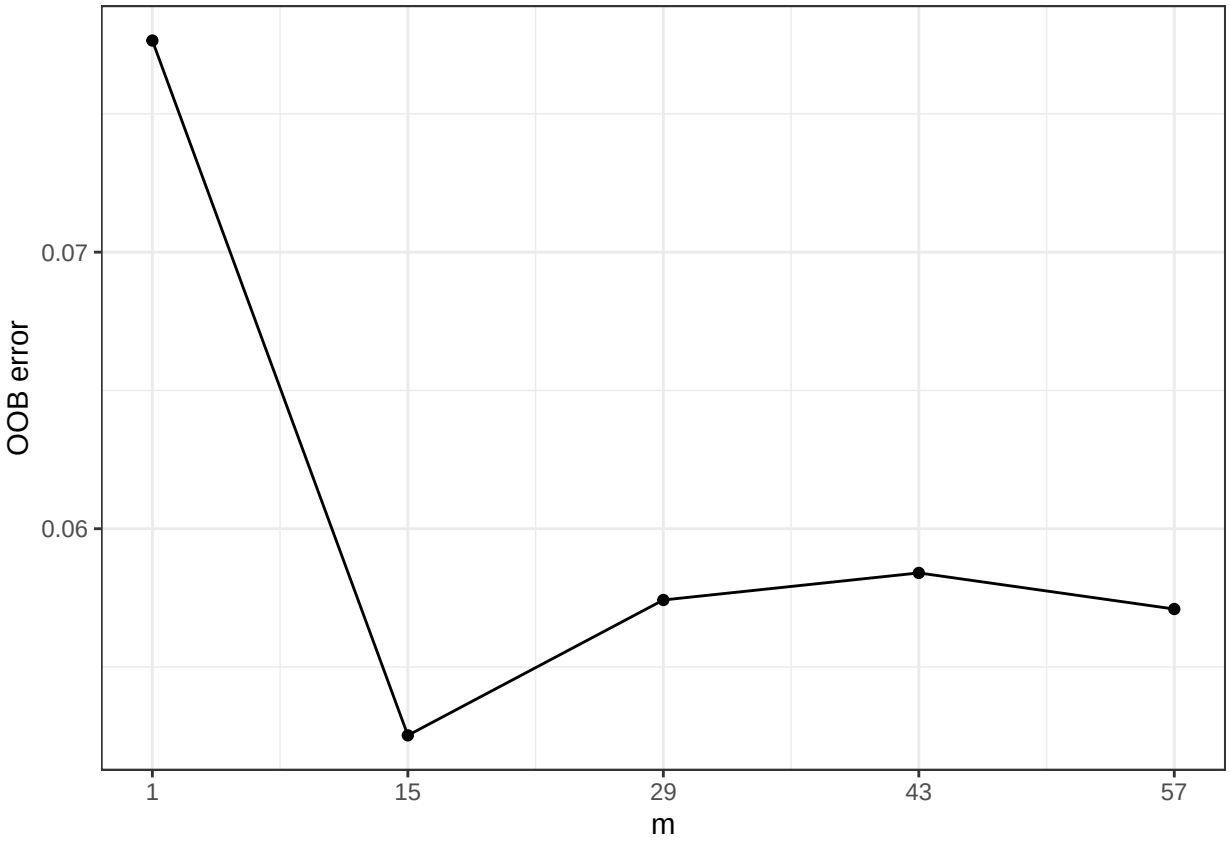

Figure 7: OOB error version m

Solution.

```
set.seed(1) # for reproducibility (DO NOT CHANGE)
# randomForest with optimal value of m (15)
rf_fit_optimal = randomForest(factor(spam) ~ .,
                              importance = TRUE,
                              mtry = 15,
                              data = spam_train)
# plot OOB error as a function of the number of tree
tibble(oob_error = rf_fit_optimal$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw()
```

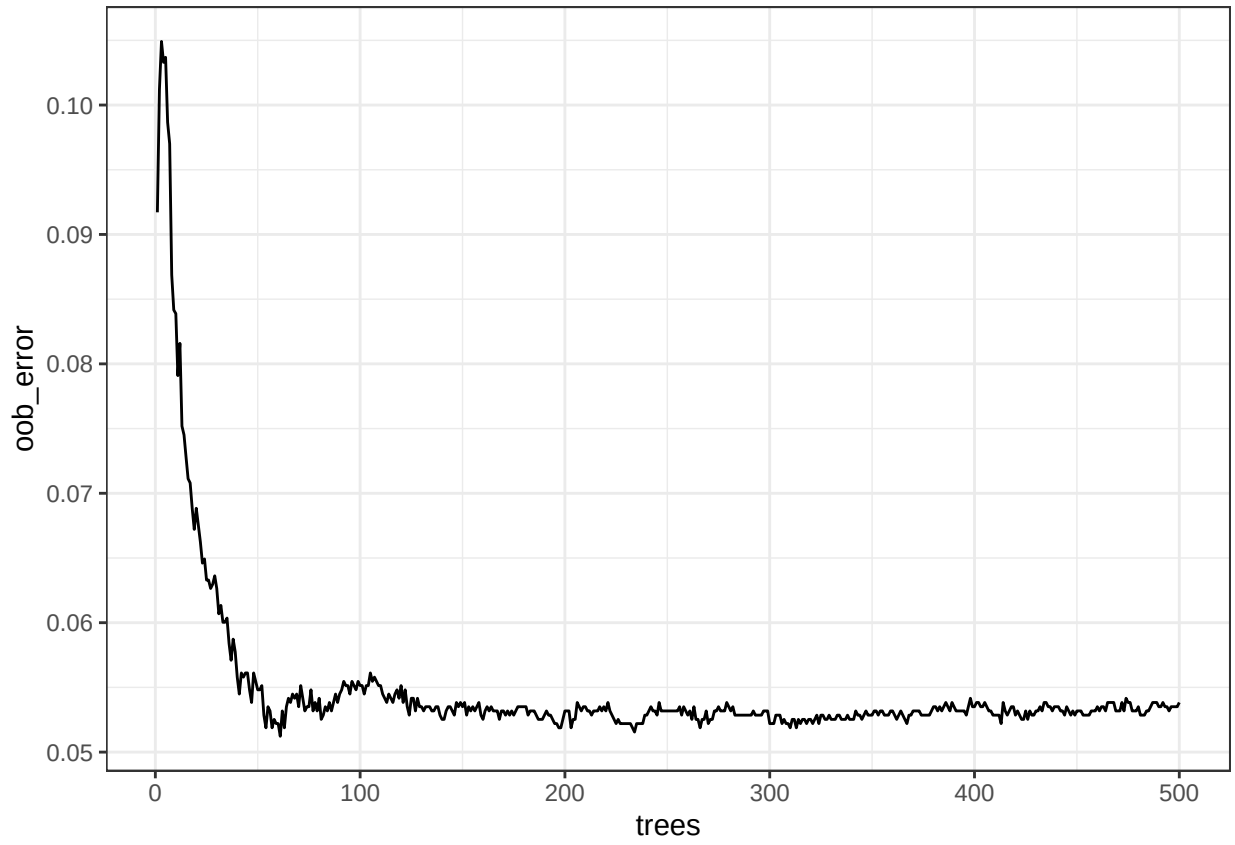

Figure 8: OOB error as a function of the number of trees with the optimal value of m.

See Figure 8. The OOB error has flattened out around 0.054.

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.

Solution.

```r
# plot the variable importance plot
varImpPlot(rf_fit_optimal, main = NA, n.var = 10)
```

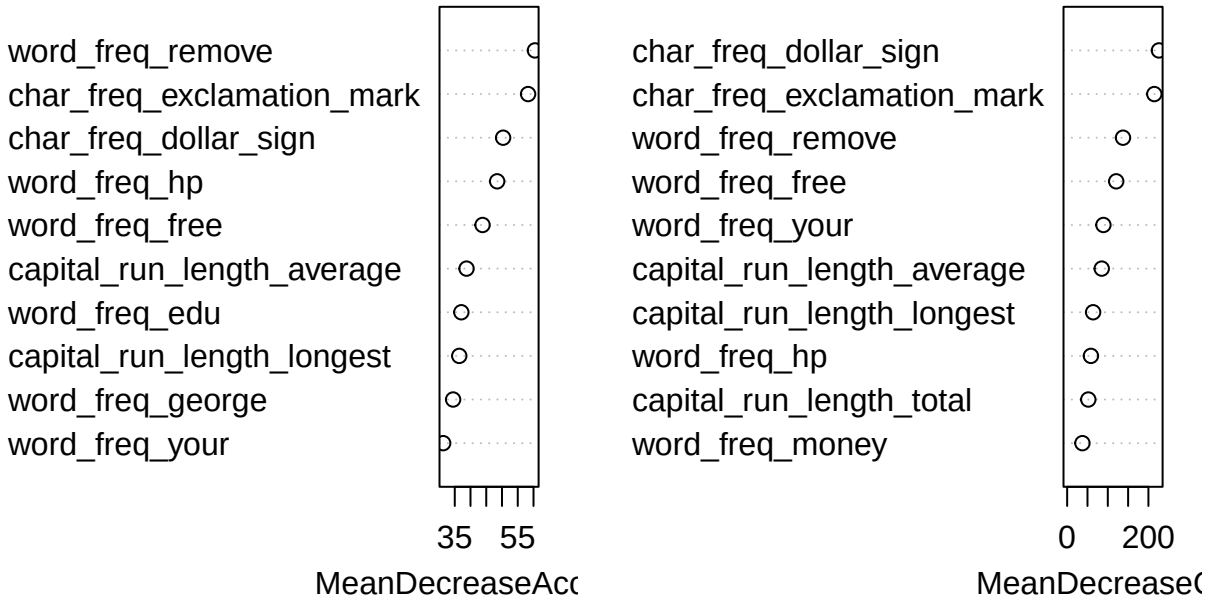

Figure 9: Variable importance plot for the random forest trained on the optimal value of m.

See Figure 9.

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.

Solution.

The top three features by the OOB metric in order from first to third are `word_freq_remove`, `char_freq_exclamation_mark`, and `char_freq_dollar_sign`. The top three features by the Gini metric are the same but in the following order from first to third: `char_freq_dollar_sign`, `char_freq_exclamation_mark`, and `word_freq_remove`. From the top 10 features of both lists, 8 appear in both lists. The `char_freq_exclamation_mark` feature makes sense to be in the top lists as an indicator for spam because many spam emails try to get the attention of the recipient and using exclamation marks can be a way to do this. I would expect an increased frequency of this feature to indicate a greater probability of spam.

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.

Solution.

```
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,
                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?

Solution.

```
# create tibble
cv_num_trees =
tibble(Iteration = 1:1000,
       CV_1 = gbm_fit_1$cv.error,
       CV_2 = gbm_fit_2$cv.error,
       CV_3 = gbm_fit_3$cv.error) %>%
  pivot_longer(-Iteration, names_to = "depth",
```

```
                names_prefix = "CV_", values_to = "CV")
# plot
cv_num_trees %>%
  ggplot(aes(x = Iteration, y = CV, color = depth)) + geom_line() +
  geom_vline(xintercept = gbm.perf(gbm_fit_1, plot.it = FALSE),
             linetype = "dashed", color = "red") +
    geom_vline(xintercept = gbm.perf(gbm_fit_2, plot.it = FALSE),
             linetype = "dashed", color = "green") +
    geom_vline(xintercept = gbm.perf(gbm_fit_3, plot.it = FALSE),
             linetype = "dashed", color = "blue") +
  geom_hline(yintercept = min(gbm_fit_1$cv.error),
             linetype = "dashed", color = "red") +
  geom_hline(yintercept = min(gbm_fit_2$cv.error),
             linetype = "dashed", color = "green") +
  geom_hline(yintercept = min(gbm_fit_3$cv.error),
             linetype = "dashed", color = "blue") +
  labs(y = "CV error") +
  scale_y_log10() +
  theme_bw()
```

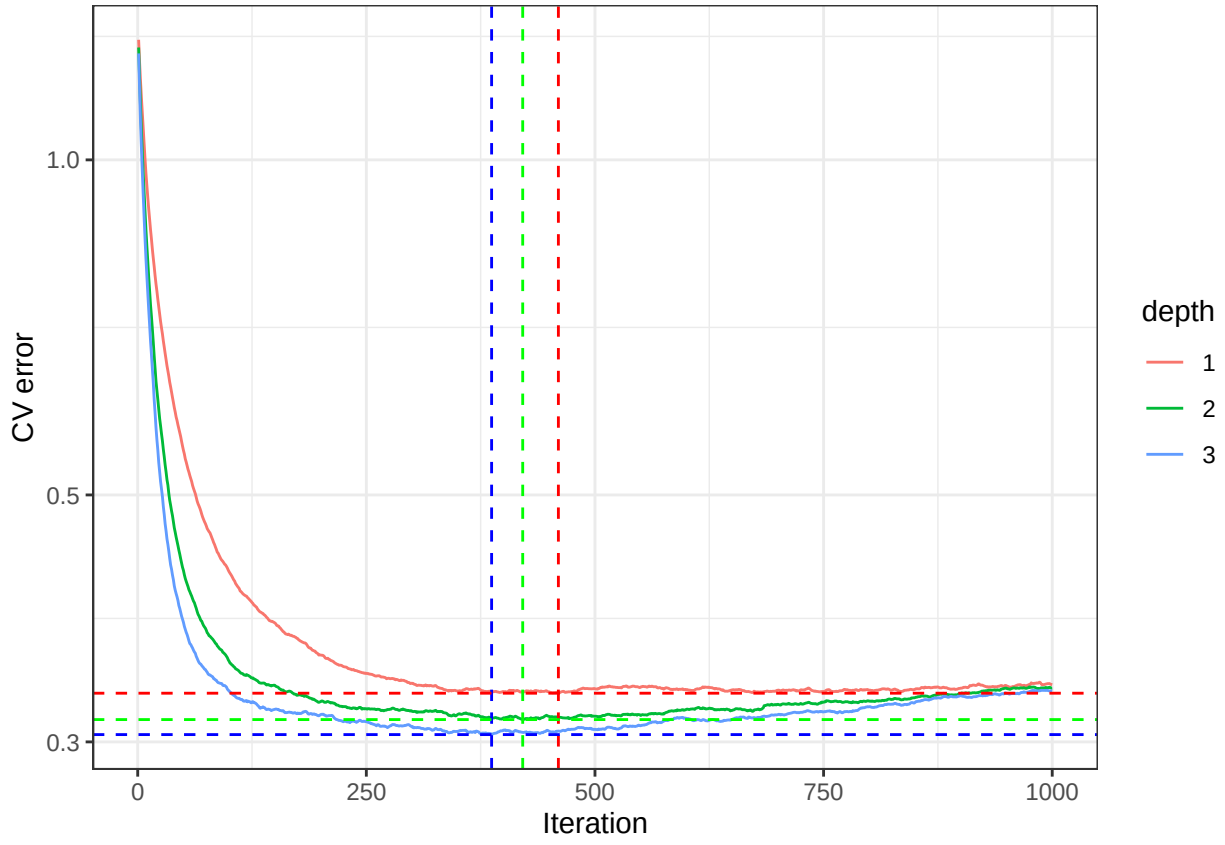

Figure 10: CV errors against the number of trees for each interaction depth.

See figure 10. The optimal interaction depth is 3 and the optimal number of trees is 387

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?

Solution.

```
# get the optimal fit
gbm_fit_optimal = gbm(spam ~ .,
                      distribution = "bernoulli",
                      n.trees = gbm.perf(gbm_fit_3, plot.it = FALSE),
                      interaction.depth = 3,
                      shrinkage = 0.1,
                      cv.folds = 5,
                      data = spam_train)
optimal_num_trees = gbm.perf(gbm_fit_3, plot.it = FALSE)
# table using kable
summary(gbm_fit_optimal,
        n.trees = optimal_num_trees,
        plotit = FALSE) %>%
  slice_head(n= 10) %>%
  select(Variable = var, `Relative Influence` = rel.inf) %>%
  kable(format = "latex", longtable = T,
        row.names = NA, booktabs = TRUE,
        col.names = NA,
        digits = 3,
        caption = "The relative influence table for the optimal boosting model - top ten features.") %>%
  kable_styling(latex_options = c("repeat_header"))
```

Table 2: The relative influence table for the optimal boosting model - top ten features.

	Variable	Relative Influence
char_freq_exclamation_mark	char_freq_exclamation_mark	20.26
char_freq_dollar_sign	char_freq_dollar_sign	17.64
word_freq_remove	word_freq_remove	10.88
word_freq_free	word_freq_free	7.76
word_freq_your	word_freq_your	6.74
word_freq_hp	word_freq_hp	6.01
capital_run_length_average	capital_run_length_average	5.81
capital_run_length_longest	capital_run_length_longest	5.10
word_freq_edu	word_freq_edu	2.42
word_freq_george	word_freq_george	2.38

See Table 2. The top three features are `char_freq_exclamation_mark`, `char_freq_dollar_sign`, and `word_freq_remove`. These are the same of the random forest trained above.

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.

Solution.

```
plot(gbm_fit_optimal, i.var = "char_freq_exclamation_mark", n.trees = optimal_num_trees)
```

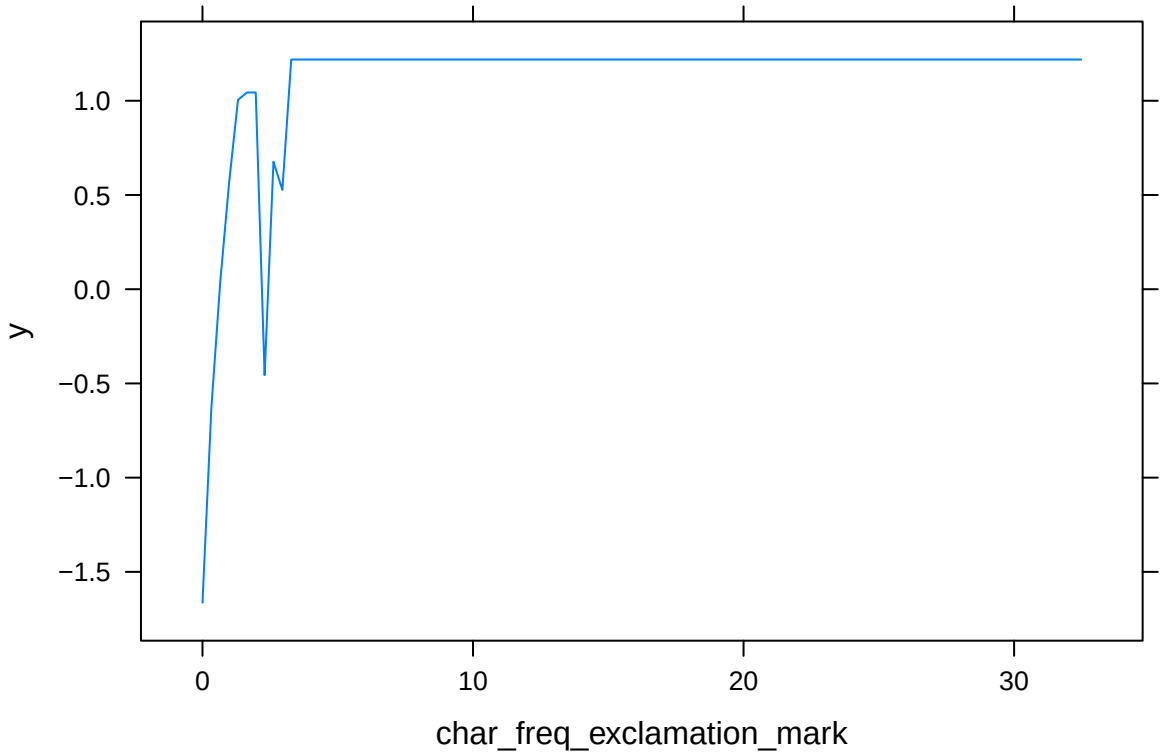

Figure 11: Partial dependence plot for top feature.

```
plot(gbm_fit_optimal, i.var = "char_freq_dollar_sign", n.trees = optimal_num_trees)
```

```
plot(gbm_fit_optimal, i.var = "word_freq_remove", n.trees = optimal_num_trees)
```

See Figures 11, 12, and 13.`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. 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.

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?

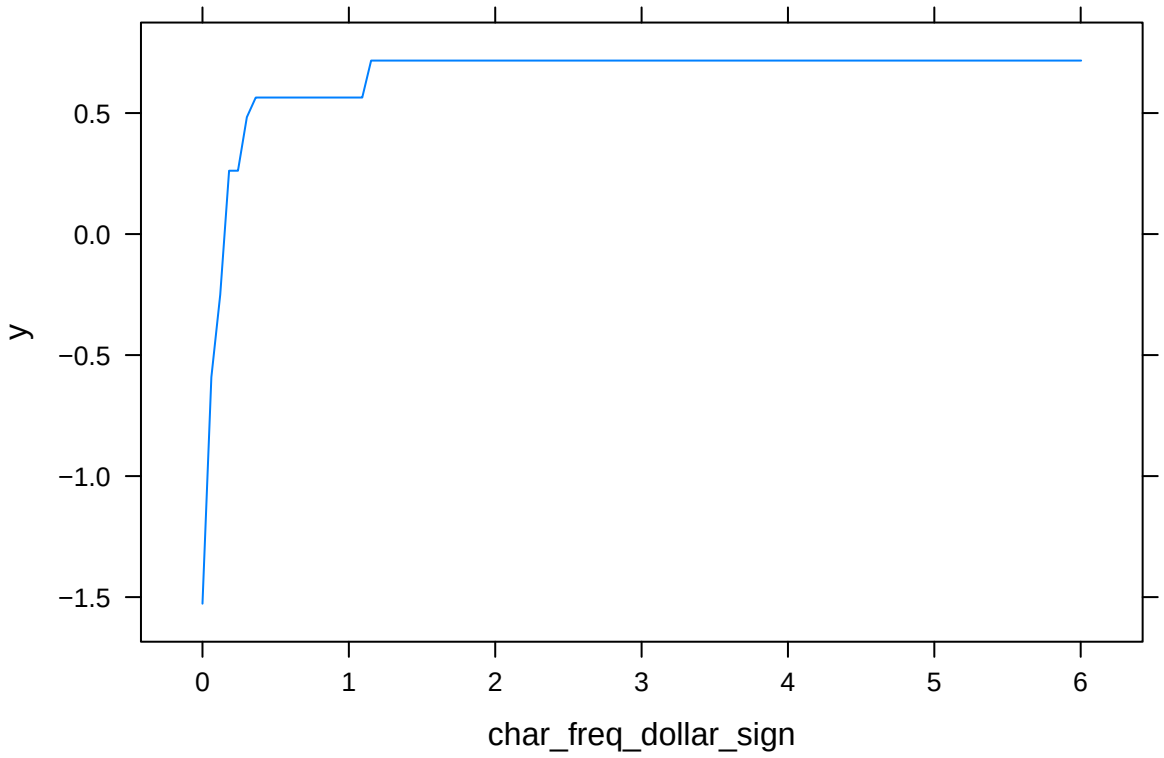

Figure 12: Partial dependence plot for second feature.

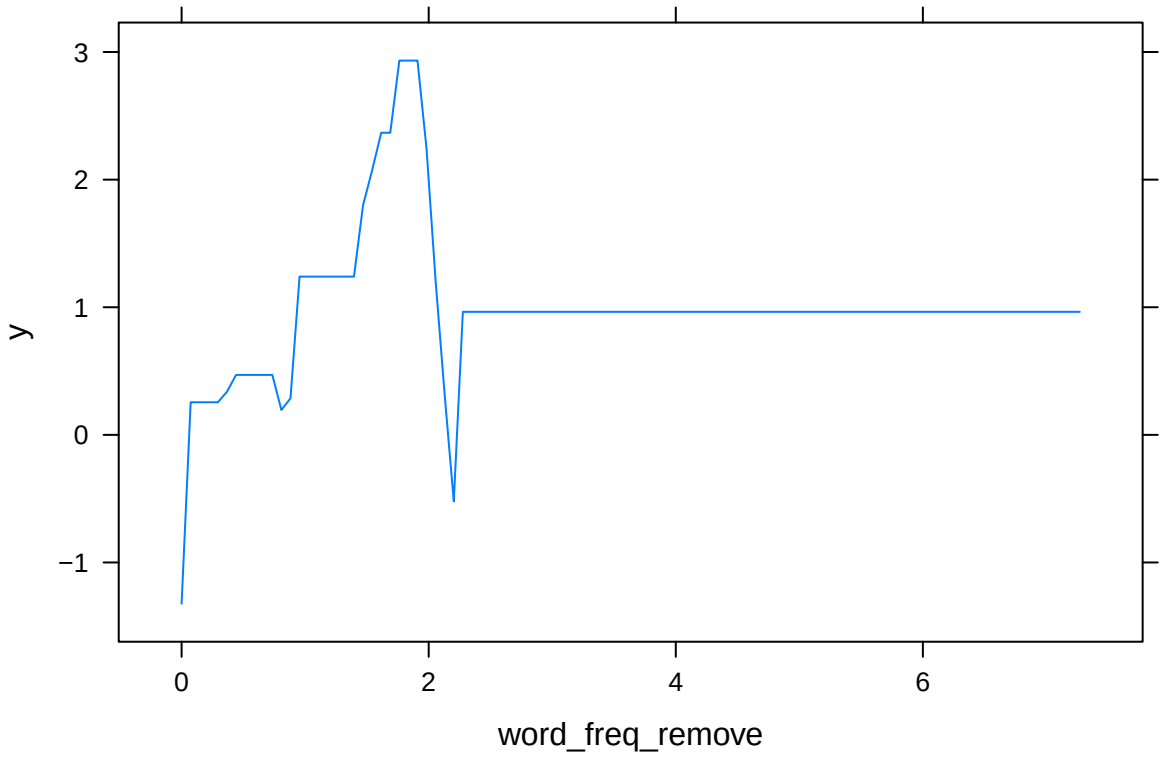

Figure 13: Partial dependence plot for third feature.

Solution.

```
# tuned decision tree
tree_pred = predict(optimal_tree, type = "class",  newdata = spam_test)
tree_misclass = mean(as.numeric(tree_pred)-1 != spam_test$spam)
# random forest
rf_pred = predict(rf_fit_optimal, type = "response", newdata = spam_test)
rf_misclass = mean(as.integer(rf_pred)-1 != spam_test$spam)
# boosting
boosting_prob = predict(gbm_fit_optimal, n.trees = optimal_num_trees,
                        type = "response", newdata = spam_test)
boosting_pred = as.numeric(boosting_prob > 0.5)
boosting_misclass = mean(boosting_pred != spam_test$spam)
# table
tibble(Method =
         c("tuned decision tree", "random forest", "boosting"),
       `Misclassification Rate` =
         c(tree_misclass, rf_misclass, boosting_misclass)) %>%
  kable(format = "latex", longtable = T,
        row.names = NA, booktabs = TRUE,
        col.names = NA,
        digits = 3,
        caption = "Misclassification errors of the three methods.") %>%
  kable_styling(latex_options = c("repeat_header"))
```

Table 3: Misclassification errors of the three methods.

Method	Misclassification Rate
tuned decision tree	0.083
random forest	0.047
boosting	0.048

See Table 3. Random forests performed the best and barely out performed boosting.

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_fit_tuned = rf_fit_optimal
gbm_fit_tuned = gbm_fit_optimal
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.

Solution.

```
# tibble
misclass_trees =
tibble(`Number of Trees` = 1:500,
       `Random Forest` = rf_test_err,
       `Boosting` = gbm_test_err) %>%
  pivot_longer(-`Number of Trees`, names_to = "type",
               values_to = "test_err")
# plot
misclass_trees %>%
  ggplot(aes(x = `Number of Trees`, y = test_err, color = type)) + geom_line() +
  labs(y = "Misclassification rate") +
  geom_hline(yintercept = min(gbm_test_err), linetype = "dashed", color = "red") +
  geom_hline(yintercept = min(rf_test_err), linetype = "dashed", color = "blue") +
  scale_y_log10() +
  theme_bw()
```

See Figure 14.

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?

Solution.

Random forests' misclassification error drops more quickly. This makes sense because boosting takes many trees to come close to a good prediction. Like the golfing example, it takes many putts to even come close to the hole, where as random forests from the start gets somewhere near the whole and then gets averaged.

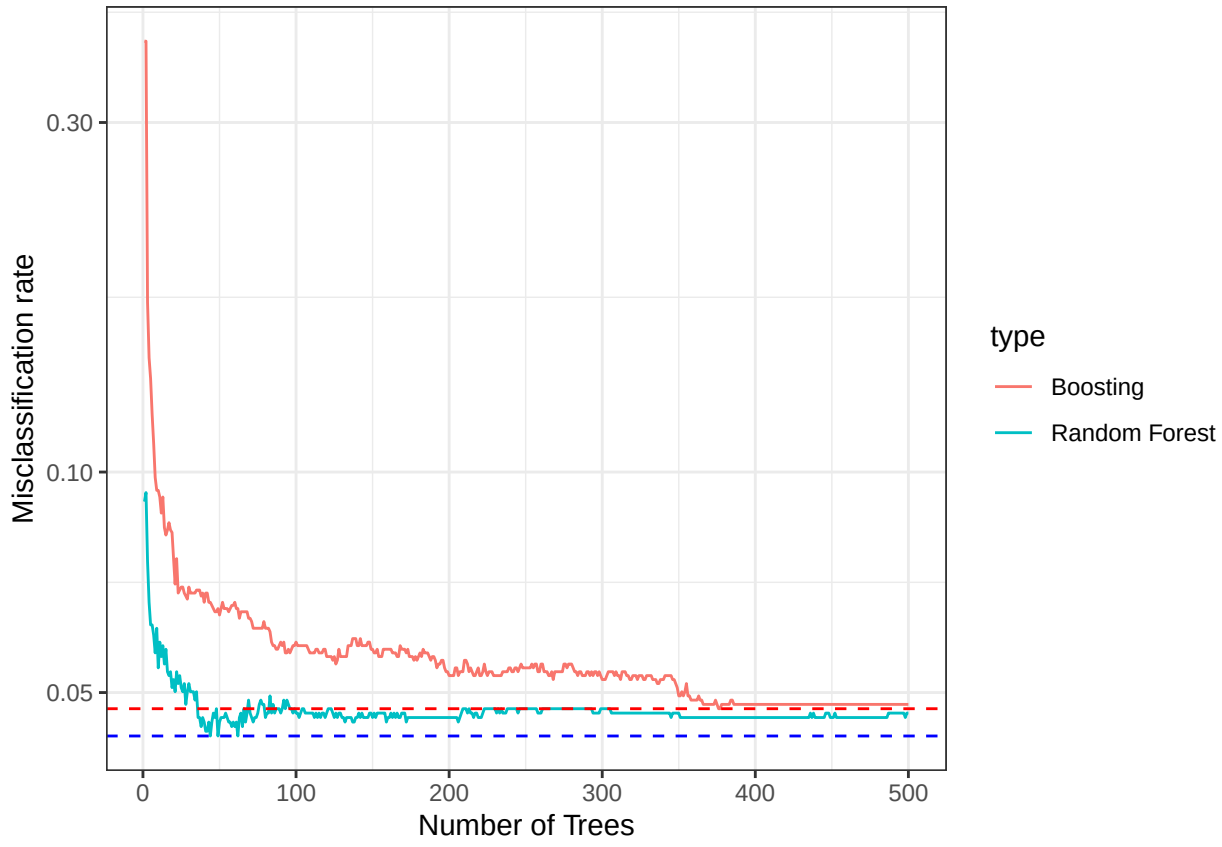

Figure 14: Misclassification errors of the random forest and boosting classifiers as a function of the number of trees