STAT 471: Homework 4

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)     # for printing tables
library(cowplot)        # for side by side plots
```

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?

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

About 0.397‘ 0.397 (or 39.7%) 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? **Assuming the test data contain**

the same class proportions, the misclassification error of a naive classifier that always predicts the majority class would be 0.397 (39.7%). This is because the majority class would be not spam such that we would be erroneously classifying the data that are spam (which proportionally makes up 0.397 (or 39.7%) of the data) as not spam.

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"))`).

```
# create a tibble containing the average frequencies of each word
avg_word_freq = spam_train %>%
  summarise_at(vars(starts_with("word_freq")), mean)
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.]

```
# create tibble containing each word and its average frequency
avg_word_freq_long = avg_word_freq %>%
  pivot_longer(cols = everything(),
               names_to = "word",
               names_prefix = "word_freq_",
               values_to = "avg_freq")
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?

```
# create a histogram of word frequencies
avg_word_freq_long %>%
  ggplot(aes(x = avg_freq)) +
  geom_histogram(bins = 20) +
  labs(x = "Word frequency", y = "Count") +
  theme_bw()
```

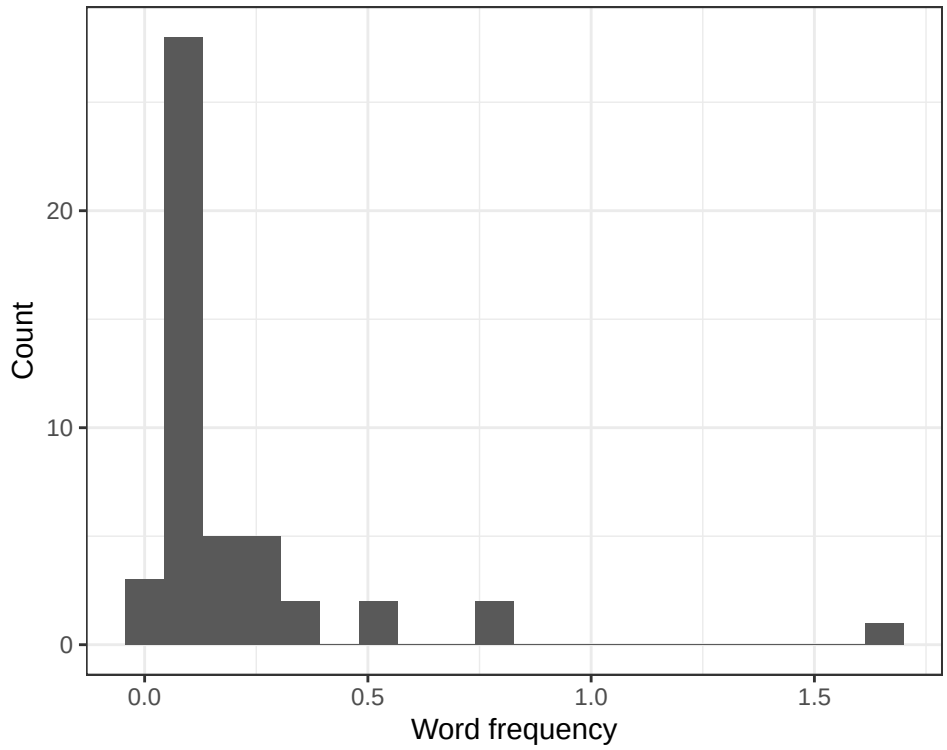

Figure 1: This is a histogram of the word frequencies in the data.

```
# determine the top three most frequent words
avg_word_freq_long %>%
  arrange(desc(avg_freq)) %>% head(3) %>%
  kable(format = "latex", row.names = NA,
                          booktabs = TRUE,
                          digits = 2,
                          col.names = c("Word",
                                        "Average frequency"),
```

```
                        caption = "These are the top three
      most frequent words.") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 1: These are the top three most frequent words.

Word	Average frequency
you	1.66
your	0.82
george	0.77

As we can see in Table 1, the top three most frequent words are "you", "your", and "george". It is possible for a word to have a frequency of more than 1, because the frequencies are percentages (as opposed to proportions between 0 and 1), 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.]

```
# create diff_avg_word_freq for diff between avg freq with one sequence of pipes
diff_avg_word_freq = spam_train %>%
  group_by(spam) %>%
  summarise_at(vars(starts_with("word_freq")), mean) %>%
  pivot_longer(word_freq_make:word_freq_conference,
               names_to = "word",
               names_prefix = "word_freq_",
               values_to = "avg_freq") %>%
  ungroup() %>%
  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)

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?

```
# create histogram of word frequency differences
diff_avg_word_freq %>%
  ggplot(aes(x = diff_avg_freq)) +
  geom_histogram(bins = 20) +
  labs(x = "Word frequency difference", y = "Count") +
  theme_bw()
```

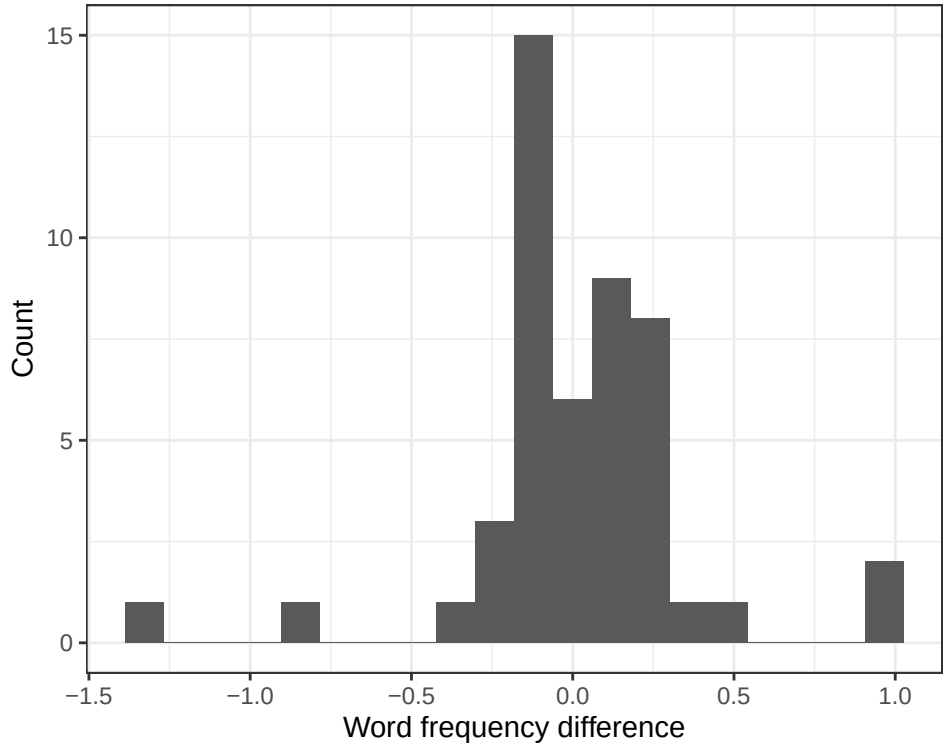

Figure 2: This is a histogram of the word frequency differences in the data, where the word frequency difference for a word is the average frequency in spam emails minus average frequency in non-spam emails.

```
# determine the three words most overrepresented in spam emails
diff_avg_word_freq %>%
  arrange(desc(diff_avg_freq)) %>% head(3) %>%
  kable(format = "latex", row.names = NA,
                          booktabs = TRUE,
                          digits = 2,
                          col.names = c("Word",
                                        "Difference in average frequency
```

```
                                    (spam - non-spam)"),
                        caption = "These are three words most overrepresented
      in spam emails based on the word frequency differences (i.e., average
      frequency in spam emails minus average frequency in
      non-spam emails).") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 2: These are three words most overrepresented in spam emails based on the word frequency differences (i.e., average frequency in spam emails minus average frequency in non-spam emails).

Word	Difference in average frequency (spam - non-spam)
you	1.01
your	0.98
free	0.45

```
# determine the three words most underrepresented in spam emails
diff_avg_word_freq %>%
  arrange(diff_avg_freq) %>% head(3) %>%
  kable(format = "latex", row.names = NA,
                        booktabs = TRUE,
                        digits = 2,
                        col.names = c("Word",
                                      "Difference in average frequency
                                      (spam - non-spam)"),
                        caption = "These are three words most underrepresented
                        in spam emails based on the word frequency differences
                        (i.e., average frequency in spam emails minus average
                        frequency in non-spam emails).") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 3: These are three words most underrepresented in spam emails based on the word frequency differences (i.e., average frequency in spam emails minus average frequency in non-spam emails).

Word	Difference in average frequency (spam - non-spam)
george	-1.28
hp	-0.85
hpl	-0.41

As we can observe in Table 2, the three words that are most overrepresented in spam emails are "you", "your", and "free". As we can observe in Table 3, the three words that are most underrepresented in spam emails are "george", "hp", and "hpl". These make sense since we may expect that spam emails would try to get the attention of the receiver by using words like "you" and "your" that are directive. Also, the word "free" is enticing, such that it would make sense for spammers to include this in emails to get people to read the emails and perhaps be motivated to click on URLs. It would make sense that words like "george", "hp", and "hpl" would be underrepresented in spam emails since they may perhaps suggest relevant work emails that would not be spam (as the company is HP, the employee who helped compile the data worked in the HP Laboratory, and George is that employee working at the company).

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.

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

rpart.plot(tree_fit) # plot tree
```

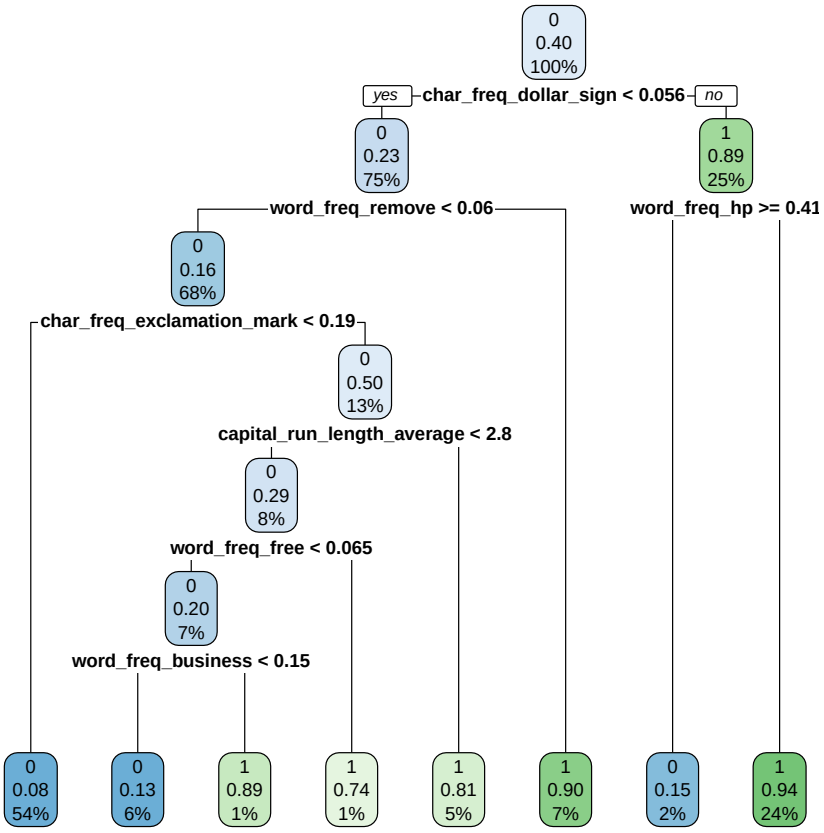

Figure 3: This is a plot of the classification tree we fit with splits based on the Gini index, with default control parameters.

In Figure 3, we can observe the plot of the classification tree.

ii. (2 points) How many splits are there in this tree? How many terminal nodes does the tree have? **As in Figure 3, there are seven splits. The tree has eight terminal nodes (which we can see in the plot but also we know to be one more than the number of splits).**

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? **We can observe in Figure 3 that the terminal node that has the largest fraction of spam observations is on the far right in the tree plot (with 0.94 spam probability). The sequence of splits that leads to this terminal node is the following: 1. feature: the frequency of the dollar sign character, split point: frequency < or >= 0.056, direction: frequency is >= 0.056 (i.e., right); 2. feature: the frequency of the word "hp", split point: frequency >= or < 0.41, direction: frequency is < 0.41 (i.e., right).**

This sequence of splits makes some sense as flagging likely spam emails, as we may expect spam emails to have some monetary relation (e.g., offering/suggesting monetary reward) and not see the word "hp" frequently (as emails with this word would likely reflect work emails, i.e., emails less likely to be spam). The fraction of spam observations that this node has is 0.94 (as the fraction of observations in this node that are spam, since this value reflects the probability that the observations in the node are spam and the observations are independent). The fraction of the training observations that are in this node is 0.24.

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? **In terms of the bias-variance tradeoff, it would be a problem if the default tree were not large enough because then bias would be rather high, given that the fewer splits are leading to fewer regions defined. (Pruning back a tree leads to less complexity (less ability to capture the underlying trend), which would only result in more 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? **By definition, `minsplit` is the minimum number of observations that must exist in a node in order for a split to be attempted. Thus, `minsplit` being 0, 1, or 2 will lead to the deepest possible tree. We would attempt the most splits when the parameter is at a value of 2 since when we have 2 observations, we can split this into one observation in each resulting bucket. We could also consider 1 (or even 0) to be the minimum, serving as a lower bound, as the results when we use either of these values for the parameter yield the same result. (Once we have single node, there is nothing to split.) (Since these two values achieve the same results, we use a value of 1 below.)**

The value of `minbucket` that will lead to the deepest possible tree is 1. This is because by definition, `minbucket` is the minimum number of observations in any terminal leaf node. Thus, with the parameter set to 1, we have only one observation in each terminal node, reflecting the greatest possible splits of the tree. Effectively, the larger the `minsplit` and `minbucket`, the fewer nodes there will be in the tree.

The value of `cp` (complexity parameter) that will lead to the deepest possible tree is 0. This is because by definition, `cp` dictates that any split that does not decrease the overall lack of fit by a factor of cp is not attempted. Thus, if we set the parameter as 0, we are not constraining/limiting the splits attempted.

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)

deepest_tree_fit = rpart(spam ~ .,
                         method = "class",
                         parms = list(split = "gini"),
                         control = rpart.control(minsplit = 1,
                                                 minbucket = 1,
                                                 cp = 0),
                         data = spam_train)

# print the CP table for this tree
cp_table = deepest_tree_fit$cptable %>% as_tibble()
cp_table %>% kable(format = "latex", row.names = NA,
                   booktabs = TRUE, digits = 5,
                   col.names = c("CP",
                      "Number of splits",
                      "Relative error",
                      "CV error",
                      "CV std"),
                      caption = "This is the CP table for this
                   deepest tree.") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 4: This is the CP table for this deepest tree.

CP	Number of splits	Relative error	CV error	CV std
0.49343	0	1.00000	1.000	0.0222
0.14450	1	0.50657	0.530	0.0185
0.04187	2	0.36207	0.396	0.0165
0.02791	4	0.27833	0.303	0.0148
0.01724	5	0.25041	0.274	0.0142
0.01149	6	0.23317	0.255	0.0137
0.00821	7	0.22167	0.241	0.0134
0.00575	8	0.21346	0.229	0.0131
0.00493	10	0.20197	0.222	0.0129
0.00411	11	0.19704	0.221	0.0129
0.00369	12	0.19294	0.219	0.0128
0.00328	14	0.18555	0.223	0.0129
0.00246	18	0.17241	0.216	0.0127
0.00164	30	0.14286	0.210	0.0126
0.00137	45	0.11823	0.204	0.0124
0.00123	53	0.10427	0.205	0.0124
0.00109	59	0.09442	0.205	0.0124
0.00103	62	0.09113	0.208	0.0125
0.00082	66	0.08703	0.208	0.0125
0.00055	134	0.03120	0.211	0.0126
0.00047	138	0.02874	0.218	0.0128
0.00041	145	0.02545	0.224	0.0129
0.00000	205	0.00082	0.223	0.0129

In Table 4, we can observe the CP table for this deepest tree with the `minsplit`, `minbucket`, and `cp` parameters set to 1, 1, and 0 respectively.

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 in the sequence of trees produced in part iii. The biggest tree has 205 splits (and so 206 terminal nodes). It has about 15 average observations per terminal node as computed below. It is not one because it is certainly possible (and thus observed here) for all regions to be entirely pure (homogeneous) before increasing the splits all the way down to one such that increasing the splits further (such that we continue to decrease the number of observations in terminal nodes) will not decrease the RSS further.

```
average_obs_per_term_node = nrow(spam_train) / 206
round(average_obs_per_term_node)
```

```
## [1] 15
```

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()`.

```
# produce CV plot based on info in CP table
cp_table %>%
```

```
  filter(nsplit >= 2) %>%
  ggplot(aes(x = nsplit+1, y = xerror,
             ymin = xerror - xstd, ymax = xerror + xstd)) +
  scale_x_log10() +
  geom_point() + geom_line() +
  geom_errorbar(width = 0.25) +
  xlab("Number of terminal nodes on log scale") + ylab("CV error") +
  geom_hline(aes(yintercept = min(xerror)), linetype = "dashed") +
  theme_bw()
```

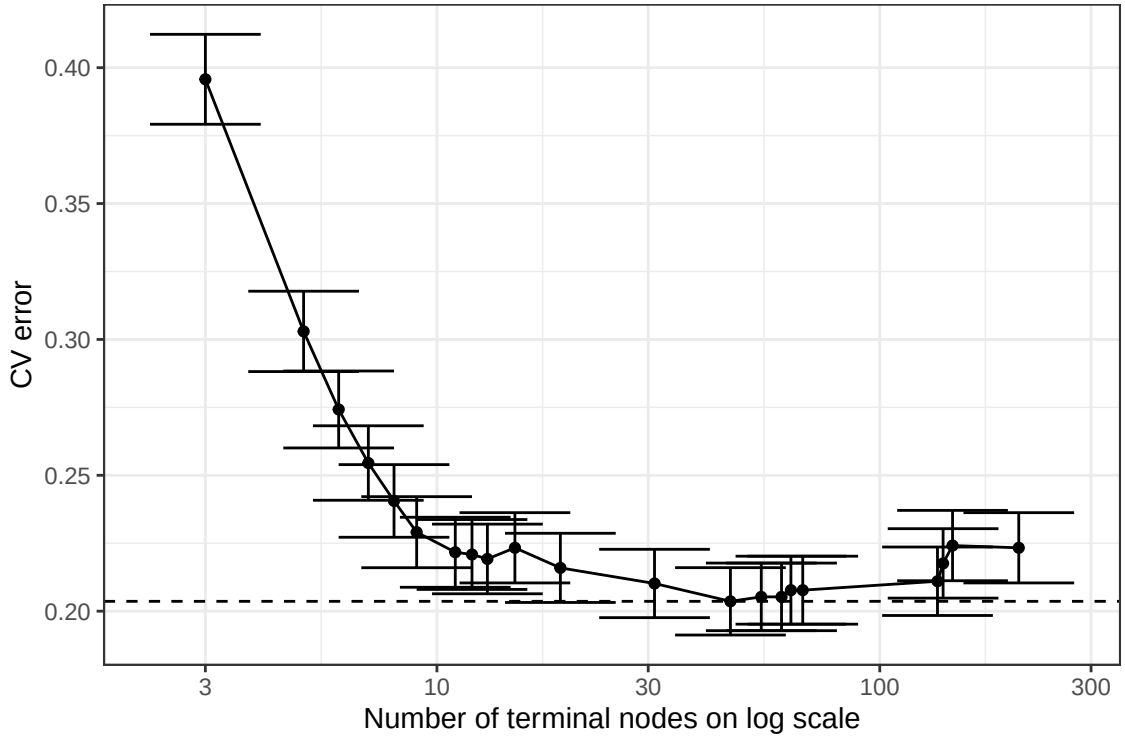

Figure 4: This is the CV plot based on the CP table printed above. Only trees with number of splits at least two are plotted, and the x-axis is on a log scale.

In Figure 4, we can observe the CV plot based on the information in the CP table printed above (i.e., in Table 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?

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

```
## [1] 18
```

Using the one-standard-error rule, the optimal tree has 19 terminal nodes (since there are 18splits and one more terminal node than number of splits so 18 + 1 =19‘). This is larger/more than the number of terminal nodes in the default tree above, in which there had been 8 terminal nodes.

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, 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)

num_features = ncol(spam_train) - 1
mtry = floor(sqrt(num_features))
mtry
```

```
## [1] 7
```

```
mtry_verified = rf_fit$mtry
mtry_verified
```

```
## [1] 7
```

We know that for random forests, the default value of `mtry` is the square root of the number of features. Thus, in this case, the value of `mtry` used was floor(sqrt(57)) = 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?

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

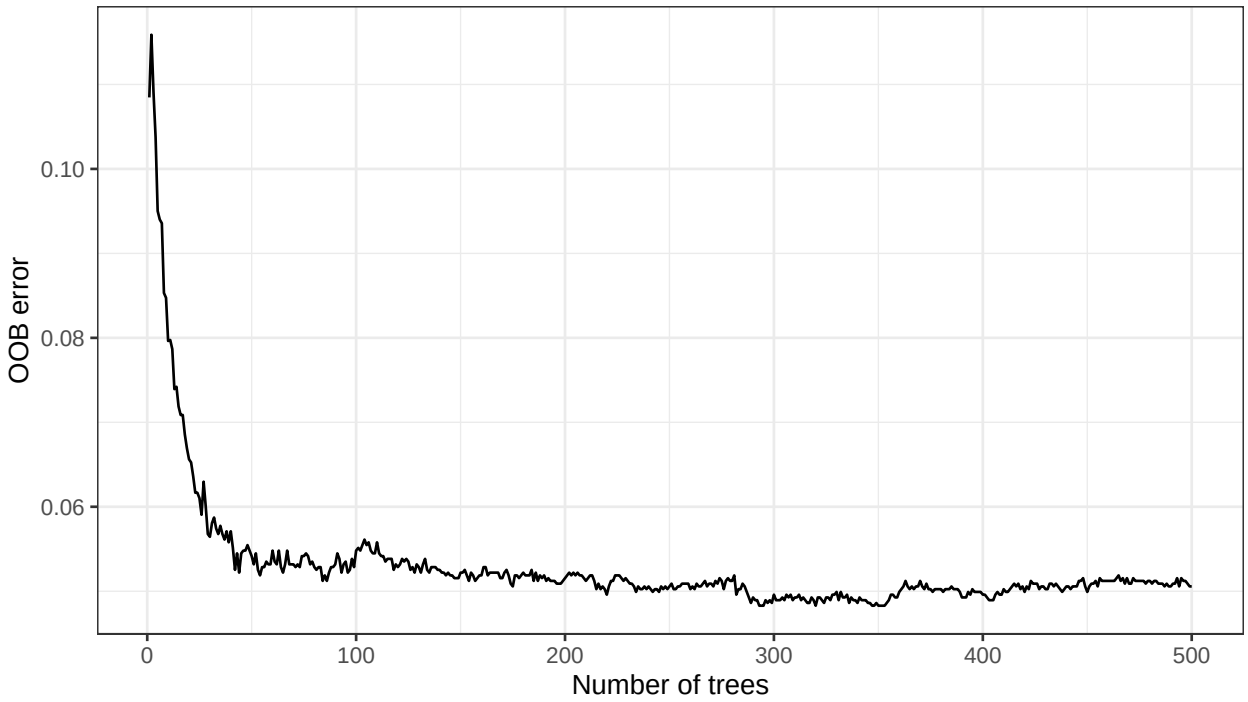

Figure 5: This is the plot of the OOB error as a function of the number of trees for the random forest fit.

In Figure 5, we have the OOB error plotted as a function of the number of trees. We see that this error stays flat as soon as the number of trees is large enough. That is, we can see that the OOB error has certainly stabilized by 500 trees, and the OOB error seems to stabilize for roughly 100 trees, if we are to consider a lower number of trees for which it has stabilized.

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.

```
set.seed(1)
time_importance_false = system.time(randomForest(factor(spam) ~ .,
                                                 importance = FALSE,
                                                 data = spam_train))
time_importance_false
```

```
##    user  system elapsed
##   4.746   0.067   4.817
```

Thus, the elapsed time taken to train the random forest with default parameter settings, with `importance = FALSE` is 4.817 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`?

```
set.seed(1)
time_importance_true = system.time(randomForest(factor(spam) ~ .,
                                                importance = TRUE,
                                                data = spam_train))
time_importance_true
```

```
##    user  system elapsed
##  12.170   0.052  12.231
```

Thus, the elapsed time taken to train the random forest with default parameter settings except `importance = TRUE` is 12.231 seconds. Thus, the computation, is around 2.539 times faster when `importance = FALSE`.

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"]`

```
set.seed(1)
# record time for training five random forests
time_100 = system.time(randomForest(factor(spam) ~ ., importance = FALSE,
                                    ntree = 100, data = spam_train))["elapsed"]
time_200 = system.time(randomForest(factor(spam) ~ ., importance = FALSE,
                                    ntree = 200, data = spam_train))["elapsed"]
time_300 = system.time(randomForest(factor(spam) ~ ., importance = FALSE,
                                    ntree = 300, data = spam_train))["elapsed"]
time_400 = system.time(randomForest(factor(spam) ~ ., importance = FALSE,
                                    ntree = 400, data = spam_train))["elapsed"]
time_500 = system.time(randomForest(factor(spam) ~ ., importance = FALSE,
                                    ntree = 500, data = spam_train))["elapsed"]
```

```
# create tibble of ntree and random forest train fit time
rf_times = tibble(ntree = c(100, 200, 300, 400, 500),
                  elapsed_time = c(time_100, time_200, time_300,
                                   time_400, time_500))

# plot the time against ntree
rf_times %>% ggplot(aes(x = ntree, y= elapsed_time)) +
  geom_line() +
  labs(x = "Number of trees", y = "Elapsed time for training model") +
  theme_bw()
```

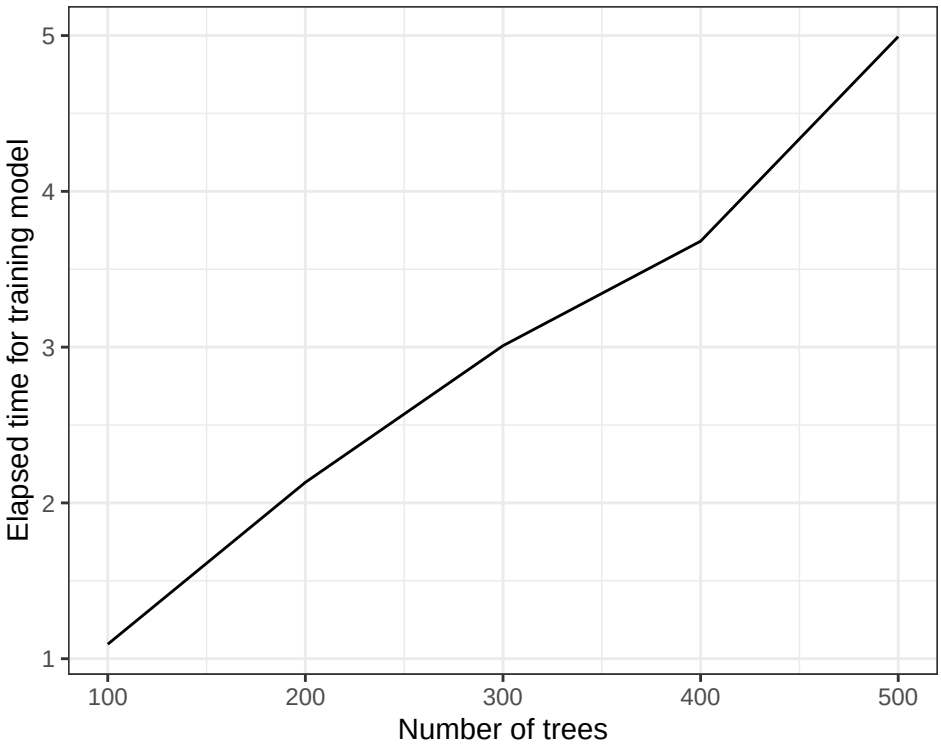

Figure 6: This is the time taken to fit random forest models against the varying values of the parameter ntree (i.e., 100, 200, 300, 400, 500) for those models.

In Figure 6, we have the time it takes to train each model plotted against against the number of trees for that model (i.e., 100, 200, 300, 400, 500).

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?

In Figure 6, we can observe that as the number of trees is increased (i.e., as the value for the parameter `ntree` is larger), the runtime for training the random forest increases. That is, we can observe a positive relationship between the number of trees and the runtime for training the model. It makes sense in the context of the training algorithm for random forests, as we have to create more trees (for which we will average their predictions together), which takes time. From lecture, we have B individual trees (based on bootstrapping the data B times such that we have B bootstrap samples; for each sample (for each of the B bootstrap samples we extract from training data) we create a tree), and each individual tree has a prediction, and the final prediction is the average of the predictions of the individual trees. That is, for each bootstrap sample, we grow a decision tree and at each step, randomly sample m candidate variables to split on, such that when we train each of these trees, at each split point, we randomly select a subset of the m features to split on and split on the best feature among this subset. We grow the trees until we grow them out to a certain stopping criterion. When we predict, we take all of these different trees, we make predictions for each, and then we aggregate. At training time, we need to build up each of these trees, taking a boostrap sample of the data and building a tree, so when we have more trees, it makes sense for this to take more time. (Also note that if we use too few trees, we do not achieve variance reduction we set out to achieve, as then there is not really any averaging going on. Thus with few trees we will have high error and as we start using more trees, there is more averaging kicking in such that at some point we have averaged enough trees that our prediction performance is improved.)

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?

To this end, it makes sense to tune the random forest with `importance = FALSE` since we can avoid the extra time necessary to compute the the OOB-based variable importance measure, as this is only computed if `importance = TRUE` is specified. Based on OOB error plot from above, a reasonable number of trees to grow without significantly compromising prediction accuracy is around 100 trees, as at this point, the OOB error has stabilized (and is a lower value of trees such that we reduce computational demands without significantly compromising prediction accuracy).

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.)

```
num_minutes = time_500 * num_features / 60
num_minutes
```

```
## elapsed
##    4.74
```

There are 57 features in the data, so there are 57 possible values of m. It takes 4.993 seconds to train a random forest with 500 trees, based on the elapsed time determined for 500 trees when `importance = FALSE` (and as seen in Figure 6. Thus, to train a random forest with 500 trees for every possible value of m would take 4.743 minutes. We can compute this, as seen in the code chunk above, where we obtain that the time to train a random forest with 500 trees for every possible value of m is 4.743 minutes.

```
num_m_to_try = as.integer(15 / time_100)
num_m_to_try
```

```
## [1] 13
```

Supposing that we only have enough patience to wait about 15 seconds to tune our random forest, and we use the reduced number of trees from part i (i.e., 100 trees), we can observe the following: Training a random forest with 100 trees takes 1.093 seconds (as seen in Figure 6). Thus, we can afford to try about 13 values of m. We can compute this, as seen in the code chunk above, where we obtain that the number of values of m we can afford to try in this time with the particular value for the parameter `ntree` is 13.

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?

```
set.seed(1)
# tune the random forests with different values of m
num_m = as.integer(max(num_m_to_try, 5)) # calculate at least five values of m
mvalues = as.integer(seq.int(from = 1, to = 57, length.out = num_m))
oob_errors = numeric(length(mvalues))
ntree = 100
for(idx in 1:length(mvalues)){
```

```
  m = mvalues[idx]
  rf_fit = randomForest(factor(spam) ~ ., mtry = m, data = spam_train)
  oob_errors[idx] = rf_fit$err.rate[ntree]
}
```

```
# plot OOB error versus m
m_and_oob_errors = tibble(m = mvalues, oob_err = oob_errors)
m_and_oob_errors %>%
  ggplot(aes(x = m, y = oob_err)) +
  geom_line() +
  geom_point() +
  scale_x_continuous(breaks = mvalues) +
  labs(x = "Value for m", y = "OOB error") +
  theme_bw()
```

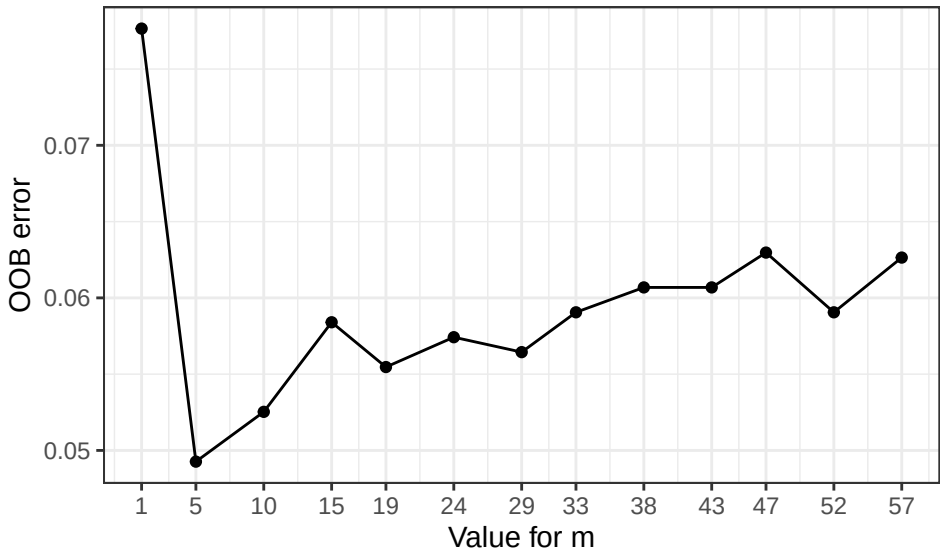

Figure 7: This is the plot of OOB error versus m for different values of m.

```
# extract m corresponding to min value of OOB error
best_m = m_and_oob_errors %>% arrange(oob_errors) %>% head(1) %>% pull(m)
```

In Figure 7, we can observe the plot of OOB error versus m. The best value of m is 5 (the minimum of the curve). Compared to the default value of m, it is higher than the default value of m (which was 7 in this context). (That is, the best value of m is 0.714 times higher than 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) ~ ., mtry = best_m, ntree = 500,
                            importance = TRUE, data = spam_train)
```

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

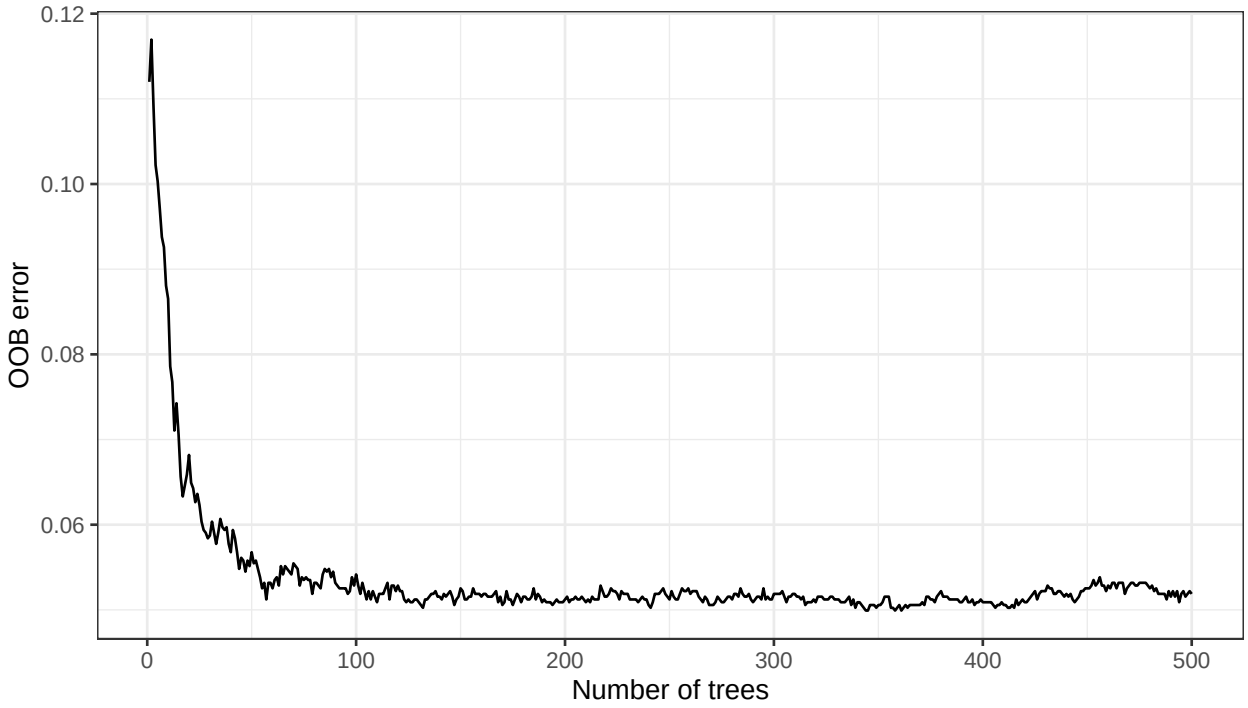

Figure 8: This is the plot of the OOB error as a function of the number of trees for the tuned random forest fit.

In Figure 8, we have the OOB error plotted as a function of the number of trees. As seen in the plot, it appears that the error has flattened out.

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.

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

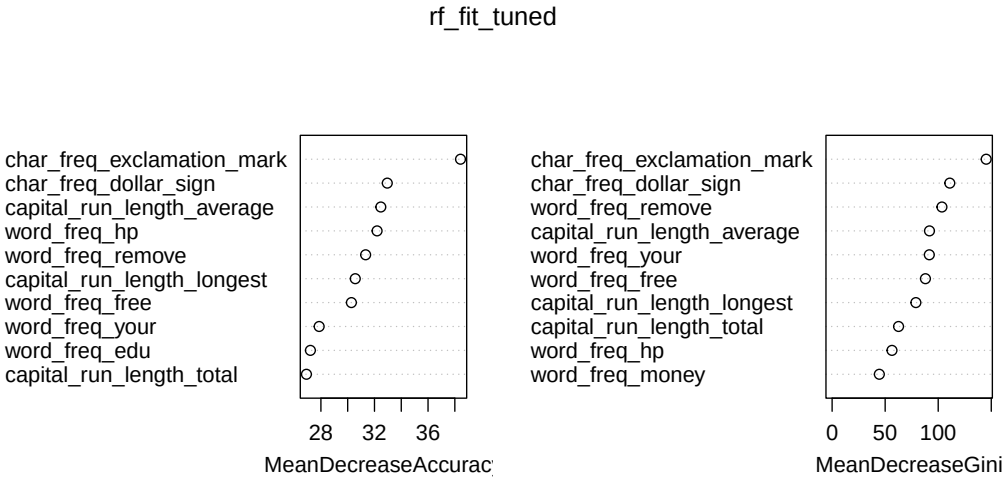

Figure 9: This is the variable importance plot for the random forest trained on the optimal value of m.

In Figure 9, we have the the variable importance plot for the random forest trained on the optimal value of m.

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.

In order, the (ordered) top three features for the left plot in Figure 9 are `char_freq_exclamation_mark`, and `char_freq_dollar_sign`, and `capital_run_length_average` (by the metric Mean decrease accuracy). For the right plot in Figure 9, the (ordered) top three features are `char_freq_exclamation_mark`, `char_freq_dollar_sign`, and `word_freq_remove` (by the metric Mean decrease Gini).

Examining the left list, `word_freq_edu` does not appear in the right list, and examining the right list, `capital_run_length_longest` and `word_freq_money` do not appear in the left list).

We can consider `char_freq_dollar_sign` (one of these top features), which is a feature in the top three for both lists, and the top feature for the Mean decrease Gini metric (corresponding to the right list). We might expect it to be predictive of spam since dollar signs are typically found in spam emails. Spam emails may frequently contain fake offers and play on one's desires, potentially suggesting that one has won a lot of money or can click a link to enter to win a monetary prize, for example. Thus, we would expect dollar signs may be a frequent occurrence in spam emails. That is, we 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.

```
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?

```
# 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)
)

# plot CV errors
cv_errors %>%
  ggplot(aes(x = ntree, y = cv_err, colour = factor(depth))) +
  # add horizontal dashed lines at the minima of the three curves
  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") +
  geom_line() +
  # set colors to match horizontal line minima
  scale_color_manual(labels = c("1", "2", "3"),
                     values = c("red", "green", "blue")) +
  labs(x = "Number of trees", y = "CV error", colour = "Interaction depth") +
  theme_bw()
```

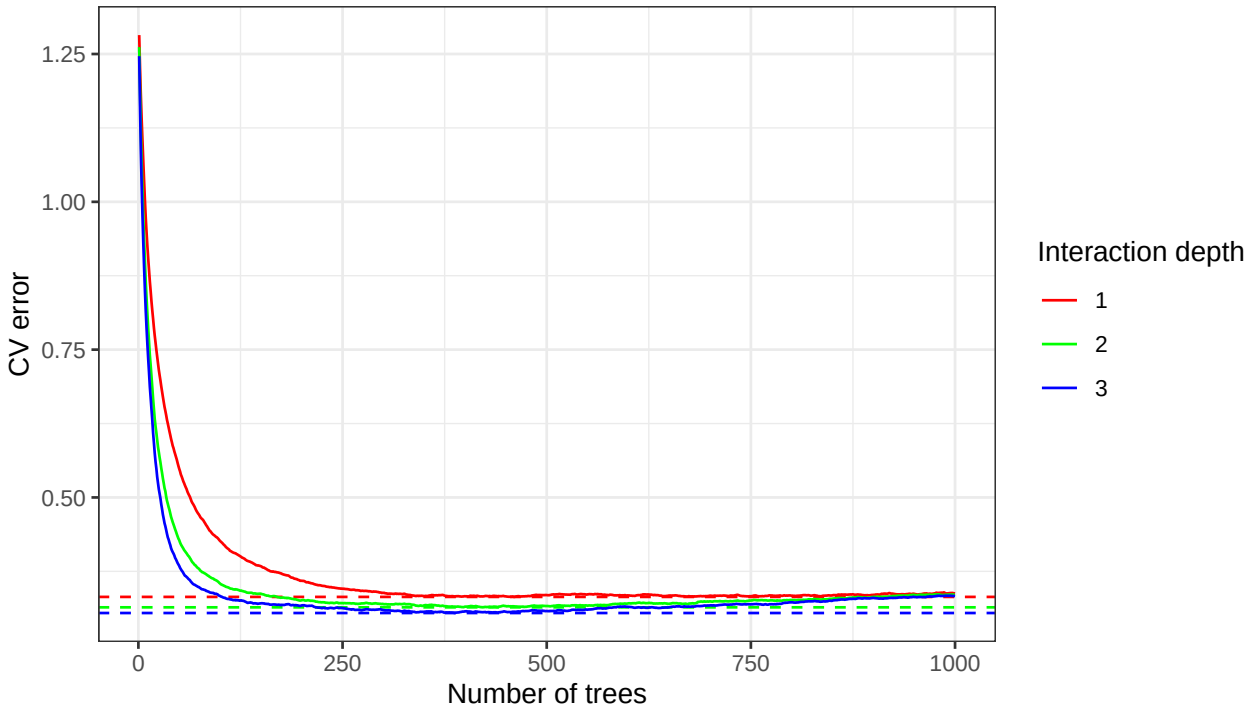

Figure 10: This is a plot of the CV errors against the number of trees for each interaction depth, i.e., 1, 2, and 3. Also plotted are horizontal dashed lines at the minima of these three curves.

```
gbm_fit_tuned = gbm_fit_3
optimal_num_trees = gbm.perf(gbm_fit_3, plot.it = FALSE)
optimal_num_trees
```

```
## [1] 387
```

From Figure 10, we can see that the optimal interaction depth is 3 and the corresponding 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?

```
# print first ten rows of relative influence table for optimal boosting model
summary(gbm_fit_tuned, n.trees = optimal_num_trees, plotit = FALSE) %>%
  head(10) %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 2,
        col.names = c("Variable", "Relative influence"),
        caption = "These are the first ten rows of the relative influence
        table for the optimal boosting model above.") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 5: These are the first ten rows of the relative influence table for the optimal boosting model above.

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

NEED TO EDIT

From Table 5, we see that the top three features are `char_freq_exclamation_mark`, `char_freq_dollar_sign`, and `word_freq_remove`. These top three features for the boosting model align with the top three features 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.

```
p1 = plot(gbm_fit_tuned,
     i.var = "char_freq_exclamation_mark",
     n.trees = optimal_num_trees,
     type = "response")

p2 = plot(gbm_fit_tuned,
     i.var = "char_freq_dollar_sign",
     n.trees = optimal_num_trees,
     type = "response")

p3 = plot(gbm_fit_tuned,
     i.var = "word_freq_remove",
     n.trees = optimal_num_trees,
     type = "response")

# use cowplot to concatenate the two plots
plot_grid(p1, p2, p3, align = "h")
```

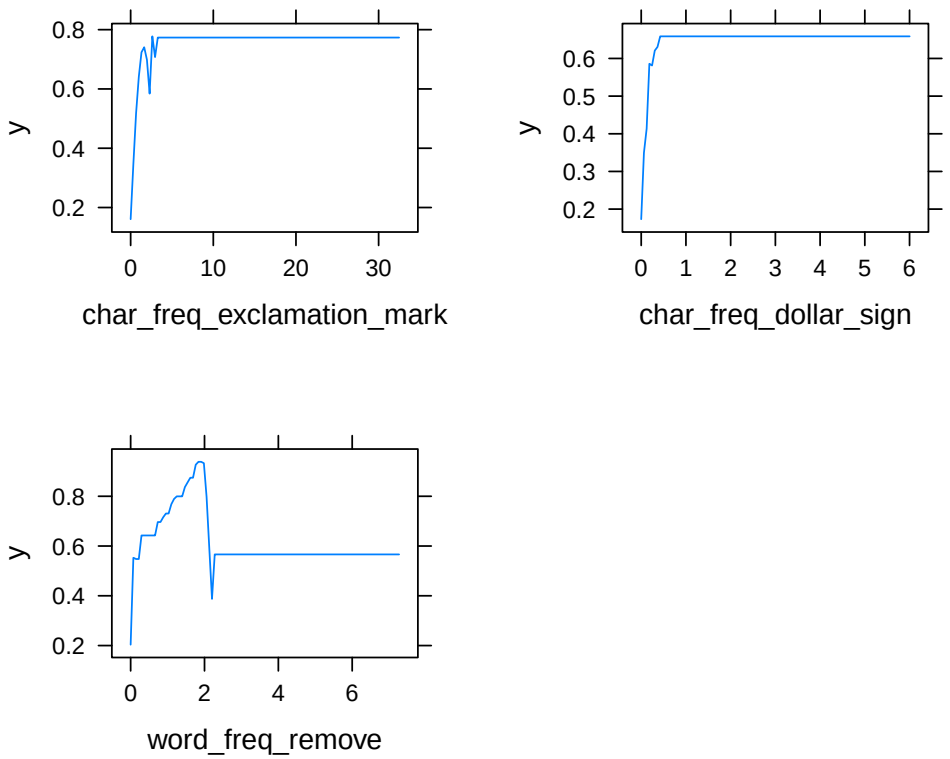

Figure 11: These are partial dependence plots for the top three features based on relative influence. The plot for the character frequency of excalamation marks appears in the top left, the plot for the character frequency of dollar signs appears in the top right, and the word frequency of remove appears in the bottom plot.

In Figure 11, we have partial dependence plots for the top three features (i.e., for the features `char_freq_exclamation_mark`, `char_freq_dollar_sign`, and `word_freq_remove`) based on relative influence.

`char_freq_exclamation_mark` and `char_freq_dollar_sign` have increasing relationships with spam, which makes sense, as exclamation marks and dollar signs are typically found in spam emails. The relationship between `word_freq_remove` and spam appears to be more complex, with intermediate word frequencies being most associated with spam. It is 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?

```
# compute test misclassification error of the tuned decision tree
dt_probabilities = predict(optimal_tree, newdata = spam_test, type = "class")
decision_tree_error = mean(dt_probabilities != spam_test$spam)

# compute test misclassification error of the random forest
rf_predictions = predict(rf_fit_tuned,
                         type = "response",
                         newdata = spam_test)
rf_error = mean(rf_predictions != spam_test$spam)
```

```r
# compute test misclassification error of the boosting classifier
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)
```

```r
# create table of these three model test errors
error_for_models = tribble(
  ~model, ~error,
  #------/-------
  "Decision tree", decision_tree_error,
  "Random forest", rf_error,
  "Boosting", gbm_error,
  )

# print these test errors in table
error_for_models %>% kable(format = "latex", row.names = NA,
                           booktabs = TRUE,
                           digits = 5,
                           col.names = c("Model type",
                                         "Misclassification error"),
                           caption = "These are the test misclassification errors
                           of the tuned decision tree, random forest, and boosting
                           classifiers.") %>%
  kable_styling(position = "center") %>%
  kable_styling(latex_options = "HOLD_position")
```

Table 6: These are the test misclassification errors of the tuned decision tree, random forest, and boosting classifiers.

Model type	Misclassification error
Decision tree	0.1022
Random forest	0.0475
Boosting	0.0495

We can observe in Table 6 that the random forest method performs best, as it has the lowest test error.

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.

```r
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.

```
rf_test_err_tibble = tibble(ntree = 1:500, rf_test_err)
gbm_test_err_tibble = tibble(ntree = 1:500, gbm_test_err)
test_errors = inner_join(rf_test_err_tibble,
                         gbm_test_err_tibble,
                         by = "ntree")

test_errors %>% ggplot(aes(x = ntree)) +
  # add misclassification error curves for random forest and boosting
  geom_line(aes(y = rf_test_err, color = "Random forest")) +
  geom_line(aes(y = gbm_test_err, color = "Boosting")) +
  # add horizontal line at misclassification error of optimal pruned tree
  geom_hline(yintercept = decision_tree_error,
             linetype = "dashed",
             color = "blue") +
  scale_y_log10() +
  scale_color_manual(labels = c("Random forest", "Boosting"),
                     values = c("red", "green")) +
  labs(x = "Number of trees",
       y = "Misclassification error on log scale",
       color = "Model") +
  theme_bw()
```

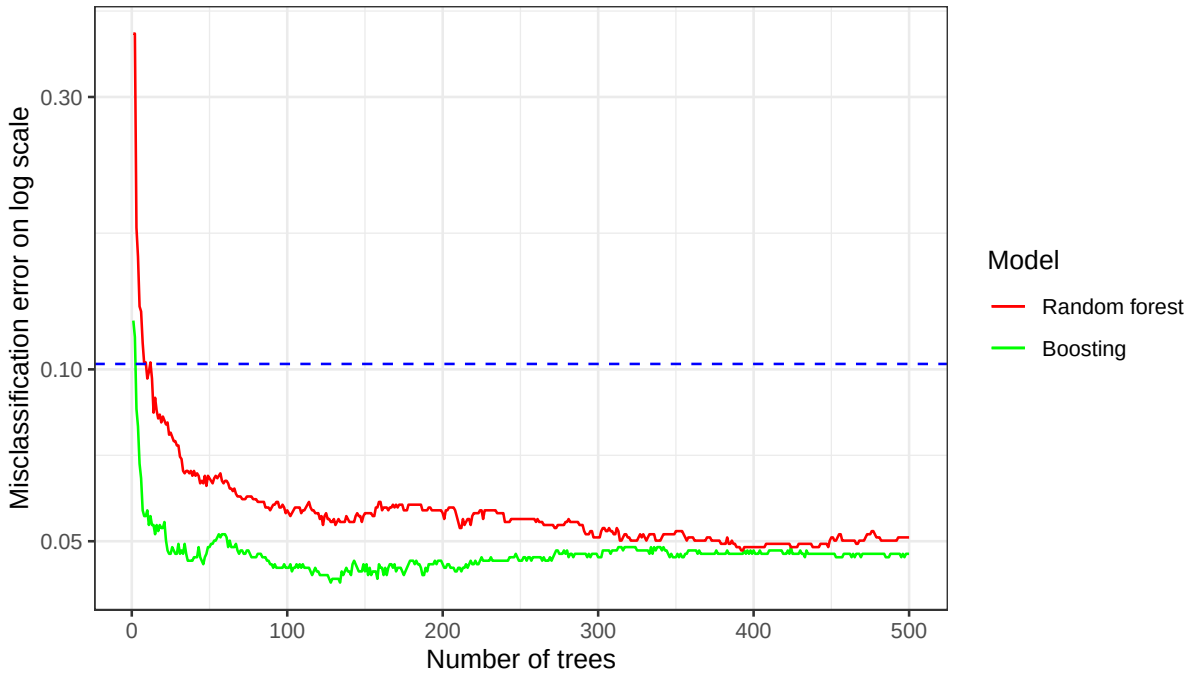

Figure 12: This is a plot showing the misclassification errors of the random forest and boosting classifiers as a function of the number of trees, as well as a horizontal line at the misclassification error of the optimal pruned tree.

In Figure 12, we have a plot showing the misclassification errors of the random forest and boosting classifiers as a function of the number of trees. There is additionally a horizontal line at the misclassification error of the optimal pruned tree, and the y-axis is on a log scale.

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?

We can observe from Figure 12 that between random forests and boosting, the random forest method's misclassification error drops more quickly as a function of the number of trees. This makes sense because boosting is a slow learning method (iterative, slowly moving towards target hole in the golf analogy). Boosting is based on weak learners; in terms of decision trees, weak learners are shallow trees (where they can even be decision stumps), and boosting aggregates the output from many models. On the other hand, the random forest method uses fully grown decision trees. (The trees are made to be uncorrelated to maximize the decrease in variance. The algorithm does not reduce bias, so there is the need for large, unpruned trees, so that the bias starts off as low as possible.) In contrast to boosting, which is sequential in nature, random forest grows trees in parallel, creating independent, parallel decision trees. (Moreover, random forest works better with a few deep decision trees and has a short fit time, whereas boosting builds trees in a successive manner where each tree improves upon mistakes of previous trees (i.e., based on the residuals), works better with multiple, shallow decision trees, and has a long fit time.) (With boosting, we take tentative steps; we are around the target for random forest, whereas boosting takes more steps to get closer to the goal (when we think of the golf analogy).)