STAT 471: Homework 4

Name

Due: November 17, 2021 at 11:59pm

Contents

Instructions

Setup

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

Collaboration

The collaboration policy is as stated on the Syllabus:

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

In accordance with this policy,

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

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

Writeup

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

Programming

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

Grading

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

Submission

Compile your writeup to PDF and submit to Gradescope.

Case Study: Spam Filtering

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

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

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

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

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

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

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

First, let's load a few libraries:

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

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

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

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

```
# extract training data
spam_train = spam_data %>%
  filter(test == 0) %>%
  select(-test) %>%
  mutate(spam = as.factor(spam)) #change spam to factor

# extract test data
spam_test = spam_data %>%
  filter(test == 1) %>%
  select(-test)  %>%
  mutate(spam = as.factor(spam)) #change spam to factor
```

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

First, let's explore the training data.

1.1 Class proportions (3 points)

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

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

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

39.7% of training observations are spam

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

1.2 Exploring word frequencies (15 points)

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

1.2.1 Overall word frequencies (8 points)

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

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

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

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

```
print((avg_word_freq))
```

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

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

```r
avg_word_freq_long = avg_word_freq %>%
  pivot_longer(cols = everything(), names_prefix = "word_freq_", names_to = 'word',
               values_to = 'avg_freq')
print(avg_word_freq_long)
```

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

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

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

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

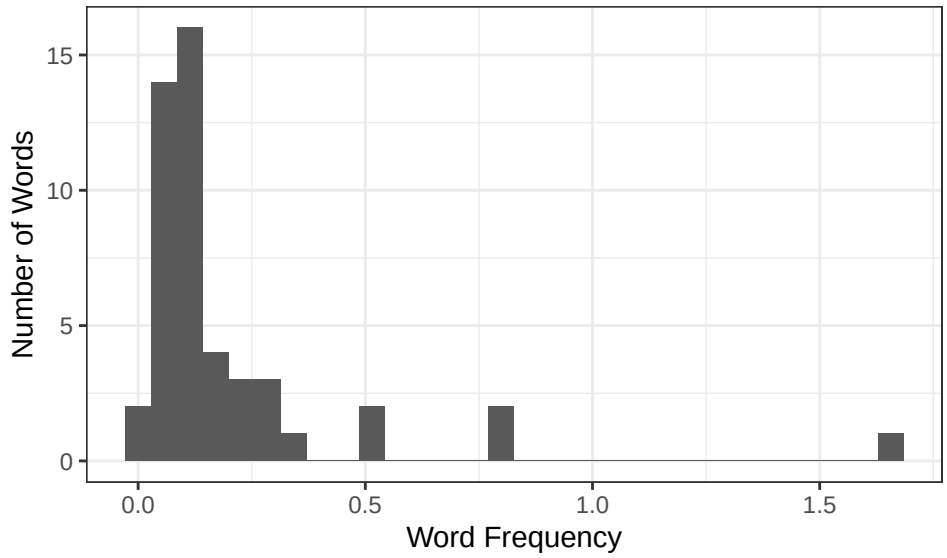


Figure 1: Distribution of word frequency

```
# find top 3 frequency
top_3_freq = avg_word_freq_long %>%
  arrange(desc(avg_freq)) %>%
  head(3)
```

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

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

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

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

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

```
diff_avg_word_freq = spam_train %>%
  group_by(spam) %>%
  summarise_at(vars(starts_with("word_freq")), funs(mean)) %>%
  pivot_longer(cols = -c(spam), names_prefix = "word_freq_", names_to = 'word',
               values_to = 'avg_freq') %>%
  pivot_wider(names_from = spam, values_from = avg_freq) %>%
  rename(not_spam = '0', spam = '1') %>%
  summarise(word, diff_avg_freq = spam - not_spam)

print(diff_avg_word_freq)
```

```
## # A tibble: 48 x 2
##    word     diff_avg_freq
##    <chr>            <dbl>
##  1 make            0.0765
##  2 address        -0.115
##  3 all             0.213
##  4 3d              0.157
##  5 our             0.310
##  6 over            0.129
##  7 remove          0.261
##  8 internet        0.177
##  9 order           0.123
## 10 mail            0.185
## # ... with 38 more rows
```

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

```
# plot histogram of word frequency
diff_avg_word_freq  %>%
  ggplot(aes(x = diff_avg_freq)) +
  geom_histogram() +
  labs(x = "Difference in Average Word Frequency",
```

```
       y = "Number of Words") +
  theme_bw()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

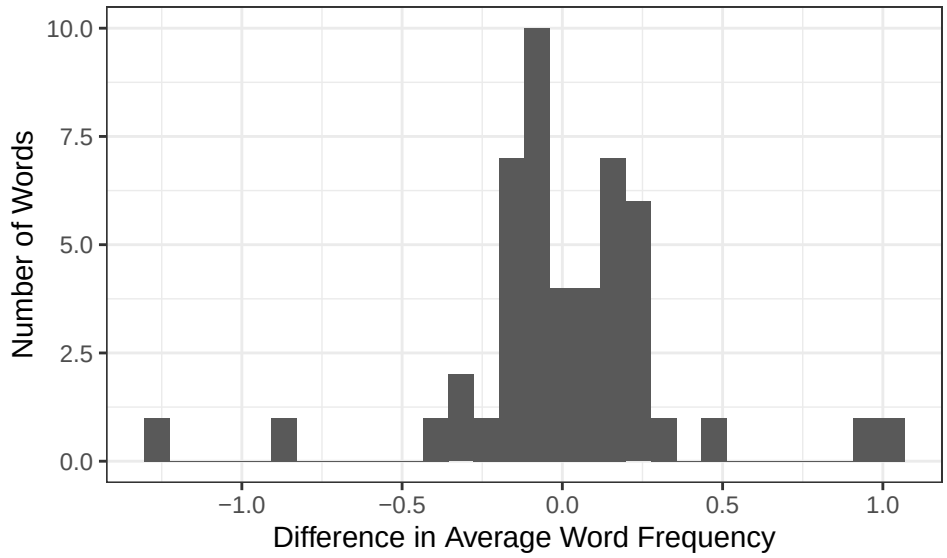


Figure 2: Distribution of word frequency differences between spam and non spam emails.

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

The top 3 underrepresented words in spam emails are george, hp, and hpl. These words make sense because they are more personalized/ descriptive words

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

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

2.1 Growing the default classification tree (8 points)

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

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

ii. (2 points) How many splits are there in this tree? How many terminal nodes does the tree have? **There are 7 splits and 8 terminal nodes in the tree**

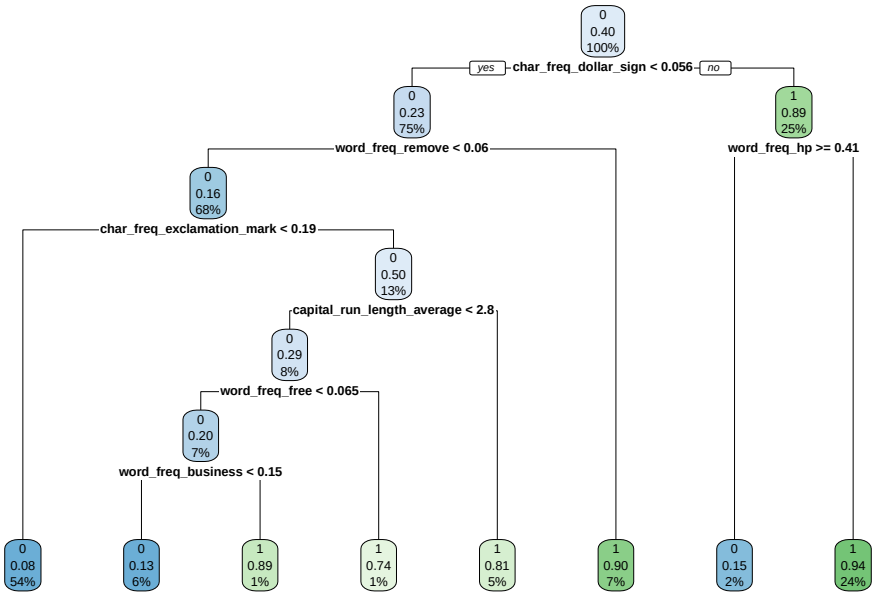


Figure 3: Classification tree for train data that predicts if an email is spam or not spam with splits based on the Gini index

iii. (5 points) What sequence of splits (specify the feature, the split point, and the direction) leads to the terminal node that has the largest fraction of spam observations? Does this sequence of splits make sense as flagging likely spam emails? What fraction of spam observations does this node have? What fraction of the training observations are in this node? **The highest fraction of spam observations occurs when the frequency of the dollar sign is greater than or equal to 0.058 and the the word frequency of hp is less than 0.41. This sequence of splits makes sense because emails with more dollar signs might indicate that someone is trying to attempt a phishing attack. Also, we know that the word hp is underrepresented in spam emails. 24% of the training observations are in this node**

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

Now let's find the optimal tree size.

2.2.1 Fitting a large tree T_0 (9 points)

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

ii. (2 points) First let us fit the deepest possible tree. In class we talked about the arguments `minsplit` and `minbucket` to `rpart.control`. What values of these parameters will lead to the deepest possible tree? There is also a third parameter `cp`. Read about this parameter by typing `?rpart.control`. What value of this parameter will lead to the deepest possible tree?

'minsplit' is the minimum number of observations that must exist in a node for a split to be attempted. Thus, when 'minsplit' is set to 1 (because there is no reason to split if there are zero observations), the deepest possible tree is created

'minbucket'is the minimum number of observations in any terminal node. Thus, when "minbucket" is set to 2 (does not make sense that you would split if there are not at least one observation in either side of the split), the deepest tree is created

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

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

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

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

```
##
## Classification tree:
## rpart(formula = spam ~ ., data = spam_train, method = "class",
##     parms = list(split = "gini"), control = rpart.control(minsplit = 1,
##         minbucket = 2, cp = 0))
##
## Variables actually used in tree construction:
##  [1] capital_run_length_average capital_run_length_longest
##  [3] capital_run_length_total   char_freq_dollar_sign
##  [5] char_freq_exclamation_mark char_freq_parenthesis
##  [7] word_freq_1999             word_freq_650
##  [9] word_freq_85               word_freq_addresses
## [11] word_freq_all              word_freq_business
## [13] word_freq_conference       word_freq_credit
## [15] word_freq_edu              word_freq_email
## [17] word_freq_free             word_freq_george
## [19] word_freq_hp               word_freq_hpl
## [21] word_freq_internet         word_freq_mail
## [23] word_freq_make             word_freq_meeting
## [25] word_freq_money            word_freq_order
## [27] word_freq_our              word_freq_over
## [29] word_freq_people           word_freq_pm
## [31] word_freq_re               word_freq_receive
## [33] word_freq_remove           word_freq_technology
## [35] word_freq_will             word_freq_you
## [37] word_freq_your
##
## Root node error: 1218/3065 = 0.4
##
## n= 3065
```

```
##
##       CP nsplit rel error xerror xstd
## 1  5e-01      0      1.00    1.0 0.02
## 2  1e-01      1      0.51    0.5 0.02
## 3  4e-02      2      0.36    0.4 0.02
## 4  3e-02      4      0.28    0.3 0.01
## 5  2e-02      5      0.25    0.3 0.01
## 6  1e-02      6      0.23    0.3 0.01
## 7  8e-03      7      0.22    0.2 0.01
## 8  6e-03      8      0.21    0.2 0.01
## 9  5e-03     10      0.20    0.2 0.01
## 10 4e-03     11      0.20    0.2 0.01
## 11 4e-03     12      0.19    0.2 0.01
## 12 3e-03     14      0.19    0.2 0.01
## 13 2e-03     18      0.17    0.2 0.01
## 14 2e-03     30      0.14    0.2 0.01
## 15 1e-03     45      0.12    0.2 0.01
## 16 1e-03     53      0.10    0.2 0.01
## 17 1e-03     61      0.09    0.2 0.01
## 18 1e-03     64      0.09    0.2 0.01
## 19 8e-04     68      0.08    0.2 0.01
## 20 5e-04     94      0.06    0.2 0.01
## 21 5e-04     97      0.06    0.2 0.01
## 22 4e-04    102      0.06    0.2 0.01
## 23 3e-04    115      0.05    0.2 0.01
## 24 3e-04    120      0.05    0.2 0.01
## 25 2e-04    123      0.05    0.2 0.01
## 26 0e+00    127      0.05    0.2 0.01
```

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

iv. (4 points) How many distinct trees are there in the sequence of trees produced in part iii? How many splits does the biggest tree have? How many average observations per terminal node does it have, and why is it not 1? **There are 26 distinct trees. The largest tree has 94 splits.NOT SURE NOT SURE**

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

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

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

Table 1: CP Table for Deepest possible 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	0.203	0.012
0.001	61	0.092	0.203	0.012
0.001	64	0.089	0.206	0.012
0.001	68	0.085	0.206	0.012
0.001	94	0.063	0.220	0.013
0.000	97	0.062	0.222	0.013
0.000	102	0.059	0.221	0.013
0.000	115	0.053	0.221	0.013
0.000	120	0.052	0.227	0.013
0.000	123	0.051	0.227	0.013
0.000	127	0.050	0.236	0.013

```
  theme_bw()
```

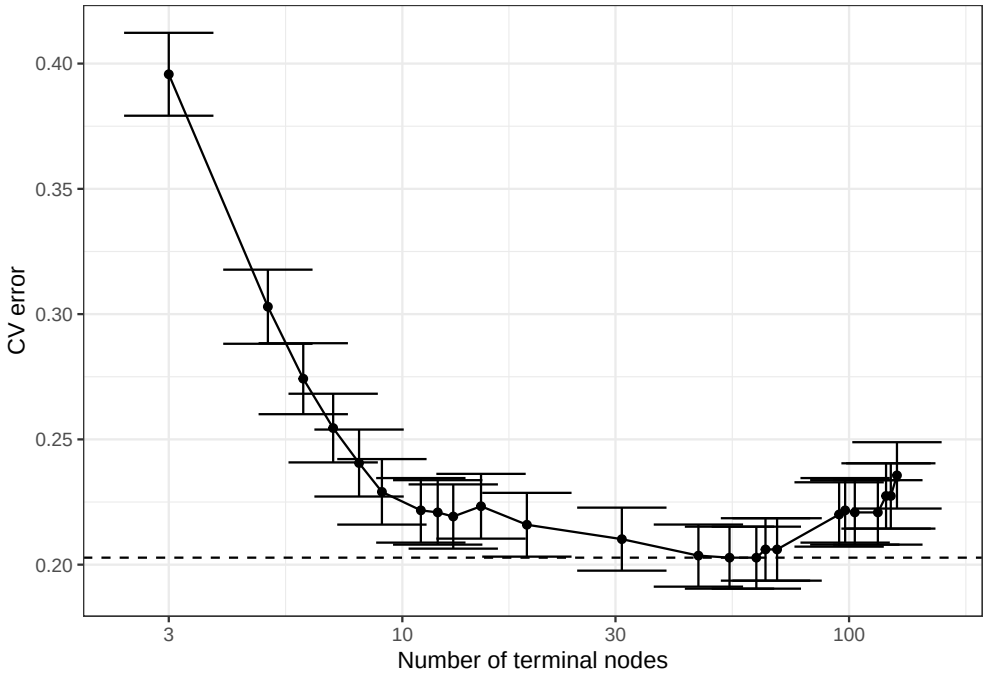


Figure 4: CV plot for deepest tree, where nsplit is grerater than 2

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

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

Using the one standard error rule, the tree has 31 terminal nodes, which is higher than than the number of terminal nodes in the default tree above

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

```
optimal_tree = prune(tree = tree_fit, 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(spam ~ ., data = spam_train)
```

The value of 'mtry' is set to the number of columns divided by 3 rounded down. Since there are 57 columns, not including the spam column, 'mtry' is equal to 19

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

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

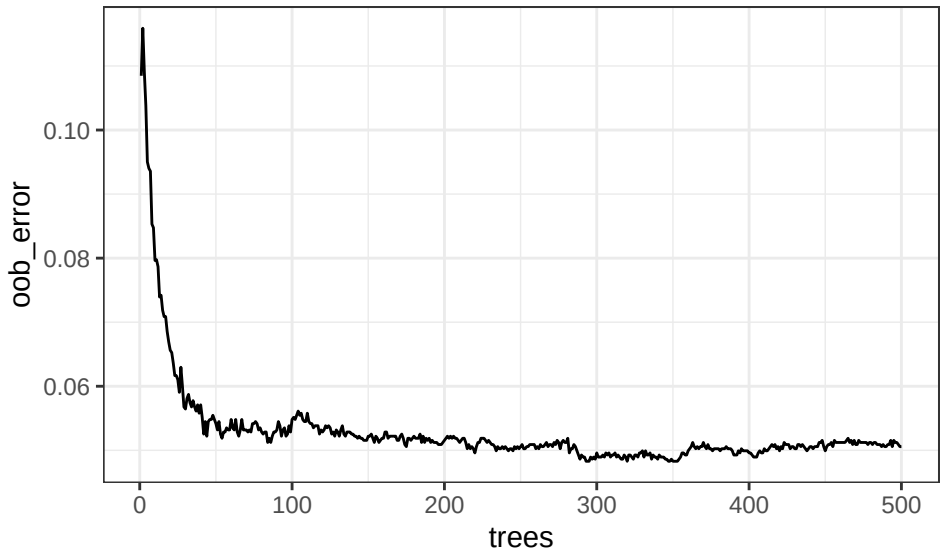


Figure 5: OBB error as a function of the number of trees

OOB error roughly stabilizes at 300 trees

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.

```
system.time(randomForest(spam ~ ., data = spam_train))
```

Table 2: Elapsed Run Time by Number of Trees Trained On

Number of Trees	Run Time
100	0.80
200	1.61
300	2.33
400	3.31
500	4.20

```
##    user  system elapsed
##    4.02    0.07    4.09
```

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

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

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

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

3.2.2 Dependence on the number of trees

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

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

```
num_tree = c(100, 200, 300, 400, 500)
times = c(0, 0, 0, 0, 0)
for(i in 1:5){
  times[i] = system.time(randomForest(spam ~ ., data = spam_train,
                                      ntree=num_tree[i]))["elapsed"]
}
tibble('Number of Trees' = num_tree, 'Run Time' = times) %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE,
        digits = 3,
        caption = "Elapsed Run Time by Number of Trees Trained On") %>%
 kable_styling(position = "center")
```

ii. (2 points) What relationship between runtime and number of trees do you observe? Does it make sense in the context of the training algorithm for random forests? **As the number of trees increases, the runtime increases. This makes sense because it is more computationally intensive to train on a greater number of trees**

3.3 Tuning the random forest (8 points)

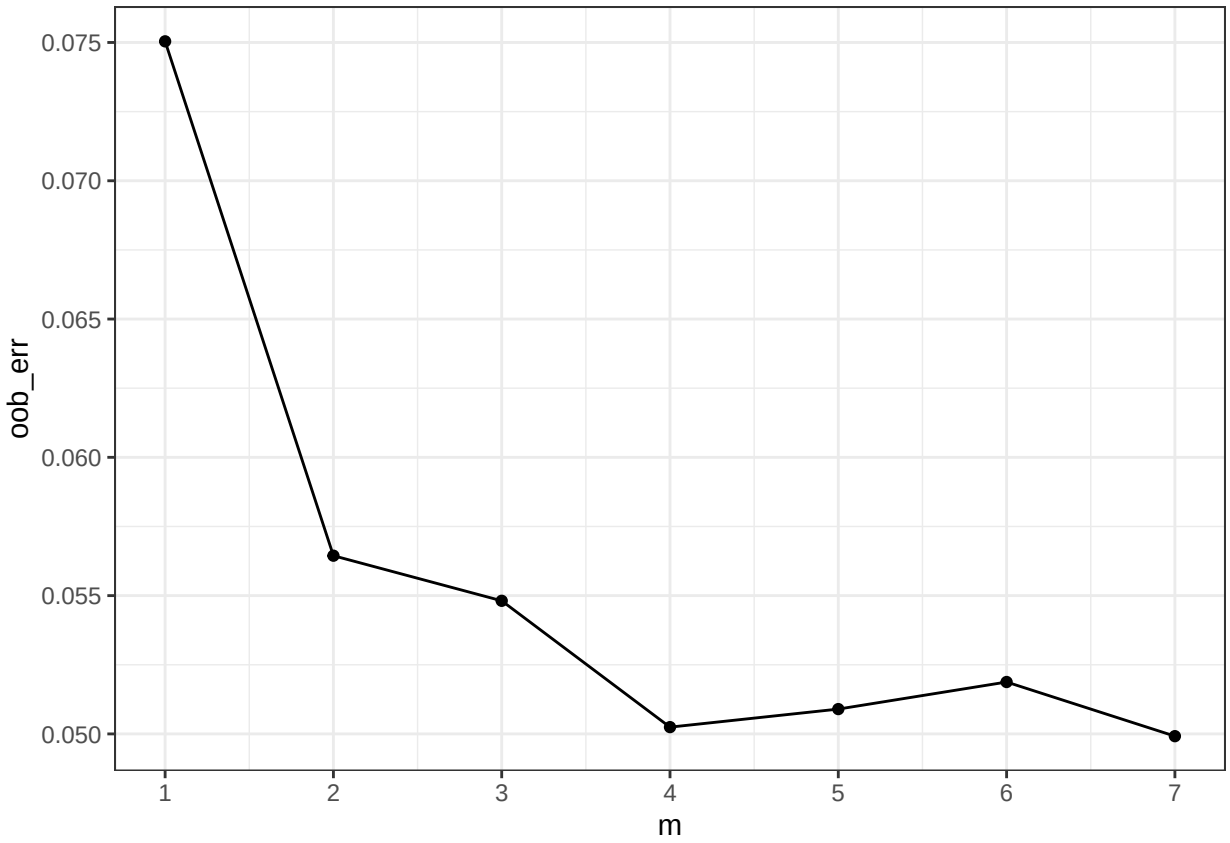
i. (2 points) Since tuning the random forest is somewhat time consuming, we want to be careful about tuning it smartly. To this end, does it make sense to tune the random forest with `importance = FALSE` or `importance = TRUE`? Based on OOB error plot from above, what would be a reasonable

number of trees to grow without significantly compromising prediction accuracy? **It makes sense to set `importance = FALSE` because doing so cuts the runtime in half. EXPLAIN WHY IT IS OKAY TO NOT HAVE THIS Based on the OOB error plot from above, it seeems reasonable to grow 300 trees because this is when OOB error starts to level off.**

ii. (2 points) About how many minutes would it take to train a random forest with 500 trees for every possible value of m? (For the purposes of this question, you may assume for the sake of simplicity that the choice of m does not impact the training time too much.) Suppose you only have enough patience to wait about 15 seconds to tune your random forest, and you use the reduced number of trees from part i. How many values of m can you afford to try? (The answer will vary based on your computer. Some students will find that there is time for only one or a few values; this is ok.) **Since there are 57 columns in the data (excluding spam), 'mtry' can range from 1 to 57. Thus, there are 57 possible values of m. To train on every value, it would take 57 x 4 seconds or 228 seconds, which is equal to around 4 minutes** **It takes around 2 seconds to train based on 500 trees. Therefore I can afford to try 7 values of m because 2*7 is equal to 14, which is under the 15 second limit.**

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

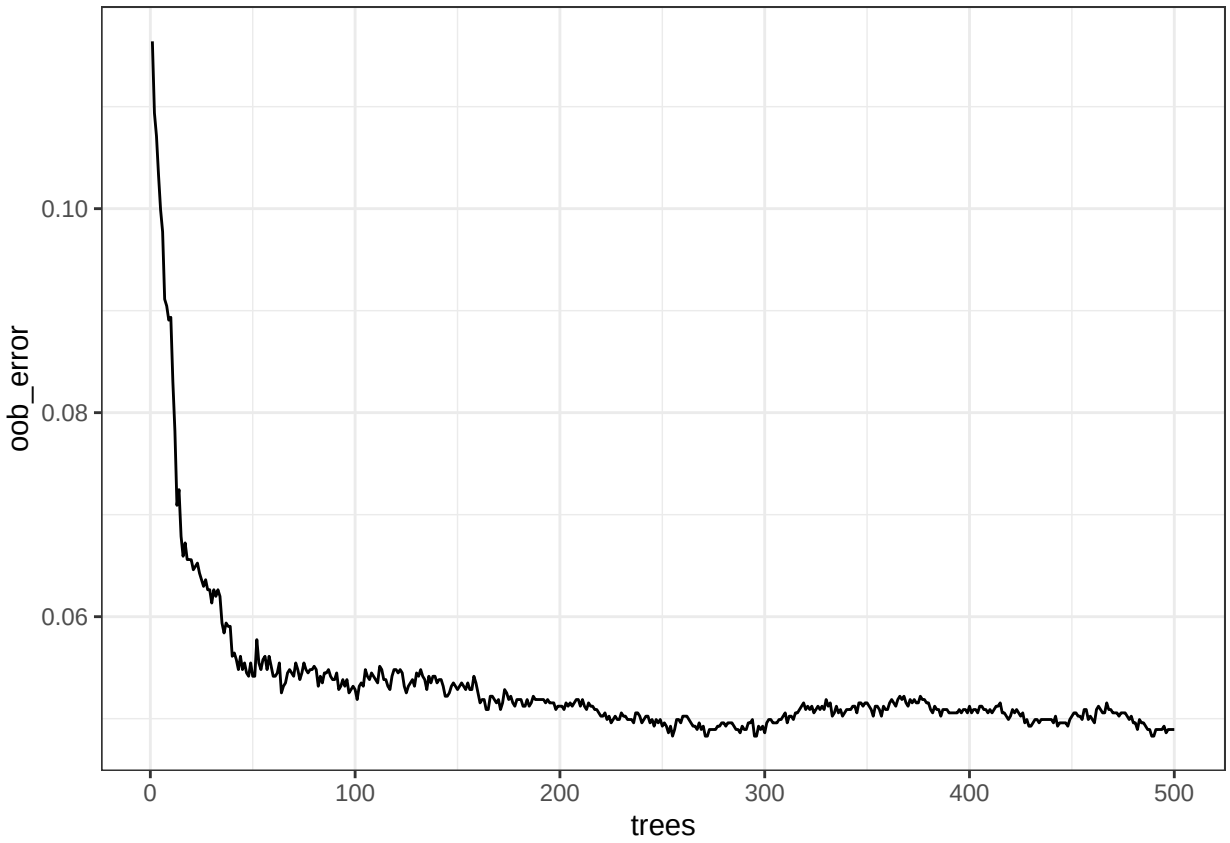
```
# might want to cache this chunk!
mvalues = seq(1,7, by = 1)
oob_errors = numeric(length(mvalues))
ntree = 300
for(idx in 1:length(mvalues)){
  m = mvalues[idx]
  rf_fit = randomForest(spam ~ ., mtry = m, data = spam_train)
  oob_errors[idx] = rf_fit$err.rate[ntree,"OOB"]
}
tibble(m = mvalues, oob_err = oob_errors) %>%
  ggplot(aes(x = m, y = oob_err)) +
  geom_line() + geom_point() +
  scale_x_continuous(breaks = mvalues) +
  theme_bw()
```




```
rf_fit_tuned  = randomForest(spam ~ ., ntree= 300, mtry = 3, data = spam_train)
```

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_500 = randomForest(spam ~ ., ntree= 500, mtry = 3, data = spam_train, importance = TRUE)
tibble(oob_error = rf_fit_500$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw()
```



Error appears to have flattened out at 300 because this is point when OOB error reaches its minimum

3.4 Variable importance (6 points)

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

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

```
varImpPlot(rf_fit_tuned)
```

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.

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)
# TODO: Fit random forest with interaction depth 1

set.seed(1) # for reproducibility (DO NOT CHANGE)
# TODO: Fit random forest with interaction depth 2
```

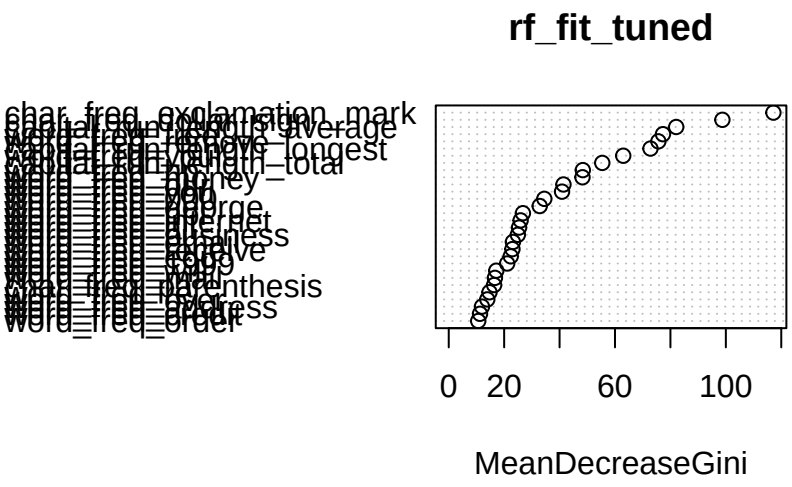


Figure 6: Variable importance plot for random forest model trained on the optimal value of m

```
set.seed(1) # for reproducibility (DO NOT CHANGE)
# TODO: Fit random forest with interaction depth 3
```

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?

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?

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.

`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?

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.

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.

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?