STAT 4710: Homework 4

Name

Due: November 14, 2023 at 9:00pm

Contents

Instructions

Materials and collaboration

The policy on allowed materials and collaboration is as stated on the Syllabus:

> "Students are permitted to work together on homework assignments, but must write up and submit solutions individually. In particular, students may not copy each others' solutions. Students may consult all course materials, textbooks, the internet, or AI tools (e.g. ChatGPT or GitHub Copilot) to complete their homework. Students may not use solutions to problems that may be available online and/or from past iterations of the course. For each homework, students must disclose all classmates with whom they collaborated, which AI tools they used, and how they used them. Failure to do so will result in a 5-point penalty."

In accordance with this policy,

Please disclose all classmates with whom you collaborated:

Please disclose which AI tools you used, and how you used them:

Failure to answer the above questions will result in a 5-point penalty.

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. In particular, if the instructions ask you to "print a table", you should use `kable`. If the instructions ask you to "print a tibble", you should not use `kable` and instead print the tibble directly.

Programming

The `tidyverse` paradigm for data visualization, manipulation, and wrangling is required. No points will be awarded for code written in base R.

We'll need to use the following `R` packages:

```
library(rpart)          # to train decision trees
library(rpart.plot)     # to plot decision trees
library(randomForest)   # random forests
library(gbm)            # boosting
library(tidyverse)      # tidyverse
library(stat471)        # for cv_tree()
library(kableExtra)     # for printing tables
library(cowplot)        # for side by side plots
```

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 4 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.

Let's load the data.

```
spam_data <- read_tsv("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)
```

We will be using `spam_train` in the first four sections of this homework, and `spam_test` in the last section.

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.

1. (1 points) What fraction of the training observations are spam?
2. (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?

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.

1. (3 points) Produce a tibble called `avg_word_freq` containing the average frequencies of each word by calling `summarize` on `spam_train`. Print this tibble (no need to use kable.
2. (3 points) Create a tibble called `avg_word_freq_long` by pivoting `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 `names_prefix = "word_freq_"` within your pivoting function to remove this prefix.]

3. (2 points) Produce a histogram or bar plot 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? [Note: You can produce either a histogram of the word frequencies (where the height of the bars is the number of words in that range of word frequencies) or a bar plot of the word frequencies (where there is one bar per word, whose height is that word's frequency). If you do the former, you will need to separately figure out the top three most frequent words. If you do the latter, you can read this information off directly from the plot. However, for the latter strategy, please format the plot in a way such that the word labels are easy to read.]

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.

4. (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. See if you can accomplish this using one continuous sequence of pipes.]

5. (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?

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

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

2.1 Growing the default classification tree (8 points)

1. (1 point) Fit a classification tree with splits based on the Gini index, with default `control` parameters. Plot this tree.
2. (2 points) How many splits are there in this tree? How many terminal nodes does the tree have?
3. (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 the observations in this node are spam? What fraction 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)

1. (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?
2. (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?

3. (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)
```

4. (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?

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

1. (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()`.
2. (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?
3. (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).

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)

1. (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)
```

2. (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?

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.

Hint: This question asks you to run code and report how long it took. The runtime of code varies from run to run, and setting a seed in this case will not help. In particular, the answers will be different when you run the code chunks in R Markdown and when you compile to PDF. For consistent answers in your PDF, you can save any relevant variables in the R code chunk and then reference them in your answer using inline code.

1. (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(...))["elapsed"]`; see `?system.time` for more details.

2. (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`?

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`.

1. (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`.

2. (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?

3.3 Tuning the random forest (8 points)

1. (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? [Note: You can answer this question "by eye"; there is no need to be too precise here. The OOB curve is somewhat noisy, so there is not an exact right answer here.]

2. (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 1. How many values of m can you afford to try?

3. (2 points) Tune the random forest based on the choices in parts 1 and 2, choosing the values of m to be roughly equally spaced between 1 and p (and including these two boundary values). 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? [Hint: You can generate roughly equally spaced integer values using the `seq.int()` function.]

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

4. (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)
```

3.4 Variable importance (6 points)

1. (2 points) Produce the variable importance plot for the random forest trained on the optimal value of m. Please plot the variable importances only for the top ten features, using the argument `n.var = 10`.

2. (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)

1. (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 `n.cores = 1`.

```
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

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

2. (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)

1. (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?

2. (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.

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

1. (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?

2. (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. Set `eval = TRUE` in the chunk header.

```
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 logarithmic scale for clearer visualization.

3. (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?