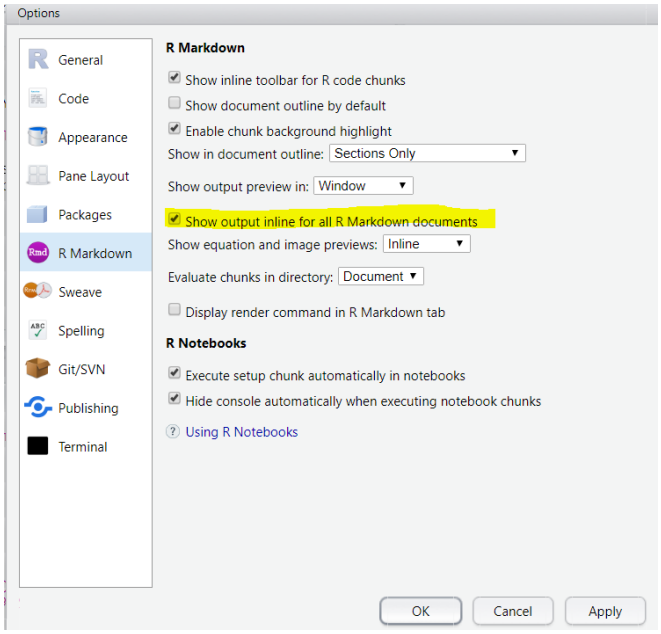
Observations from Apple health data using Gradient Boosting Machines and other handy tools

If you own an iPhone or an Apple Watch, your movements are being tracked and recorded in the Health app. The intent behind this statement is not to increase feelings of paranoia around modern-day technology, but instead to express that this is a dataset that you can access easily from within your device. The Health app creates a handful of summarizing plots to help users to analyze their data. These are huge datasets with many possibilities, so it's a great option to download and begin playing with on your own. If you've been wearing an Apple Watch, you likely have quite a few health metrics recorded in detail that you could choose to analyze. If, like me, you have just the iPhone, you should at least have enough data on your walking and running movements to try a few machine learning techniques out. This tutorial will take you through step by step as you access your health data, and then use R to load, preprocess, and apply a few tree-based models to these data.

The tree-based models that will be used in this analysis include decision trees, random forest algorithm and a gradient boosting machine (GBM). All three of these methods are based off of the most simple, decision tree and they can be used either for classification or regression types of tasks. The questions asked of this dataset are purely classification questions, because the datset doesn't have that many numeric variables to work with. Decision trees split the data into two groups at each node based on the values of the predicting variables. For example, for a tree predicting rain vs snow, there would probably be a node that split at 32 degrees F that predicted snow from freezing temperatures and rain from non-freezing temperatures. With decision trees, it is easy to look at the output of the model and figure out exactly how the results were achieved. Random forests and GBM models both have an element of black box-iness to them and thus are not as interpretable. They will be covered in more detail in theire respective sections.

I usually work in RStudio markdown documents because this is how I first learned to code in R. For anyone who also prefers these tools, I'm going to draw attention to a particular issue that I faced and which I expect most laptops to experience at some point while running this code. As I was working through this project, I found a few points where my computer memory had trouble keeping up with all the output that I was printing out in-line. If this becomes a problem for your computer, at any point, just go to the Global Options menu, found under the Tools tab, and deselect the option to show output inline. I prefer to see some of the output printed inline, so I found myself toggling this option on and off frequently based on which functions I was running.



Accessing your Apple health data

Regardless of whether your data is being recorded by your phone or a watch, both are connected by your Apple ID, so you should look in the Health app on your iPhone to access your data. Choose the "Browse" button on the bottom panel and then select your apple id icon at the top right of the screen. Scroll to the end of this page and choose the button that reads "Export All Health Data." Confirm this action by choosing "Export" again. This will compress your health data into a file named "export.zip" that you can now email to yourself or upload to google drive. Make sure that this file is saved and unzipped to a folder on your computer. Inside the apple_health_export folder, you will find two .xml documents. We will be working with the file named "export.xml".

Install required packages

The following packages were used to perform this analysis. If you do not already have these packages installed on your device, you can install them by uncommenting and running the second line of code. Next you must load these packages, by running line 3.

```
1.  packages = c("tidyverse", "XML", "lubridate", "rpart", "rpart.plot", "randomForest", "gbm",
    "caret")
2.  #sapply(packages, install.packages)
3.  sapply(packages, library, character.only = T)
```

Load in the data

These data are formatted as XML files, so we will use the XML package to load the data into R. There are several different files provided with the download from the health app, but we are only going to work with the file named "export.xml". Read in the file using the xmlParse function, then convert the data

object to a data frame for ease of use. You will need to change the expression inside of the xmlParse function to reflect the correct file path on your computer.

```
1.  xml = xmlParse(str_c(base_dir, "export_2/apple_health_export/export.xml"))
2.  dist_data = XML:::xmlAttrsToDataFrame(xml["//Record"])
```

Preprocess the data

These data come with an observation at each time movement is detected. For instance, if I left the house at 10 am, walked to the park for 15 minutes, sat down on a bench for 5 minutes, walked 15 minutes back home, and then sat down on the couch for the remaining 25 minutes of the hour, my phone would log two observations during that hour- one for the first 15-minute walk and one for the second 15-minute walk. Nothing would be logged during the 5 minutes sitting on the park bench or the 25 minutes on the couch. All distances are recorded in miles. If you wear an Apple watch, while the same principles still apply, you might notice more sensitivity in your data.

There are several “types” of observations that the Health app records. Again, the Apple Watch will have even more types to choose from in these analyses, but we will only work with the walk/run data. This next code block will filter for the walk/run data and also use the lubridate package to extract some additional variables from those already provided. As each observation occurs over an interval in time, just because the interval started during the 7 o’clock hour does not mean the interval will also finish within that hour. For this reason, I have set the following code block to create detailed start and end times for each observation. I prefer to save the result to a new variable to preserve the original data in my workspace and prevent the need to reload it as I work with it. You may not need to make any edits to the code in these preprocessing steps unless you are interested in variables that I have not included in my analysis.

```
1.   dist_data1 = dist_data %>%
2.    filter(type == "HKQuantityTypeIdentifierDistanceWalkingRunning") %>%
3.    mutate(value = as.numeric(value)) %>%
4.    select(-type, -sourceName, -unit, - device, -sourceVersion) %>%
5.    mutate(start = ymd_hms(startDate),
6.           end = ymd_hms(endDate),
7.           startYear = year(start),
8.           startMonth = month(start),
9.           startDay = day(start),
10.          startHour = hour(start),
11.          startMin = minute(start),
12.          interval = interval(start, end),
13.          dT = time_length(interval, "hour"),
14.          dT = if_else(dT == 0, .00001, dT),
15.          endYear = year(end),
16.          endMonth = month(end),
17.          endDay = day(end),
18.          endHour = hour(end),
19.          endMin = minute(end))
```

The second set of manipulations draws from outside data sources. For example, my data goes from 2017 until 2021. I created a new column that indicates that 2020 and 2021 are pandemic years and that 2017-2019 were not pandemic years. You could add any additional columns that may be of interest to you.

For example, if you just got a new puppy and have since spent time walking it, perhaps “puppy” versus “no puppy” is predictive of your time spent walking and running.

```
1.   dist_data1 = dist_data1 %>%
2.    mutate(pandemic = as.factor(if_else(startYear == 2021 | (startYear == 2020 & startMonth >=
    3), T, F)),
3.           season = if_else(startMonth %in% c(6,7,8), "summer", "NA"),
4.           season = if_else(startMonth %in% c(12,1,2), "winter", season),
5.           season = if_else(startMonth %in% c(3,4,5), "spring", season),
6.           season = if_else(startMonth %in% c(9,10,11), "fall", season),
7.           school_month = if_else(startMonth %in% c(1,2,3,4,5,9,10,11), T, F),
8.           weekend = if_else(weekdays(start) %in% c("Saturday", "Sunday"), "weekend",
    "weekday"),
9.           day_of_week = weekdays(start),
10.          daytime = if_else(startHour %in% c(21,22,23, 24, 1,2,3,4,5,6), "nighttime",
    "daytime"),
11.          mph = value/dT,
12.          startYear = as.factor(startYear),
13.          startMonth = as.factor(startMonth),
14.          startDay = as.factor(startDay),
15.          startHour = as.factor(startHour))
```

One way to work with these data is to take it in hourly chunks. Select potentially relevant variables and summarize the data such that it contains only one observation per hour. In some cases where no movement was logged during a particular hour, this will mean there is no observation for that hour. Weighted means were used over a regular mean to prevent a short increase or decrease in speed during the interval from having a disproportionate effect on the overall mean.

```
1.  dist_data2 = dist_data1 %>%
2.    group_by(startHour, startDay, startMonth, startYear) %>%
3.    mutate(hourly_value = sum(value)) %>%
4.    mutate(mph = weighted.mean(mph, dT/60, na.rm = T),
5.           dT_hour = sum(dT, na.rm = T)) %>%
6.    select(hourly_value, daytime, season, weekend, startMonth, startHour, startDay, startYear,
    school_month, day_of_week, pandemic, mph, dT_hour) %>%
7.    distinct()
```

We are going to create a few different versions of this data now. The following version (3) includes both the hourly value and mph calculated above, but also a monthly value so that the data can be used for analyses on a monthly scale.

```
1.  dist_data3 = dist_data2 %>%
2.    group_by(startMonth, startYear) %>%
3.    mutate(monthly_value = sum(hourly_value))
```

Version 4 does not include hourly or monthly summaries, but instead looks at the data with a daily granularity.

```
1.  dist_data4 = dist_data2 %>%
2.    group_by(startMonth, startYear, startDay) %>%
3.    mutate(daily_value = sum(hourly_value))%>%
4.    mutate(mpd = weighted.mean(mph,dT_hour/24, na.rm = T)) %>%
5.    select(-dT_hour, -mph, - startHour, - hourly_value) %>%
6.    distinct()
```

Explore the data

For the majority of this project, I focused on answering questions related to the predictability of weekends and global pandemics from my walking and running data. So ahead of presenting those methods and results, first, let's take a look at the data.

I primarily worked with versions 3 and 4 of the data and even processed these versions of the data further in later steps to prepare them for the models that I was interested in applying. Version 3, introduced below, includes both monthly and hourly dimensions on the data.

hourly_value <dbl>	daytime <chr>	season <chr>	weekend <chr>	startMonth <fctr>	startHour <fctr>	startDay <fctr>	startYear <fctr>	school_month <lgl>	day_of_week <chr>	pandemic <fctr>	mph <dbl>	dT_hour <dbl>	monthly_value <dbl>
0.012427400	nighttime	fall	weekday	11	23	25	2016	TRUE	Friday	FALSE	1.355716364	0.009166667	14.55188
0.183043430	daytime	fall	weekend	11	0	26	2016	TRUE	Saturday	FALSE	0.236354501	0.774444444	14.55188
0.013744700	nighttime	fall	weekend	11	1	26	2016	TRUE	Saturday	FALSE	0.124323920	0.110555556	14.55188
0.068680200	nighttime	fall	weekend	11	2	26	2016	TRUE	Saturday	FALSE	0.180210437	0.381111111	14.55188
0.065778412	nighttime	fall	weekend	11	4	26	2016	TRUE	Saturday	FALSE	0.361530203	0.181944444	14.55188
0.041538720	nighttime	fall	weekend	11	5	26	2016	TRUE	Saturday	FALSE	0.610364865	0.068055556	14.55188
0.325169810	daytime	fall	weekend	11	14	26	2016	TRUE	Saturday	FALSE	0.371740653	0.874722222	14.55188
0.656099900	daytime	fall	weekend	11	15	26	2016	TRUE	Saturday	FALSE	1.010247921	0.649444444	14.55188
1.414545000	daytime	fall	weekend	11	16	26	2016	TRUE	Saturday	FALSE	1.286274817	1.099722222	14.55188
2.494712000	daytime	fall	weekend	11	17	26	2016	TRUE	Saturday	FALSE	2.506548479	0.995277778	14.55188
0.690363000	daytime	fall	weekend	11	18	26	2016	TRUE	Saturday	FALSE	1.140572189	0.605277778	14.55188
0.164265610	daytime	fall	weekend	11	19	26	2016	TRUE	Saturday	FALSE	0.237206657	0.692500000	14.55188
0.353789610	daytime	fall	weekend	11	20	26	2016	TRUE	Saturday	FALSE	0.831359397	0.425555556	14.55188
0.121478100	nighttime	fall	weekend	11	21	26	2016	TRUE	Saturday	FALSE	0.459370966	0.264444444	14.55188
0.004038912	nighttime	fall	weekend	11	22	26	2016	TRUE	Saturday	FALSE	0.037474441	0.107777778	14.55188
0.026010590	daytime	fall	weekend	11	0	27	2016	TRUE	Sunday	FALSE	0.459010412	0.056666667	14.55188

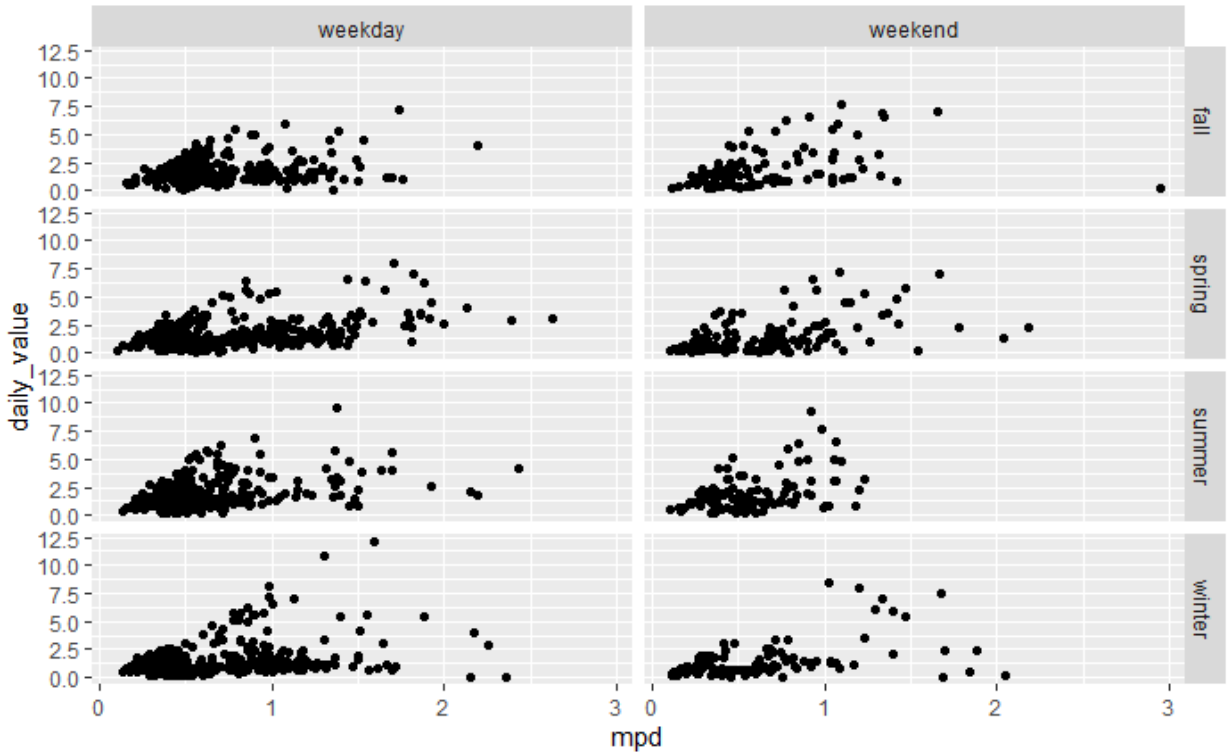
1-16 of 14,120 rows

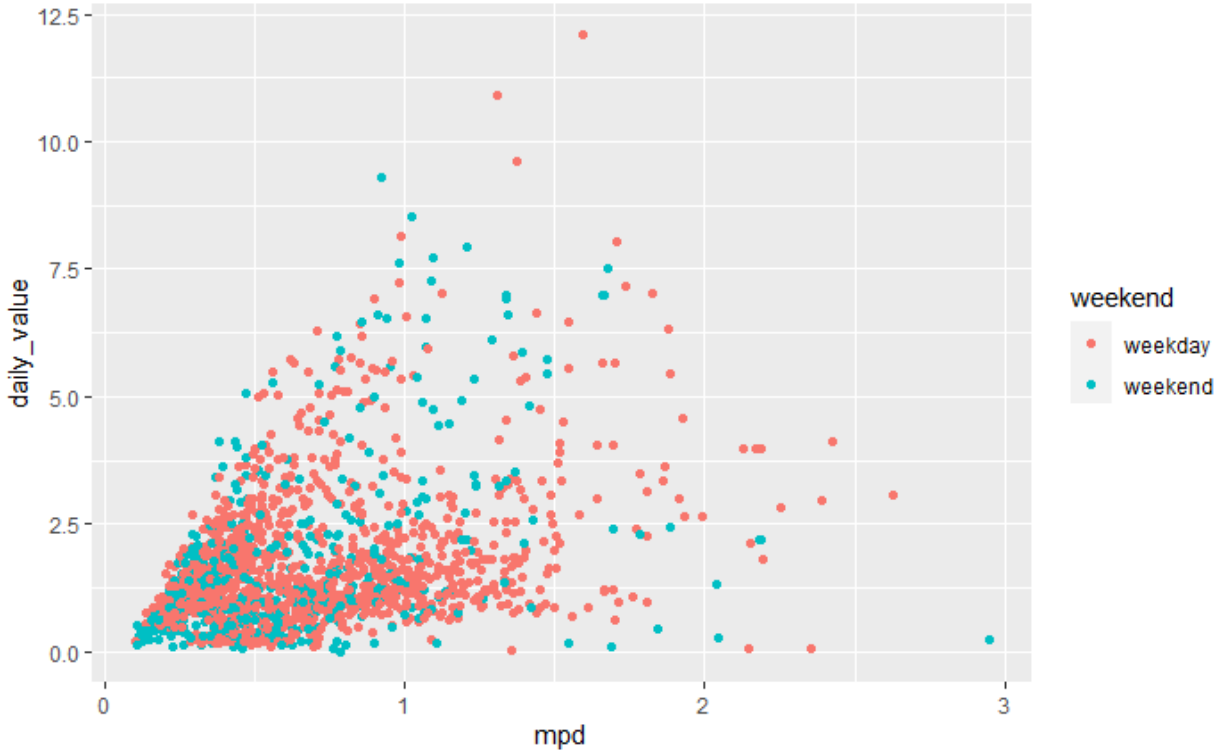
Previous 1 2 3 4 5 6 ... 63 Next

Version 4 of the data includes many of the same variables, but instead approaches the data with a daily perspective, for example, it includes a miles per day column rather than a mile per hour column.

To prepare to ask questions about weekend predictability, below are a couple scatter plots illustrating the shapes of the weekend data. By eyeball, I'm not seeing huge differences in daily movement based on these variables.

```
1.   dist_data4 %>%
2.    ggplot(aes(x = mpd, y = daily_value))+
3.    geom_point()+
4.    facet_grid(season~weekend)
5.
6.  dist_data4 %>%
7.    ggplot(aes(x = mpd, y = daily_value, color = weekend))+
8.    geom_point()
```





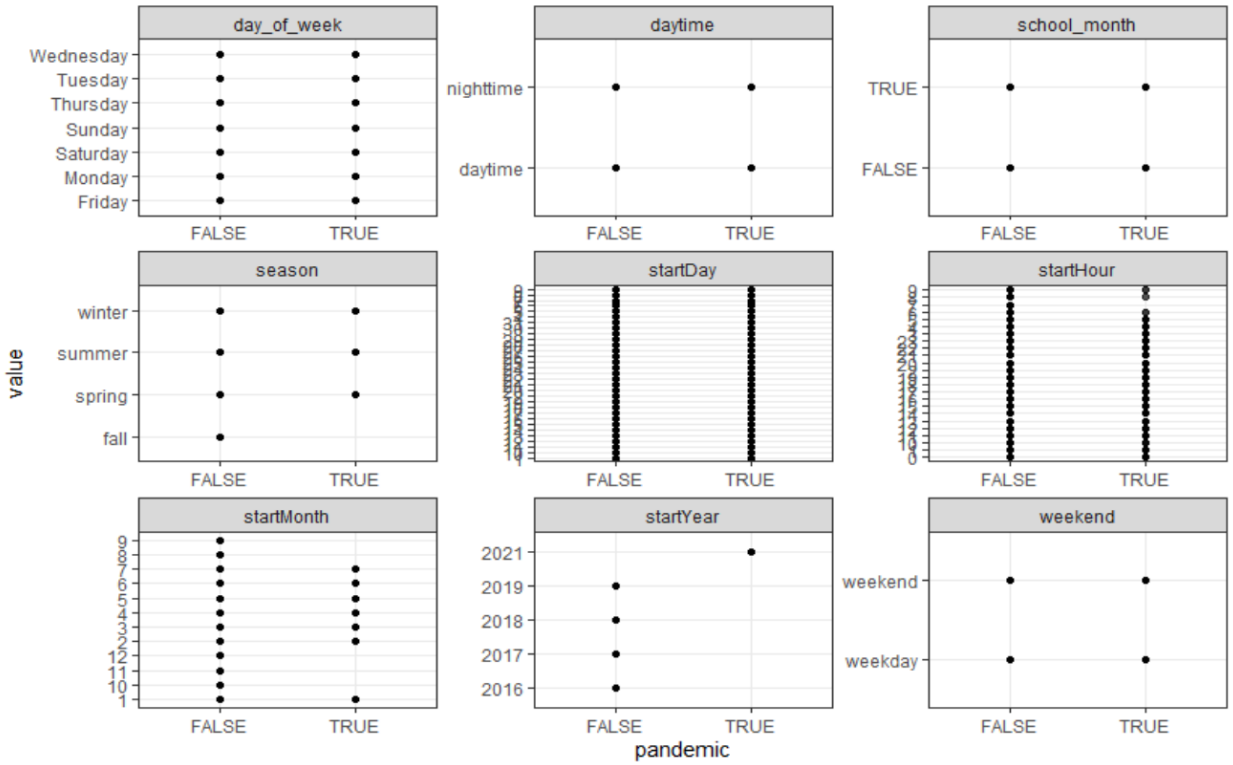
The following two plots show an overview of categorical and numeric variables as they relate to the logical variable, pandemic. There are clear differences between the pandemic and pre-pandemic distributions for several variables.

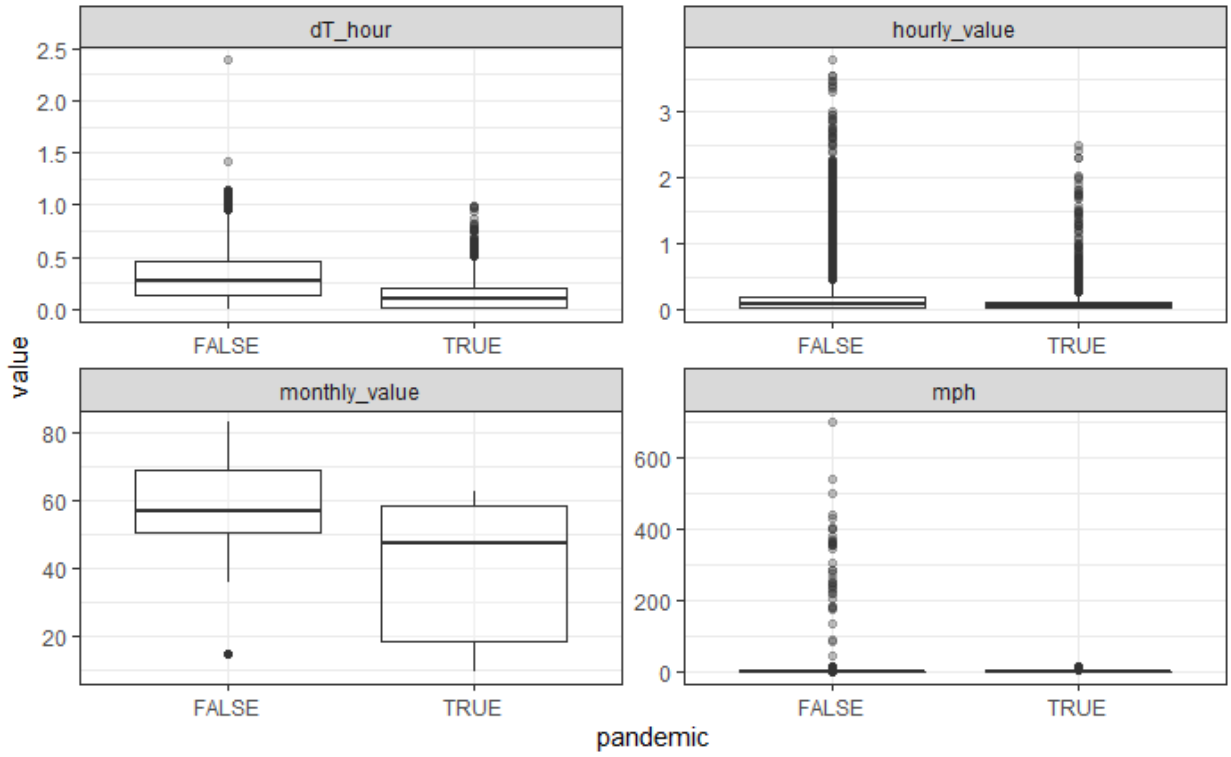
```
df = dist_data3 %>%
  select(-hourly_value, - mph, -dT_hour, -monthly_value) %>%
    gather(-pandemic, key="var", value = "value")

ggplot(df, aes(x = pandemic, y = value)) +
    geom_point() +
    facet_wrap(~var, scales = "free") +
    theme_bw()

df = dist_data3 %>%
  ungroup() %>%
  select(hourly_value,  mph, dT_hour, monthly_value, pandemic) %>%
    gather(-pandemic, key="var", value = "value") %>%
  mutate(value = as.numeric(value))

ggplot(df, aes(x = pandemic, y = value)) +
    geom_boxplot() +
    facet_wrap(~var, scales = "free") +
    theme_bw()
```





Apply a classification tree

At this point, the data has been processed and explored so we can start applying ML models. Quick-R has a good lesson on classification trees which I borrowed for this step of the project. It can be accessed at the following link: https://www.statmethods.net/advstats/cart.html. This method for classification trees uses the rpart and rpart.plot packages.

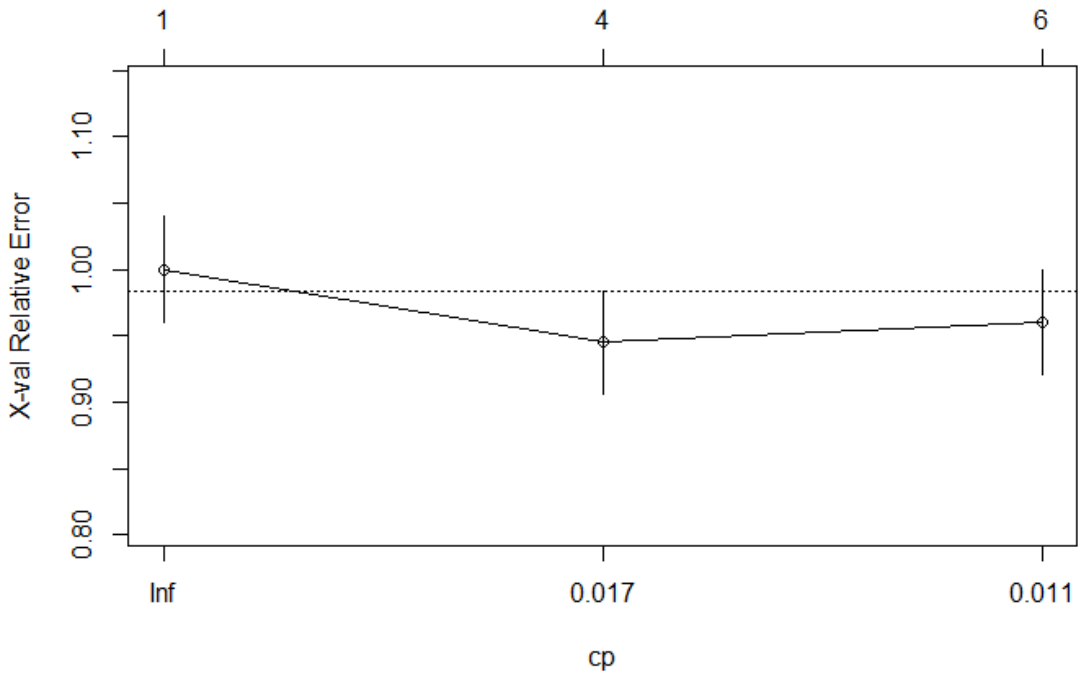
The data first needs to be split into training and test datasets. The following code block splits version 4 of the dataset randomly into 50% training and 50% validation. Usually, a 20/80 split is recommended when training an ML model for a purpose, but given that my only goal is to look at the performance of the models, I chose to give it a large percentage of observations to use in training. I've set a random seed so that the results should be reproducible for anyone following along with my dataset.

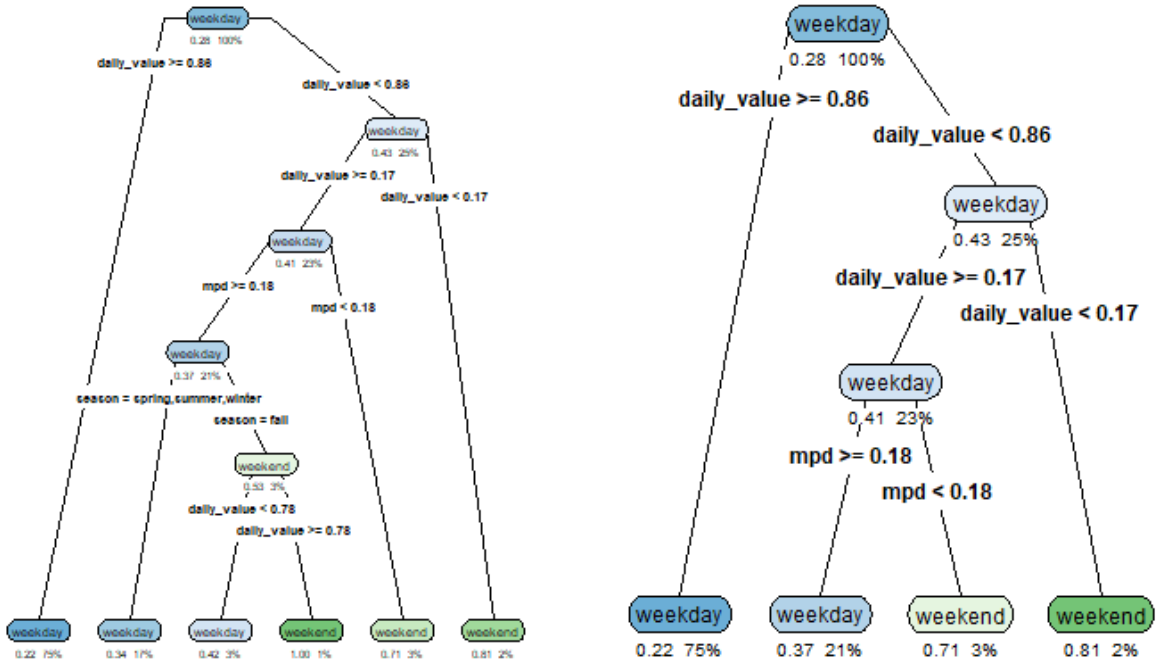
```
1.  set.seed(10)
2.  sample = sample(nrow(dist_data4), nrow(dist_data4)*.5)
3.  train = dist_data4[sample,]
4.  test = dist_data4[-sample,]
```

Next, we can apply a classification tree. Notice the formula given in line 1. Weekend is the response variable, while mpd, daily_value, and season are the predicting variables. These can be adjusted to whatever question you would like to ask. I am asking if my walking and running movements are predictive of whether it is a weekend or not. These variables might be predictive of a day's weekend status because it would make sense if I walked a very regular small amount on weekdays to get to classes, on weekends I might expect to walk more or less depending on my weekend plans. And all of this might change depending on the season because I might stay in more during the winter or change my patterns during the summer when school isn't in session.

This will produce a handful of plots, including a relative error plot and two trees. The best model is going to have the lowest relative error with the lowest complexity (cp). In this case, the model is best when the complexity parameter is 0.01103. This number was calculated by the expression fit$cptable[which.min(fit$cptable[,"xerror"]),"CP"] in line 7. Of the two trees, the first and more complicated tree was the end result of the classification model. However, the end result was not the best result in this case due to overfitting, so the prune function cuts the tree back to the cp of 0.01103 and the pruned result is the second tree produced by this code block. The two numbers given under each node on the tree are the predicted probability that an observation is going to be a weekend and the percent of the total observations that ended up in that node. The table produced by line 10, shows that this model isn't performing very well. It is predicting 1136 weekdays and 35 weekends correctly. It is wrongly predicting 18 weekdays as weekends and 449 weekends as a weekday. This means that it is predicting weekdays very well, but predicting weekends very poorly. The caret package's confusionMatrix function gives balanced accuracy as 0.528, which is calculated by averaging sensitivity (the true positive rate) with the specificity (the true negative rate).

```
1.  fit = rpart(weekend~mpd+daily_value+season,
2.     method="class", data=train)
3.  printcp(fit)
4.  plotcp(fit)
5.  summary(fit)
6.  rpart.plot(fit, type = 4, clip.right.labs = FALSE, branch = .3, under = TRUE)
7.  pfit= prune(fit, cp=fit$cptable[which.min(fit$cptable[,"xerror"]),"CP"])
8.  rpart.plot(pfit, type = 4, clip.right.labs = FALSE, branch = .3, under = TRUE)
9.  pred = predict(pfit, test, type="class")
10. confusionMatrix(table(pred, test$weekend))
```



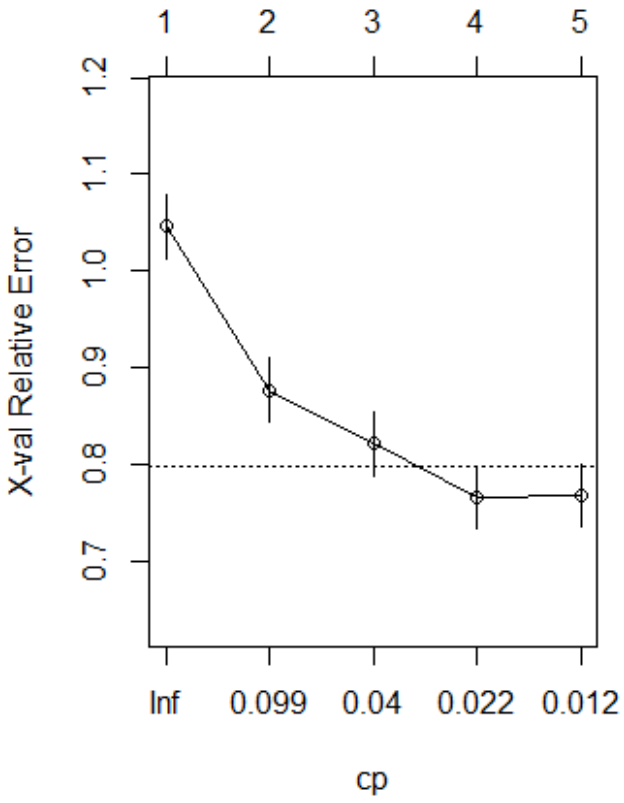


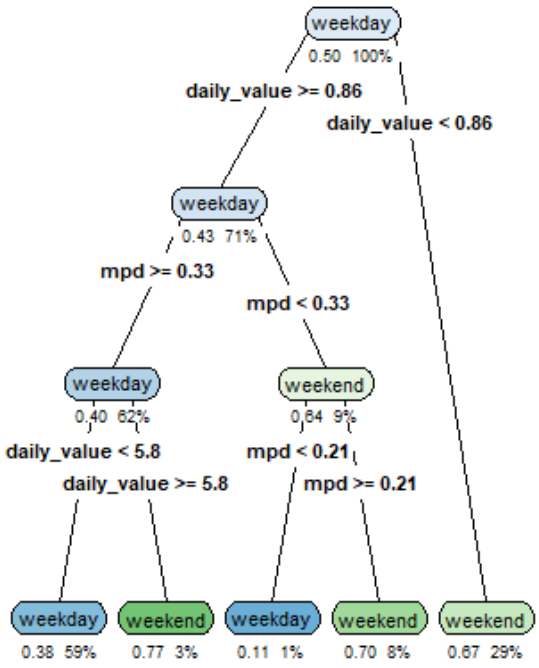
The next thing to do to try to improve this model is to equalize the number of weekends and weekdays present in the data so that weekdays aren't overrepresented. The next code block takes random weekdays until the number of weekdays is equal to the number of weekend days. The best cp to use for the pruned tree was calculated at 0.01545. This time the confusion matrix showed that the model correctly predicted weekdays 794 times, correctly predicted weekends 245 times, incorrectly predicted weekdays as weekends 360 times, and incorrectly predicted weekends as weekdays 239 times. This is a

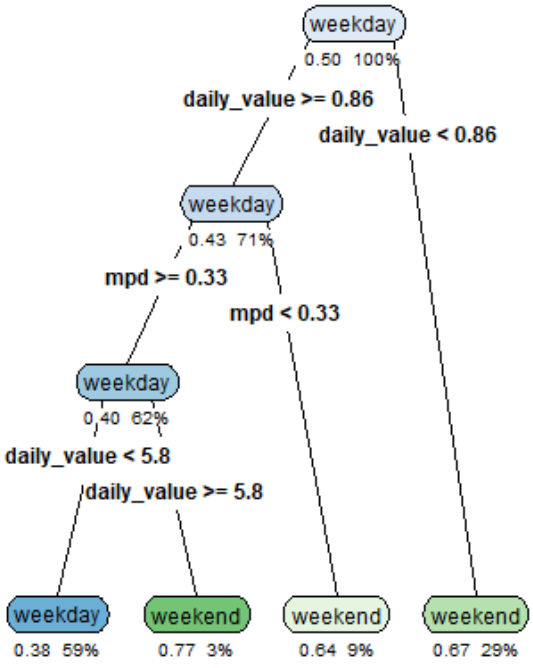
clear improvement in weekend predictions, but with diminished performance predicting weekdays. This time caret calculated a balanced accuracy of 0.597, a small improvement on the accuracy of the previous model.

```
dist_data4_weekend = train %>%
  filter(weekend == 'weekend')
dist_data4_weekday = train %>%
  filter(weekend == 'weekday')
rows = sample(nrow(dist_data4_weekday), nrow(dist_data4_weekend))
dist_data4_weekday = dist_data4_weekday[rows,]
dist_data4_both = bind_rows(dist_data4_weekday,dist_data4_weekend)

fit = rpart(weekend~mpd+daily_value+season,
   method="class", data= dist_data4_both)
printcp(fit)
plotcp(fit)
summary(fit)
rpart.plot(fit, type = 4, clip.right.labs = FALSE, branch = .3, under = TRUE)
pfit= prune(fit, cp=fit$cptable[which.min(fit$cptable[,"xerror"]),"CP"])
rpart.plot(pfit, type = 4, clip.right.labs = FALSE, branch = .3, under = TRUE)
pred = predict(fit, test, type="class")
confusionMatrix(table(pred, test$weekend))
```







Based on the results of these classification models I'm going to go out on a limb and say that my movements are not terribly predictive of whether it is a weekend or not.

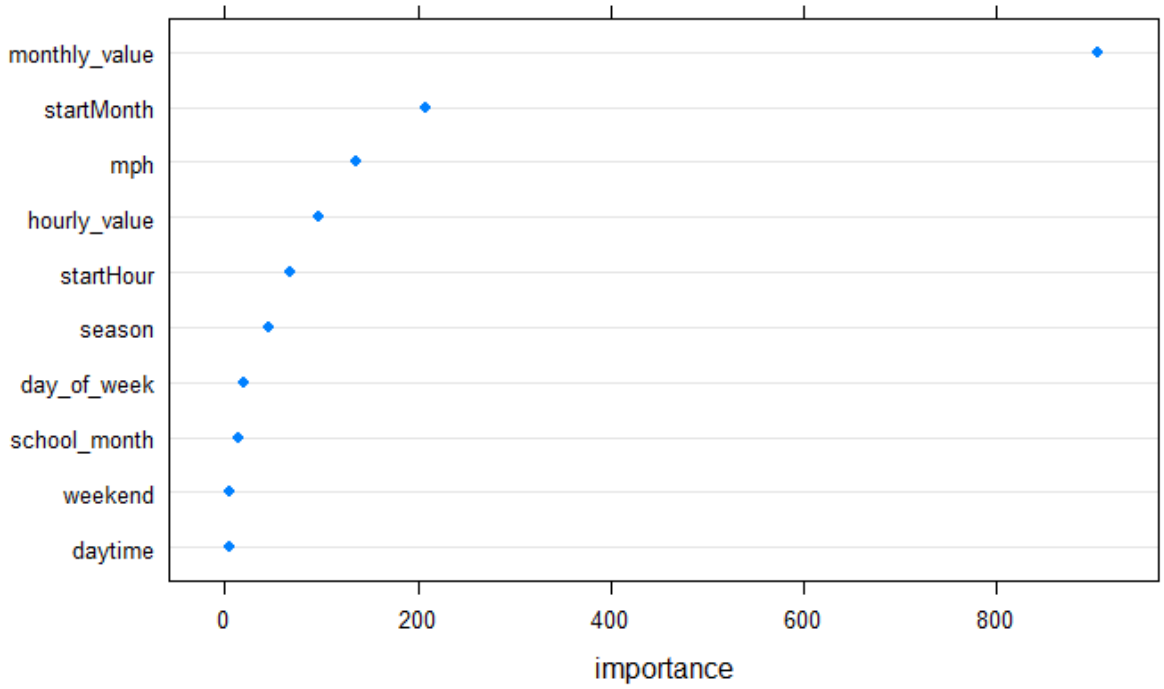
Apply a random forest algorithm

Decision trees like those performed in the previous section tend to have difficulty when applied to noisy data because the solution is reached with the creation of just one tree. A random forest algorithm takes the results of many trees, each produced from different subsets of the data. Each of those individual decision trees is going to come up wit its own result independent of each other. The only cooperation between these trees happens at the end when they compare their results and compromise or vote on a final result. The final result of the random forest is going to be the average result of all the trees, or in the case of classification trees, instead of the average it would take the most popular result.

The following code block will use version 3 of the Apple health data that we pre-processed earlier to predict whether an observation occurred during the COVID-19 pandemic from all available predicting variables except for those that would be a dead giveaway, for example, the year. First, I ensured that there was no missing data for the response variable, to prevent errors in the randomForest function. Then I divided the data into training and validation sets and ran the model.

The variables that had the largest effect on the results were monthly value and start month, which seemed reasonable to me given that the pandemic is best measured by the dimensions of months or even years. Daily and hourly data are probably noisier than it is worth. The results of this model looked very promising, even suspiciously promising, with an out-of-bag error rate of 0.23. When the model is set to predict the testing data, it performed at a .99% balanced accuracy. The model correctly predicted non-pandemic observations 100% of the time, and only incorrectly predicted pandemic observations 2.2% of the time, despite the number of pandemic and non-pandemic observations not being balanced in the training dataset.

```
1.  dist_data3 = dist_data3 %>%
2.    filter(!is.na(pandemic))
3.
4.  set.seed(10)
5.  sample = sample(nrow(dist_data3), nrow(dist_data3)*.5)
6.  train = dist_data3[sample,]
7.  test = dist_data3[-sample,]
8.
9.  fit = randomForest(pandemic~ hourly_value+daytime+ season+
    weekend+startHour+school_month+day_of_week+mph+startMonth+monthly_value,   data= train)
10. print(fit)
11. imp = importance(fit)
12. imp
13. imp_asc = imp[order(imp[,1]),]
14. dotplot(imp_asc)
15.
16. pred = predict(fit, test, type = "class")
17. confusionMatrix(table(pred, test$pandemic))
```



Out of curiosity, I also chose to apply this method to the question of weekend predictability. The response variable needed to be converted to a factor for this one to work. Also, the test confusion matrix required rearranging so that it would match the output of the training confusion matrix. Because the classification tree in the previous section responded best to balancing the training data, I repeated those methods for this model. As expected, daily value and miles per day were the most important variables in this calculation. Despite the more sophisticated method, balanced accuracy remained low, this time at 0.557.

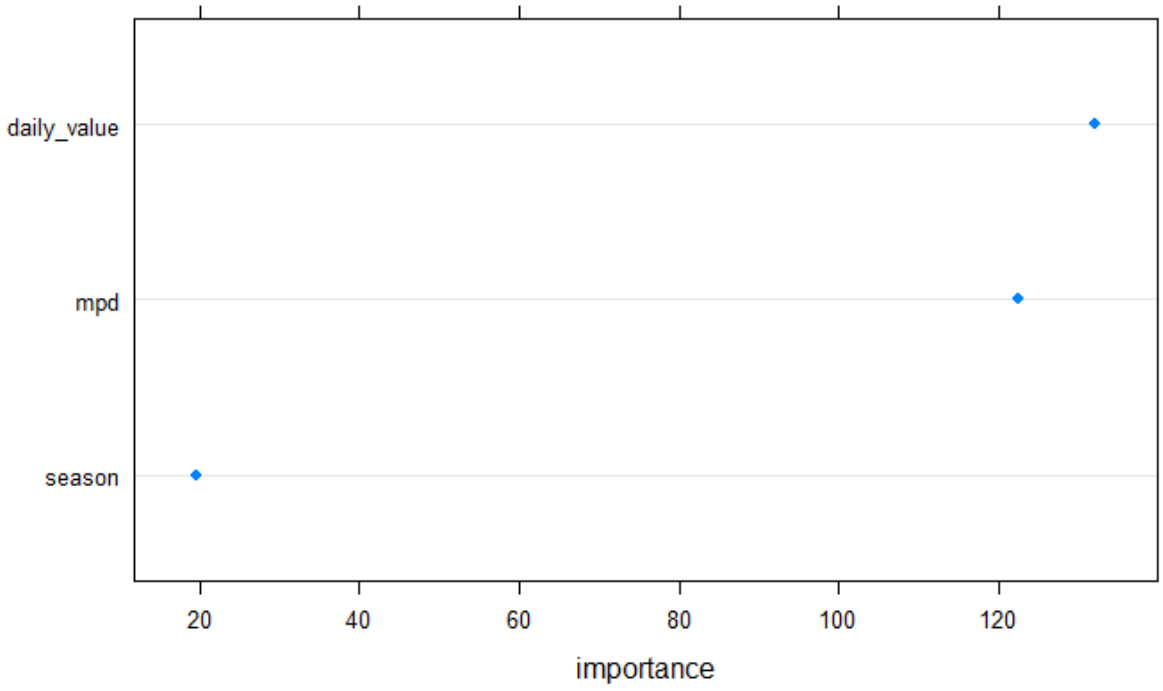
```
dist_data4 = dist_data4 %>%
  mutate(weekend = ifelse(weekend == "weekend", 1, 0))
set.seed(10)
sample = sample(nrow(dist_data4), nrow(dist_data4)*.5)
train = dist_data4[sample,]
test = dist_data4[-sample,]
train = train %>%
  mutate(weekend = as.factor(weekend))
dist_data4_weekend = train %>%
  filter(weekend == 1)
dist_data4_weekday = train %>%
  filter(weekend == 0)
rows = sample(nrow(dist_data4_weekday), nrow(dist_data4_weekend))
dist_data4_weekday = dist_data4_weekday[rows,]
dist_data4_both = bind_rows(dist_data4_weekday,dist_data4_weekend )
test = test %>%
  mutate(weekend = as.factor(weekend))

fit <- randomForest(weekend~mpd+daily_value+season,   data= train)
print(fit)
imp = importance(fit)
```

```
imp
importance <- imp[order(imp[,1]),]
dotplot(importance)


pred = predict(fit, test, type = "class")

confusionMatrix(table( pred, test$weekend)[1:2, 2:1])
```



Apply a Gradient Boosting Machine

The next model that we will work with is a GBM, or gradient boosting machine. This analysis requires that variables be coded as numeric vectors. The following code block creates a version 5 data frame that is derived from version 3, except all the variables are now coded as numerics.

```
  dist_data5 = dist_data3 %>%
   mutate(daytime = if_else(daytime == "daytime", 1, 0)) %>%
   mutate(weekend = if_else(weekend == "weekend", 1, 0)) %>%
   mutate(day_of_week = if_else(day_of_week == "Monday", "1", day_of_week),
          day_of_week = if_else(day_of_week == "Tuesday", "2", day_of_week),
          day_of_week = if_else(day_of_week == "Wednesday", "3", day_of_week),
          day_of_week = if_else(day_of_week == "Thursday", "4", day_of_week),
          day_of_week = if_else(day_of_week == "Friday", "5", day_of_week),
          day_of_week = if_else(day_of_week == "Saturday", "6", day_of_week),
          day_of_week = if_else(day_of_week == "Sunday", "7", day_of_week)) %>%
  mutate(season = if_else(season == "fall", "4", season),
          season = if_else(season == "winter", "1", season),
          season = if_else(season == "spring", "2", season),
          season = if_else(season == "summer", "3", season)) %>%
```

```
15.   mutate(season = as.numeric(season),
16.          day_of_week = as.numeric(day_of_week)) %>%
17.   mutate(school_month = as.numeric(school_month),
18.          pandemic = as.numeric(pandemic),
19.          startMonth = as.numeric(startMonth),
20.          startHour = as.numeric(startHour),
21.          startDay = as.numeric(startDay),
22.          startYear = as.numeric(startYear))
```

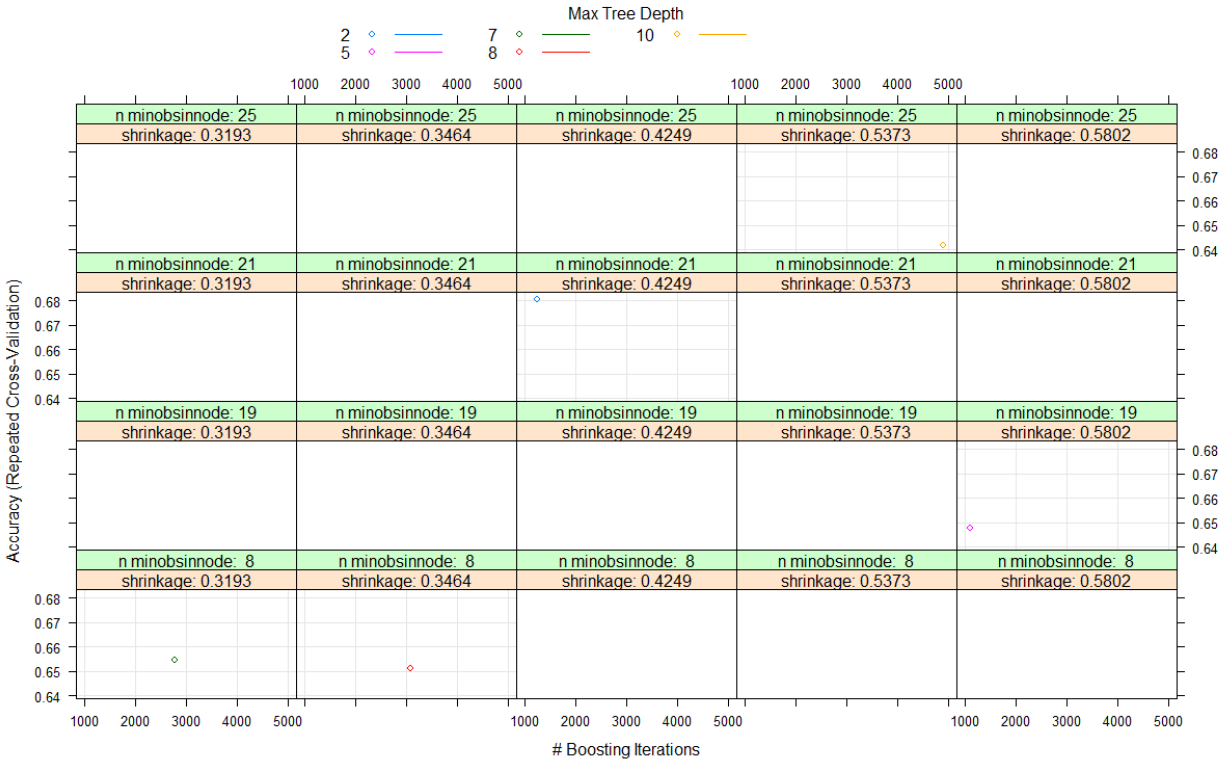
The GBM that I will be running will be using the AdaBoost distribution. This model works by starting with all training data observations equally weighted. Then a classification tree is built off of that version of the data. As the model learns, it adjusts the weights on the observations so that difficult classifications have higher weight and easy classifications have lower weight. New classification trees are produced off of each new set of weights and the new trees are added to the previous ones, in theory creating better models with each iteration. This type of model is more advanced than the rondom forest algorithm because each iterative tree has the ability to learn from the errors of the previous decision trees in addition to the training data.

We will continue to work with the question of weekend prediction. At this point, I would not expect any model to predict this variable well, but with the knowledge of the performance of the previous models, at least I can set reasonable expectations for the GBM results. We will begin with a random grid search to help determine which parameters to use. This step is important because overfitting is common with this type of model. This is one of those functions that you'll want to set it to run and then go take your dog for a walk because it won't run quickly.

```
1.  sample = sample(nrow(dist_data5), nrow(dist_data5)*.5)
2.  training = dist_data5[sample,]
3.  test = dist_data5[-sample,]
4.
5.  fitControl = trainControl(method = "repeatedcv",
6.  number = 3,
7.  repeats = 5,
8.  search = "random")
9.
10. GBM_grid_random = train(weekend~ hourly_value+daytime+ season+
    pandemic+startHour+school_month+mph+startMonth+monthly_value,
11. data = training,
12. method = "gbm",
13. distribution= "adaboost",
14. trControl = fitControl,
15. verbose = TRUE,
16. tuneLength = 5)
17.
18. plot(GBM_grid_random)
```

The output of this is a list of parameters and accuracy values. Plotting this list helps to interpret it. The highest accuracy came from using an interaction.depth = 2, n.minobsinnode = 21, shrinkage = 0.4249, and n.trees = 1257. The interaction depth is a measure of how many of the variables can depend on each other for the purposes of the model. Minimum observations in node prevents further branching at a minimum value. Shrinkage is another term for step size. This shrinkage value is pretty large, but this is also the parameter that is going to have the greatest impact on computational time. If you are trying to run a GBM and it is taking too long, raising the shrinkage will help, perhaps at the cost of accuracy. Additionally, if the shrinkage is low, then a larger number of trees will be required to reach peak

accuracy. Because the recommended shrinkage is high in this example, a relatively low number of trees may be suitable.



The next code block is going to run the GBM using the parameters recommended above. The results will output into a “large gbm” object.

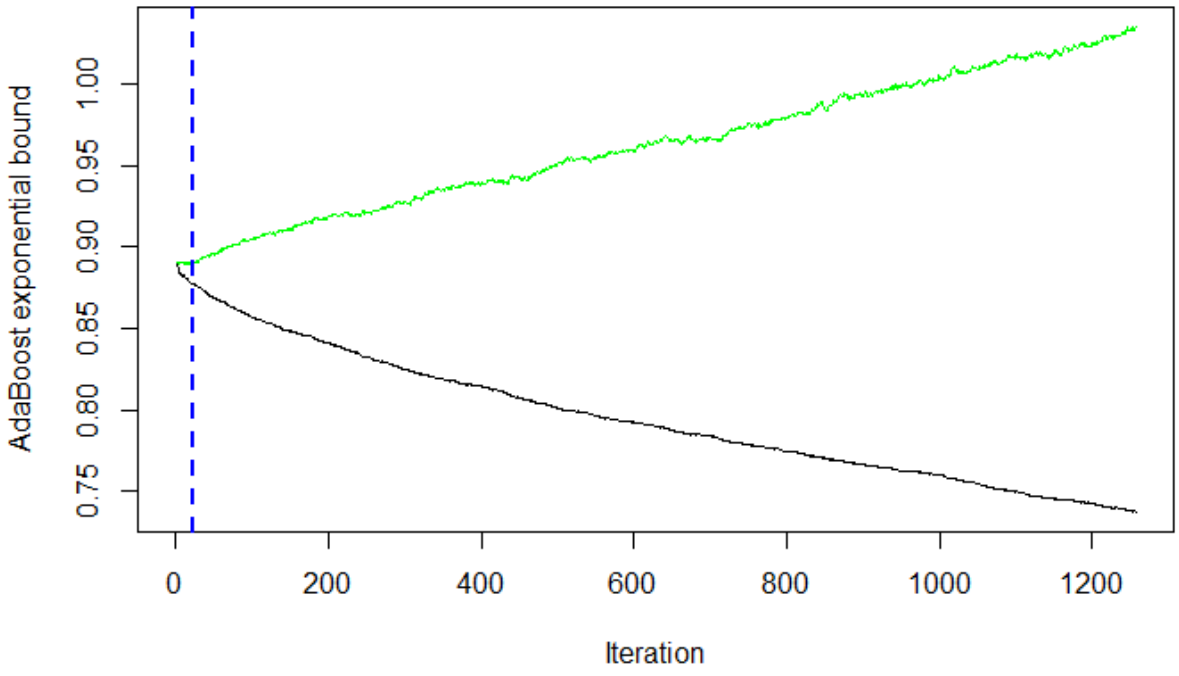
```
1.  gbm = gbm::gbm(formula=weekend~ hourly_value+daytime+ season+
    pandemic+startHour+school_month+mph+startMonth+mph,
2.             distribution = "adaboost",
3.             data= training,
4.             interaction.depth = 2,
5.             n.trees = 1257,
6.             shrinkage = .4249,
7.             verbose = T,
8.             cv.folds = 5,
9.             n.minobsinnode = 21)
```

Then, I run the following lines of code to view several different summaries of these results. Hourly value had the highest relative influence value, followed up by miles per hour. The remaining variables had lower influences on the result.

```
1.   for (i in 1:length(gbm$var.names)){
2.    plt = plot(gbm, i.var = i)
3.    print(plt)
4.  }
5.
6.  summary(gbm)
7.
8.  gbm.perf(gbm, method = "cv")
```

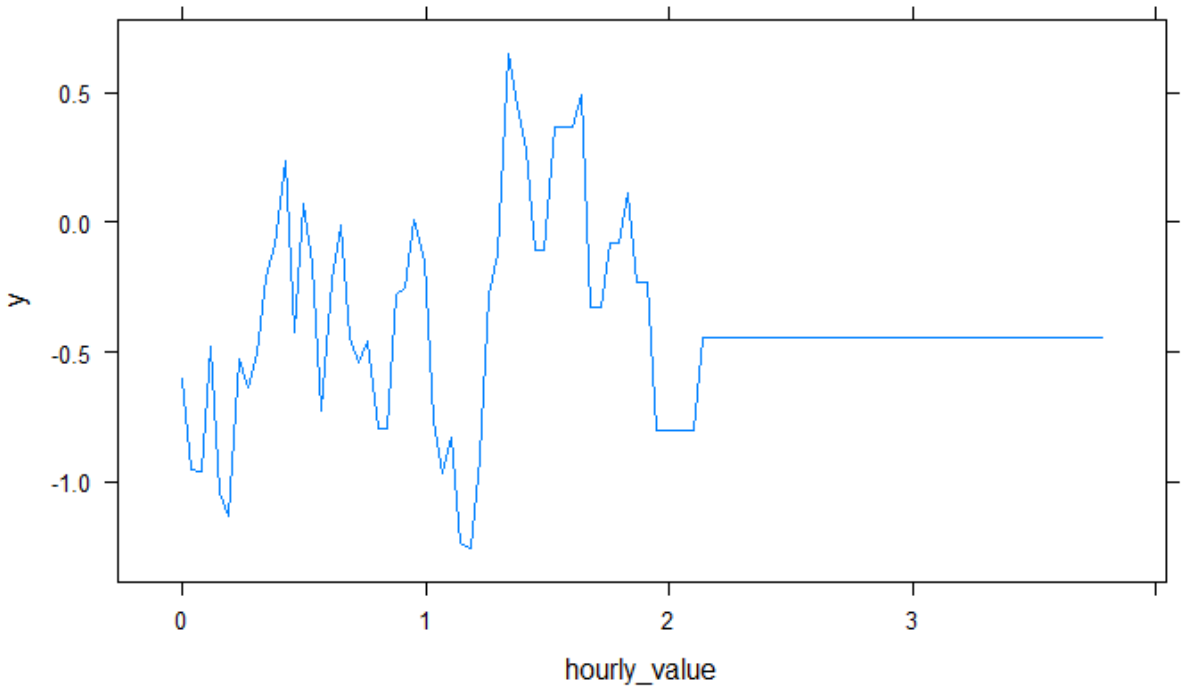
	var	rel.inf
mph	mph	38.457982
hourly_value	hourly_value	38.080252
startHour	startHour	10.316470
startMonth	startMonth	7.138328
season	season	2.437673
pandemic	pandemic	1.321849
daytime	daytime	1.145651
school_month	school_month	1.101795

The below plot shows the model's performance. The gbm function further split the training data into testing and training. The black line is showing the error on the training data while the green line shows the error on the test data. Ideally, these lines would look more similar to each other. When the green line stretches upwards, as it is doing in this example, that indicates overfitting. The dotted blue line is the number of iterations where the test data error is minimized. In this case, the number of iterations to minimize the error in the validation set was 22, meaning that the GBM improved itself for 22 iterations then proceeded to use the remaining iterations to overfit. A potential way to improve the overfitting is to increase the number of cv.folds. Currently the number of folds is set to 5, which means that the data has been divided into five groups and the model is run independently on each of those groups. The benefit of increasing the number of folds is that the model will not have access to the whole dataset at one time and therefore the GBM won't be able to memorize the entire dataset.



A set of partial dependency plots will also print from this code block. The y axis is a measure of how likely the variable is to predict the response variable. A negative value on the y axis would indicate that

the variable is highly unlikely to predict the response variable, while a positive value indicates that the variable is highly likely to predict the response variable. A flat line at zero indicates that the variable does not affect the model. Because the y value is changing a lot with changes to the predictor variable, hourly_value, this means that the model is very sensitive to this variable. Any variables that do not have any positive signal in these plots should probably be cut from the model because they are directly inhibiting the correct classification of the response variable and are not contributing anything useful. If I was to follow this advice, I would be removing all variables except hourly_value from my next version of this model.



Instead, I'm going to predict off of this very poor model. The model predicted 2392 weekdays correctly, and 1181 weekends correctly. It incorrectly predicted 2766 weekdays as weekends and 721 weekends as weekdays. Interestingly, this model did a better job at predicting weekends than the previous models, a result that I would speculate could be attributed to how this type of model puts more weight on difficult observations. Unfortunately, it had to sacrifice accuracy in predicting weekdays to achieve this result. The balanced accuracy off of this confusion matrix was 0.5423, which is about as poor as both previous attempts to answer this question.

```
1.  preds = format(round(predict(gbm, test, na.action = na.pass)))
2.
3.  preds = as.factor(preds)
4.  ref = as.factor(test$weekend)
5.
6.  levels(preds) = list ("0"  = "-1", "1" = " 0")
7.
8.  confusionMatrix(table(preds, ref))
```

Conclusion

These models are handy because tree-based models can be used on either categorical or numeric data types to answer classification or regression types of questions. These models can also handle data that isn't normally distributed. These models can still pick up on patterns in the data even if those patterns are non-linear. Though weekends were not particularly predictable using my data, I hope that this tutorial has provided enough background to begin working with your own Apple health data.

I have uploaded a version of my script as well as the data used to generate these results on my GitHub and I invite anyone to access these resources at https://github.com/ammquets/Apple_health_trees.

Citations

Anoushka Patel. (2018, August 21). *Advantages of Tree-Based Modeling*. https://www.summitllc.us/blog/advantages-of-tree-based-modeling

Bhalla, D. (n.d.). GBM (Boosted Models) Tuning Parameters. *ListenData*. Retrieved January 17, 2022, from https://www.listendata.com/2015/07/gbm-boosted-models-tuning-parameters.html

Brownlee, J. (2016, April 24). Boosting and AdaBoost for Machine Learning. *Machine Learning Mastery*. https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/

Dangeti, P. (2017). *Statistics for Machine Learning*. Packt Publishing Ltd.

GBM Tutorial. (2016). https://allstate-university-hackathons.github.io/PredictionChallenge2016/GBM

kjytay. (2020, January 23). What is balanced accuracy? *Statistical Odds & Ends*. https://statisticaloddsandends.wordpress.com/2020/01/23/what-is-balanced-accuracy/

Kuhn, M. (2019). *The caret Package*. https://topepo.github.io/caret/

Milborrow, S. (2021). *Plotting rpart trees with the rpart.plot package*. 36.

Rendyk. (2021, April 15). Tree-Based Machine Learning Algorithms | Compare and Contrast. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/04/distinguish-between-tree-based-machine-learning-algorithms/

Robert I. Kabacoff. (2017). *Quick-R: Tree-Based Models*. https://www.statmethods.net/advstats/cart.html

Sheppard, C. (2017). *Tree-based Machine Learning Algorithms: Decision Trees, Random Forests, and Boosting*. CreateSpace Independent Publishing Platform.

Vishal Sharma. (2017, June 9). *RPubs—Partial Dependence Plots: Quick Introduction in R*. https://rpubs.com/vishal1310/QuickIntroductiontoPartialDependencePlots