STAT 5400 Final Project

Tim Ambrose, Carter Huggins, Rebecca Rachan

December 5, 2016

1 Introduction

In this project, we introduce two R functions and give examples of their uses. The first R function, `dplyr` is a commonly used function for manipulating a dataset. We show an example of how `dplyr` can be used to create a dataframe and how to clean the data in order to produce a useful analysis. The second R function, `rpart` allows us to create classification trees. A classification tree is used to create a prediction model for categorical data. The model is obtained through recursively partitioning the data set and creating a simple prediction model within each partition. This is represented graphically as a decision tree. We will illustrate how to create a classification tree with the data set Titanic in the `MASS` library in R.

2 Creating a dataframe with `dplyr`

The first R function we considered was `dplyr` [4]. `dplyr` is a function that allows the user to work with a data frame. As a statistician, it is often times the case that the data sets we receive are not suitable in their current form for data analysis. This paper will illustrate an example of carrying out an analysis similar to what someone working as an applied statistician in the airline industry might be given as a project. This analysis involves manipulating a large and messy data set into something much more useful for analysis, as well as creating a model to predict arrival delay for flights. The data set used is called `nycflights13` and can be downloaded within R [3].

The first step in our analysis is loading in the necessary libraries into R, `nycflights13` and `dplyr`. We named the original data set Flights, and we attached all variable names to use throughout the code.

```
library(nycflights13)
library(dplyr)
Flights <- flights #name data set
attach(Flights)
```

The original data set Flights has 336776 observations and 19 variables. Displaying the first 10 observations with the `xtable` function in R, we have:

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time
1	2013	1	1	517	515	2.00	830	819
2	2013	1	1	533	529	4.00	850	830
3	2013	1	1	542	540	2.00	923	850
4	2013	1	1	544	545	-1.00	1004	1022
5	2013	1	1	554	600	-6.00	812	837
6	2013	1	1	554	558	-4.00	740	728
7	2013	1	1	555	600	-5.00	913	854
8	2013	1	1	557	600	-3.00	709	723
9	2013	1	1	557	600	-3.00	838	846
10	2013	1	1	558	600	-2.00	753	745

...
...

	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest
1	819	11.00	UA	1545	N14228	EWR	IAH
2	830	20.00	UA	1714	N24211	LGA	IAH
3	850	33.00	AA	1141	N619AA	JFK	MIA
4	1022	-18.00	B6	725	N804JB	JFK	BQN
5	837	-25.00	DL	461	N668DN	LGA	ATL
6	728	12.00	UA	1696	N39463	EWR	ORD
7	854	19.00	B6	507	N516JB	EWR	FLL
8	723	-14.00	EV	5708	N829AS	LGA	IAD
9	846	-8.00	B6	79	N593JB	JFK	MCO
10	745	8.00	AA	301	N3ALAA	LGA	ORD

	air_time	distance	hour	minute	time_hour
1	227.00	1400.00	5.00	15.00	1357016400.00
2	227.00	1416.00	5.00	29.00	1357016400.00
3	160.00	1089.00	5.00	40.00	1357016400.00
4	183.00	1576.00	5.00	45.00	1357016400.00
5	116.00	762.00	6.00	0.00	1357020000.00
6	150.00	719.00	5.00	58.00	1357016400.00
7	158.00	1065.00	6.00	0.00	1357020000.00
8	53.00	229.00	6.00	0.00	1357020000.00
9	140.00	944.00	6.00	0.00	1357020000.00
10	138.00	733.00	6.00	0.00	1357020000.00

Now, the data here has many missing values for some of the variables. We are going to focus our analysis on finding measures of centrality for departure delay (`dep_delay`) and arrival delay (`arr_delay`) to see if there are any major differences in these variables between different airlines. We want to create a new data set that eliminates all observations where an entry for departure delay or arrival delay is missing. One very useful command in `dplyr` is the piping command which is `%>%` and this is utilized to make code much more readable when running multiple functions in the same chunk of code. The `%>%` command passes the object on the left hand side as the first argument of the function on the right hand side of the symbol. The `filter` function allows you to choose the rows you want to keep using logical arguments so we will use it to keep the rows where entries for `dep_delay` and `arr_delay` are not missing. We do this with the following code:

```
na.rm.Flights <- flights %>% filter(!is.na(dep_delay), !is.na(arr_delay))
```

Now, we look at histograms of the variables `dep_delay`, `arr_delay`, and `distance` to see what the distributions look like. This gives us insight into what estimator is best used. After considering the histograms, we noticed that the data is skewed right for all three variables, with a few outliers on the left. Thus, we choose the median as the best estimator of centrality as the mean could overestimate this measure.

The next `dplyr` function we use is `group_by`. This tells R to run the commands that follow this function separately for each unique value in whichever variable we pass in to the `group_by` function. So, `group_by(carrier)` allows us to run other functions separately for each unique carrier. After grouping, we use the `mutate` function to create new variables. At this state, we are only concerned with `dep_delay`, `arr_delay` and `distance` so we create new variables for each of these using the median, our chosen measure of centrality. By grouping by carrier before mutating these new variables, the mutate function will calculate the median for each airline carrier separately.

Now we want to prune our data set from over 300,000 observations and 22 variables to just 16 rows (1 row for each carrier) and 4 variables. This is where the `select` and `summarise` functions are used. The `select` function allows us to select only the variables we want to keep from the data set and removes all other columns. The `summarise` function will then extract the unique values for each of the variables passed into it so in our case, it will extract one row for each unique carrier and its corresponding summary variables. The code to do all of this is actually very simple and is as follows and the resulting data set is shown in Table 1

```
FlightSummary <- na.rm.Flights %>% group_by(carrier) %>%
                  mutate(MedDepDelay = median(dep_delay),
                         MedArrivDelay = median(arr_delay),
                         MedDist = median(distance)) %>%
                  select(carrier, MedDepDelay, MedArrivDelay, MedDist) %>%
                  group_by(carrier, MedDepDelay, MedArrivDelay, MedDist) %>%
                  summarise()
```

	carrier	MedDepDelay	MedArrivDelay	MedDist
1	9E	-2.00	-7.00	509.00
2	AA	-3.00	-9.00	1096.00
3	AS	-3.00	-17.00	2402.00
4	B6	-1.00	-3.00	1023.00
5	DL	-2.00	-8.00	1020.00
6	EV	-1.00	-1.00	533.00
7	F9	0.00	6.00	1620.00
8	FL	1.00	5.00	762.00
9	HA	-4.00	-13.00	4983.00
10	MQ	-3.00	-1.00	502.00
11	OO	-6.00	-7.00	419.00
12	UA	0.00	-6.00	1400.00
13	US	-4.00	-6.00	529.00
14	VX	0.00	-9.00	2475.00
15	WN	1.00	-3.00	748.00
16	YV	-2.00	-2.00	229.00

Table 1: The Flight Summary Dataset

Suppose that instead of showing the flight carrier abbreviations, we want to show the actual names of the carriers. We can use `dplyr` to merge two data sets by one or more of their variables, much like we have done in SAS to accomplish this. The function `left_join` will generate a new data set that creates a new row for every match in the right data set (the second data set passed to the function) for each and every row of the left data set (the first data passed passed). Two potential problems exist with `left_join`. The first being that if there is not a match from the right data set, we are left with NA's for that row from the left data set. The other issue is that if there are multiple matches in the right data set, `left_join` will create multiple rows for that single row from the left data set which can sometimes be overkill. In our situation, both data sets have the same number of observations and match up perfectly so `left_join` works well. We merge our Flight Summary Data with a data set called airlines which has abbreviations as well as the full names, so we have a name for each carrier rather than the abbreviation. After merging, we use `select` to keep the variables we want and to get rid of the abbreviation. The code is below, and the resulting data set is in Table 2.

```
CarrierSummary <- left_join(FlightSummary, airlines, by = "carrier") %>% ungroup() %>%
                    select(name, MedDepDelay, MedArrivDelay, MedDist)
```

	name	MedDepDelay	MedArrivDelay	MedDist
1	Endeavor Air Inc.	-2.00	-7.00	509.00
2	American Airlines Inc.	-3.00	-9.00	1096.00
3	Alaska Airlines Inc.	-3.00	-17.00	2402.00
4	JetBlue Airways	-1.00	-3.00	1023.00
5	Delta Air Lines Inc.	-2.00	-8.00	1020.00
6	ExpressJet Airlines Inc.	-1.00	-1.00	533.00
7	Frontier Airlines Inc.	0.00	6.00	1620.00
8	AirTran Airways Corporation	1.00	5.00	762.00
9	Hawaiian Airlines Inc.	-4.00	-13.00	4983.00
10	Envoy Air	-3.00	-1.00	502.00
11	SkyWest Airlines Inc.	-6.00	-7.00	419.00
12	United Air Lines Inc.	0.00	-6.00	1400.00
13	US Airways Inc.	-4.00	-6.00	529.00
14	Virgin America	0.00	-9.00	2475.00
15	Southwest Airlines Co.	1.00	-3.00	748.00
16	Mesa Airlines Inc.	-2.00	-2.00	229.00

Table 2: The Carrier Summary Dataset

This completes the portion of the analysis that compares these different flight statistics between the 16 different airline carriers. The remaining portion is to create a model that will predict the arrival delay of any given flight.

By inference and by the above table, we can see that departure delay and distance of the flight will have an effect on the arrival delay. Another factor that we do not have yet in our data that could affect arrival time is the weather. To get this data, we illustrate merging two data sets a different way by using the function `inner_join` to join the flights data set with a weather data set to see what effect weather has on arrival time. The `inner_join` function is similar to the `left_join` function except that `inner_join` will not create rows where there is no match between the two data sets, those rows are simply thrown out so the resulting data set has no missing observations. From the weather data set, the most usable variables for our analysis are precipitation and visibility so after merging, we use `select` once again to keep only the variables we will use in our model. Similarly, this is done using the following code and the first 10 rows of the data set created are shown in Table 3.

```
FlightWeather <- inner_join(na.rm.Flights, weather, by =
                   c("time_hour"="time_hour", "origin" = "origin")) %>%
                   select(arr_delay, dep_delay, distance, precip, visib)
```

	arr_delay	dep_delay	distance	precip	visib
1	-25.00	-6.00	762.00	0.00	10.00
2	19.00	-5.00	1065.00	0.00	10.00
3	-14.00	-3.00	229.00	0.00	10.00
4	-8.00	-3.00	944.00	0.00	10.00
5	8.00	-2.00	733.00	0.00	10.00
6	-2.00	-2.00	1028.00	0.00	10.00
7	-3.00	-2.00	1005.00	0.00	10.00
8	7.00	-2.00	2475.00	0.00	10.00
9	-14.00	-2.00	2565.00	0.00	10.00
10	31.00	-1.00	1389.00	0.00	10.00

Table 3: The first 10 observations in the FlightWeather dataset

Now that we have cleaned and joined our data sets, we perform a brief analysis of the data. We first take a look at the correlations between variables to see if there will be any multicollinearity issues among the variables used. The correlations are shown below and from these, we will choose to create a model that uses all four of these predictors to predict the arrival delay.

```
 cor(FlightWeather)
           arr_delay   dep_delay     distance       precip       visib
arr_delay  1.00000000  0.91488090 -0.062003405  0.076545260 -0.14240124
dep_delay  0.91488090  1.00000000 -0.021566370  0.067633994 -0.12120996
distance  -0.06200340 -0.02156637  1.000000000  0.008074734 -0.01234325
precip     0.07654526  0.06763399  0.008074734  1.000000000 -0.31257545
visib     -0.14240124 -0.12120996 -0.012343249 -0.312575448  1.00000000

> summary(delay.lm)

Call:
lm(formula = arr_delay ~ dep_delay + distance + precip + visib,
    data = FlightWeather)

Residuals:
     Min       1Q   Median       3Q      Max
-106.397  -10.975   -1.902    8.683  188.305

Coefficients:
              Estimate Std. Error  t value Pr(>|t|)
(Intercept)  2.881e+00  1.582e-01   18.215  < 2e-16 ***
dep_delay    1.014e+00  7.870e-04 1287.944  < 2e-16 ***
distance    -2.598e-03  4.254e-05  -61.075  < 2e-16 ***
precip       1.329e+01  1.719e+00    7.734 1.04e-14 ***
visib       -6.551e-01  1.575e-02  -41.606  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 17.87 on 326164 degrees of freedom
Multiple R-squared:  0.8399,Adjusted R-squared:  0.8399
F-statistic: 4.277e+05 on 4 and 326164 DF,  p-value: < 2.2e-16
```

The above output from running the `summary` function on the model with all 4 predictors shows that all 4 of them are significant to the model. Thus, by using `dplyr` to add weather data to the model, we have created a more powerful model for predicting the arrival delay of a flight compared to a model that does not use weather data.

3 Classification Trees with `rpart`

For our second R tutorial, we will show how to create a classification tree. Again, a classification tree is a way to create a prediction model for categorical data. We use the Titanic data given in the MASS package [2]. This data classifies the passengers on the Titanic and gives the frequency of how many survived or died in a four-dimensional array. A snapshot of the original data set is given in Table 4. Since the object of interest is survival which has a binary outcome, the data is considered categorical and can be modeled using a classification tree. Note that the explanatory variables in this example (gender, social class, and age) are also categorical. The same techniques described here can be done with explanatory variables that are numerical, but the interpretation is a little less straightforward.

The data, as found in the MASS package, is in the form of a frequency table. While frequency tables are useful for examining the data, we need the raw version of the data in order to perform our analysis. We create this data set by replicating a separate observation for each passenger. Below is the code

```
library(MASS)
Titanic2=as.data.frame(Titanic)
Titanic4 <- Titanic2 %>% filter(Freq!=0) #defining a data frame that has no frequencies of 0
TitanicRaw=data.frame(Class=rep(NA,2201),Sex=rep(NA,2201),Age=rep(NA,2201),Survived=rep(NA,2201))
count=0
for(i in 1:24){ #loop for rows of original data frame
  for(j in 1:Titanic4[i,5]){ #loop for rows of new data frame
    for(k in 1:3){ #loop for columns
      TitanicRaw[count+j,k]=as.character(Titanic4[i,k])
    }
    if(Titanic4[i,4]=="Yes") TitanicRaw[count+j,4]="Survived"
    else TitanicRaw[count+j,4]="Did Not Survive"
  } #redefine "yes" and "no" to more intuitive names
  count=count+Titanic4[i,5] #keeping track of which row of TitanicRaw to write to next
}
View(TitanicRaw)
```

	Class	Sex	Age	Survived	Freq
1	1st	Male	Child	No	0.00
2	2nd	Male	Child	No	0.00
3	3rd	Male	Child	No	35.00
4	Crew	Male	Child	No	0.00
5	1st	Female	Child	No	0.00
6	2nd	Female	Child	No	0.00
7	3rd	Female	Child	No	17.00
8	Crew	Female	Child	No	0.00
9	1st	Male	Adult	No	118.00
10	2nd	Male	Adult	No	154.00

Table 4: The original Titanic frequency table

Now that our data is no longer in a frequency table, we can create the classification tree, using the package `rpart` [1]. Before we get into the function parameters, we will discuss what the function does. To start off, it places all of the observations into a single group called a "node". Then, it will run through various potential "splits" which are just ways of dividing up the data according to the explanatory variables. For instance, a split could divide the data up into men and women or young and old. It calculates which split best helps it determine the likelihood of survival. This creates two new nodes, and each of those nodes can be further split. It keeps splitting until it runs out of explanatory variables to split on, or until a specified parameter in the function call is achieved.

First, we loaded the package into R. Then, we used the `rpart` function to create our classification tree for the Titanic data set. The function takes in four arguments; formula, data, method, and control. The formula argument allows us to specify what outcome we want to model, and by which predictors. In our data set, our outcome we want to model is survival, and we want to do this by all predictors in the data set; namely age, gender, and social class. The second argument, data, simply specifies which data set we are obtaining the data from, which is our raw Titanic data set. The third argument, method, has two options: "class" or "anova". "Class" is used to create a classification tree whereas "anova" is used to create a regression tree. Regression trees are used to fit regression models on continuous data, and the code is similar. Finally, the last argument, control, allows the user to control tree growth. For example, `control=rpart.control(minsplit=30, cp=0.001)` forces the nodes to have more than 30 observations in order to do another split. Because we had a small number of predictors relative to our sample size, it was unnecessary for us to specify a control for our tree, so we left that argument blank.

The `rpart` function is all that is needed to create the classification tree, but there are many functions that can help us examine the results of the tree. First and foremost, `printcp` will display the cp table. Perhaps more usefully, `plotcp` allows us to visualize the cross validation results in the form of a plot as shown in Fig. 2. We can plot the tree itself in a few different ways. The most visually appealing is through the package `rpart.plot` which takes an `rpart` object and plots the nodes and splits. This is shown in Fig. 1. Finally, `summary` gives a detailed summary of each of the splits in the tree. Meaning, we know more about how many individuals survived in each group and how the algorithm decided to make a split. The code for creating the tree and examining the results is shown below.

```
library(rpart)
library(rpart.plot)
TitanicRaw.ct=rpart(Survived~.,data=TitanicRaw,method="class",)
printcp(TitanicRaw.ct) # display the results
plotcp(TitanicRaw.ct) # visualize cross-validation results
rpart.plot(TitanicRaw.ct) #display the tree
summary(TitanicRaw.ct) # detailed summary of splits
```

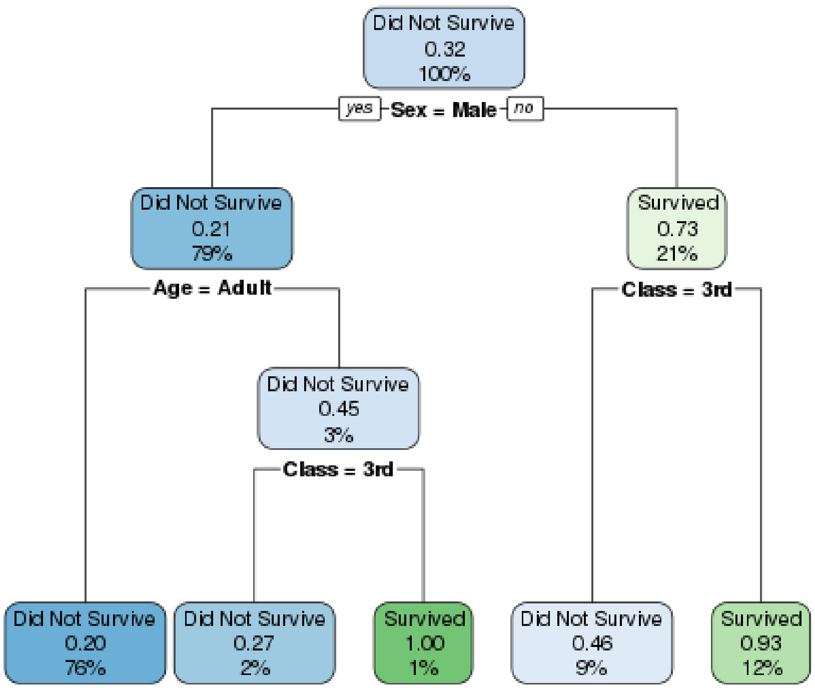

Figure 1: The classification tree for the Titanic data set

Let us examine the tree as shown in Fig. 1. Each box represents a node. The word or words at the top of each box show which class the observations are expected to be in. This is simply based on which class contains over half of the observations for that node. The decimal shows the proportion of the observations that survived. The percent at the bottom shows what percent of the total data is in that particular node. As an example for how to read the tree, let's say you wanted to know what the survival rate was for adult males. Sex=Male, so we would go the the left on the first split. Age=Adult, so we again go left and end up in the box on the bottom left. Therefore, we can see that adult males had a 0.20 survival rate.

This tree gives us some interesting results. First, notice that while the nodes on the male side are split based on both age and class, the nodes on the female side only split on class. Specifically, adult males had a low survival rate while male children had higher survival rates (particularly if they weren't third class). It would seem that on the Titanic, they placed more emphasis on saving the women and children. Another thing to notice is that the bulk of the data is in the bottom left

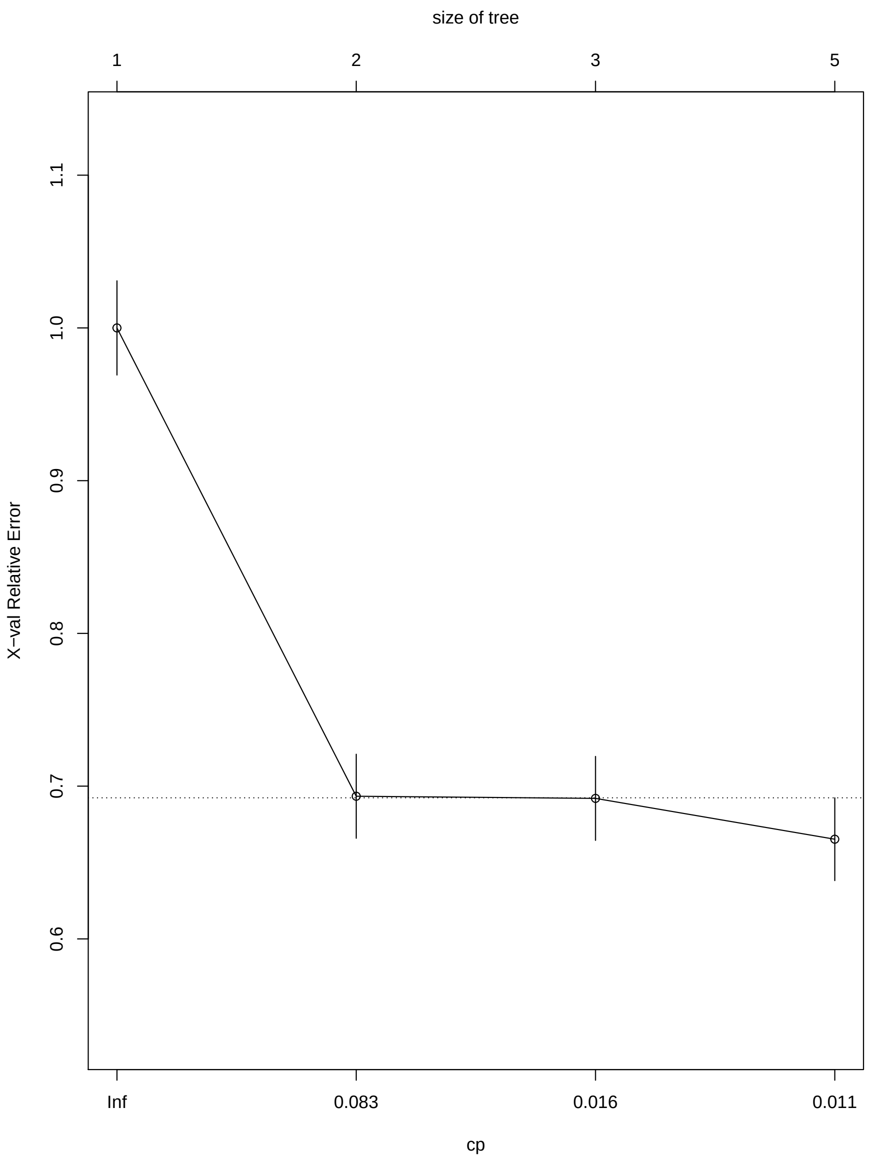

Figure 2: The Cross Validation Results

node, with very little in the male children nodes. This suggests the possible issues of overfitting, which we can examine in more depth by looking at the cp plot as will be discussed later.

Now we will examine the summary output. The following is only for the first node; calling the summary function will provide similar output for every single node for the tree. Note that the nodes are numbered left to right and top to bottom. Node number 1: 2201 observations, complexity param=0.3066104 predicted class=Did Not Survive expected loss=0.323035 P(node)=1 class counts: 1490 711 probabilities: 0.677 0.323 left son=2 (1731 obs) right son=3 (470 obs) Primary splits: Sex splits as RL, improve=199.821600, (0 missing) Class splits as RRLL, improve=69.684100, (0 missing) Age splits as LR, improve=9.165241, (0 missing) While you may find any part of the summary useful, one of the more important parts to examine is the "improve" value. This is an indication of how much the model can be improve by splitting on that particular variable. The higher the value, the better. Note that the probabilities listed are just the proportion of the data in that node that fit into each of the classes, Survived and Did Not Survive. When using a classification tree for predicting new observations, this will determine which class will be predicted. Note that this will only be considered for the final nodes whereas the example output is for the first node before splits have been made to refine the model.

The cross validation result shows us relative error of the number of nodes. The lower the relative error, the better. This helps us decide the optimal number of nodes, since often times the classification tree over-fits the model to the particular data set used for analysis. In our case the classification results show that we do not need to reduce the model since the full model has the lowest relative error. In general, with a large number of predictors, the relative error will go down initially before trending back up. In order to update the model to match the minimum relative error, `rpart` contains the function prune. An example for how to do that on the Titanic data set is as follows:

```
prunedfit=prune(TitanicRaw.ct,cp=TitanicRaw.ct$cptable[which.min(fit$cptable[,"xerror"]),"CP"])
```

This code will give you a new model in the form of an `rpart` object which you can use with all of the regular `rpart` functions for analysis and interpretation, such as `rpart.plot` and `plotcp`. Again, since our full model was actually the optimal model, this code will not remove any nodes from our tree.

To summarize, the classification tree gives a graphical representation of which factors the model uses to classify the passengers as survivors, or not survivors. This allows us to see which variables are most important in determining who survived. This can be used for prediction or to examine trends in the data and see what factors most influence survival rate.

References

[1] T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2015. R package version 4.1-10.

[2] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[3] H. Wickham. *nycflights13: Flights that Departed NYC in 2013*, 2016. R package version 0.2.0.

[4] H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2016. R package version 0.5.0.