Final Project: Analysis of delay in flights

Jeremy Colon-Shweta Anchan-Srinath Sridhar

June 27, 2014
INFO 7374 Data Science
Northeastern University

Contents

1 Introduction

We look to analyze airline data, which includes data about flights that are delayed, how long they have been delayed as well as the cause of the delay. We can combine this data with carrier name, origin airport and destination airport. By doing this, we have a myriad of options when considering different approaches for our analysis. We have the ability to cluster airports, predict delays and map the data as a directed graph for interesting visualizations.

2 Data Gathering

We obtained the data from the Research and Innovative Technology Administration (RITA) website. The website allows us to choose which fields we want, and download the data as a csv file. This process was relatively smooth when compared with most data gathering processes. We only pull data for January 2014, which includes over 470000 records.

3 Preprocessing

The preprocessing stage includes reading in the data, any lookup tables, and removing any unnecessary fields. It also involves removing any unwanted

records so that we run into less issues during our analysis, leading to more robust results.

3.1 Data Loading

We load the data simply by using the **read.csv** function.

```
library(ggplot2)
# Final Project Pre-Processing

setwd("E:/MS/Data Science/Final Project")

# Read in main data
flights <- read.table(file = "OnTime_Flight_data.csv",
    header = TRUE, sep = ",")

# Read in lookup tables
airports <- read.csv(file = "L_AIRPORT_ID.csv",
    header = TRUE)
carriers <- read.csv(file = "L_UNIQUE_CARRIERS.csv",
    header = TRUE)
weekdays <- read.csv(file = "L_WEEKDAYS.csv",
    header = TRUE)
cancellation.codes <- read.csv("L_CANCELLATION.csv",
    header = TRUE)
weather <- read.csv("Weather.csv",
    header = TRUE)
```

3.2 Data Cleansing

When cleaning the data, we have to look at what fields are relevant. We also join the lookup tables to the main data table so that we can bring in some descriptive values for departure and arrival locations, as well as carrier names. We do this using the **merge** function.

```
# Remove 'X' column that
# contains no data
flights <- flights[, -34]
# Merge main data set with
# lookup tables Get origin
# airport information
flights <- merge(x = flights, y = airports,
    by.x = "ORIGIN_AIRPORT_ID",
    by.y = "Code")
names(flights) <- c(names(flights[1:33]),
    "Origin_Airport_City", "Origin_Airport_State",
    "Origin_Airport_Description")
# Get destination airport
# information
flights <- merge(x = flights, y = airports,
    by.x = "DEST_AIRPORT_ID", by.y = "Code")
names(flights) <- c(names(flights[1:36]),
    "Dest_Airport_City", "Dest_Airport_State",
    "Dest_Airport_Description")
# Get carrier information
flights <- merge(x = flights, y = carriers,
    by.x = "UNIQUE_CARRIER", by.y = "Code")
names(flights) <- c(names(flights[1:39]),
    "Carrier_Name")
flights[1:5, ]$Origin_Airport_City
```

```
## [1] Dallas/Fort Worth
## [2] Raleigh/Durham
## [3] San Diego
## [4] Dallas/Fort Worth
## [5] Las Vegas
## 5434 Levels: 47-Mile Mine ...
```

Our next step is to remove unnecessary columns. This includes the following:

- Year, Month & Quarter: Since the entire data set is contained in January 2014.

- Date: We already have day of month and day of week fields. Date is redundant.
- Any ID fields we used for bringing in lookup values.

```
flights <- flights[, c(-2:-6, -9,
    -10:-11, -14:-15)]
flights <- data.frame(flights$Origin_Airport_Description,
    flights$Dest_Airport_Description,
    flights[, c(-23, -26)], stringsAsFactors = FALSE)
```

After removing unwanted fields, we are left with the following set of variables contained in the data:

- UNIQUE_CARRIER - A unique code given to each carrier (e.g "AA" - "$American Airlines$")$DAY\_OF\_MONTH - Numeric value signifying the day of the month.$

- DAY_OF_WEEK - Numeric value signifying the day of the week (1 - $Monday$, 7 - $Sunday$)$ORIGIN\_CITY\_NAME - Name of the departure city$
- ORIGIN_STATE_ABR - Two letter code of the departure state
- DEST_CITY_NAME - Name of the arrival city
- DEST_STATE_ABR - Two letter code of the arrival state
- DEP_TIME - Time of departure in military time
- DEP_DELAY - Number of minutes the flight departure was delayed. Negative numbers signify early departure.
- DEP_DELAY_NEW - Number of minutes the flight departure was delayed. Early departures have a value of zero.
- DEP_DEL15 - Binary variable signifying a departure delay of 15 minutes or more. 1 - 15+$minute delay$, 0 - 15$minute delay$.$ARR\_TIME - Time of arrival in military time$
- ARR_DELAY - Number of minutes the flight arrival was delayed. Negative numbers signify early arrival.
- ARR_DELAY_NEW - Number of minutes the flights arrival was delayed. Early arrivals have a value of zero.
- ARR_DEL15 - Binary variable signifying an arrival delay of 15 minutes or more. 1 - 15+$minute delay$, 0 - 15$minute delay$.$CANCELLED - Binary variable signifying if the flight wa$

- CANCELLATION_CODE - Specifies the reason for cancellation.
- DIVERTED - Binary variable signifying if the flight was diverted.
- CARRIER_DELAY - Carrier delay, in minutes.
- WEATHER_DELAY - Weather delay, in minutes.
- NAS_DELAY - National Air System delay, in minutes.
- SECURITY_DELAY - Security delay, in minutes.
- LATE_AIRCRAFT_DELAY - Late aircraft delay, in minutes.

Finally, we change our binary variables to factors.

4 Summaries & Visualizations

To get a feel for what the data looks like and if we need to subset our data, we run some summary statistics and simple visualizations.

4.1 Summaries

Below are summary tables for departure and arrival times. This tells us that flights are usually between 1.5 and 2 hours long, according to the differences between the two.

```
# Get summary statistics for
# numeric columns
summary(flights$DEP_TIME)
```

```
##    Min. 1st Qu.  Median
##       1     940    1330
##    Mean 3rd Qu.    Max.
##    1340    1730    2400
##    NA's
##   30327
```

```
# ARR_TIME
summary(flights$ARR_TIME)
```

```
##    Min. 1st Qu.  Median
##       1    1130    1530
##    Mean 3rd Qu.    Max.
##    1500    1920    2400
##    NA's
##   31496
```

Below are summary tables for the departure delay fields. Looking at textbfDEP_DELAY, we can see that the distribution is heavily skewed to the left, with the median being 0 and mean being 16.

```
# DEP_DELAY shows negative
# numbers for early departures
summary(flights$DEP_DELAY)
```

```
##    Min. 1st Qu.  Median
##    -112      -4       0
##    Mean 3rd Qu.    Max.
##      16      17    1560
##    NA's
##   30327
```

```
# DEP_DELAY_NEW shows 0 for
# early departures
summary(flights$DEP_DELAY_NEW)
```

```
##    Min. 1st Qu.  Median
##       0       0       0
##    Mean 3rd Qu.    Max.
##      18      17    1560
##    NA's
##   30327
```

```
summary(as.factor(flights$DEP_DEL15))
```

```
##      0      1   NA's
## 323347 118275  30327
```

After running the above summaries, we see that there are 30000 records that contain NA values for each variable. We want to get rid of these since the guts of our analysis revolves around flight delays.

We look further into the other numeric fields.

```
# Weather and Carrier delay
# summaries
summary(as.numeric(flights$CARRIER_DELAY))
```

```
##    Min. 1st Qu.  Median
##       0       0       4
##    Mean 3rd Qu.    Max.
##      20      20    1530
##    NA's
##  351955
```

```
summary(as.numeric(flights$WEATHER_DELAY))
```

```
##    Min. 1st Qu.  Median
##       0       0       0
##    Mean 3rd Qu.    Max.
##       5       0    1290
##    NA's
##  351955
```

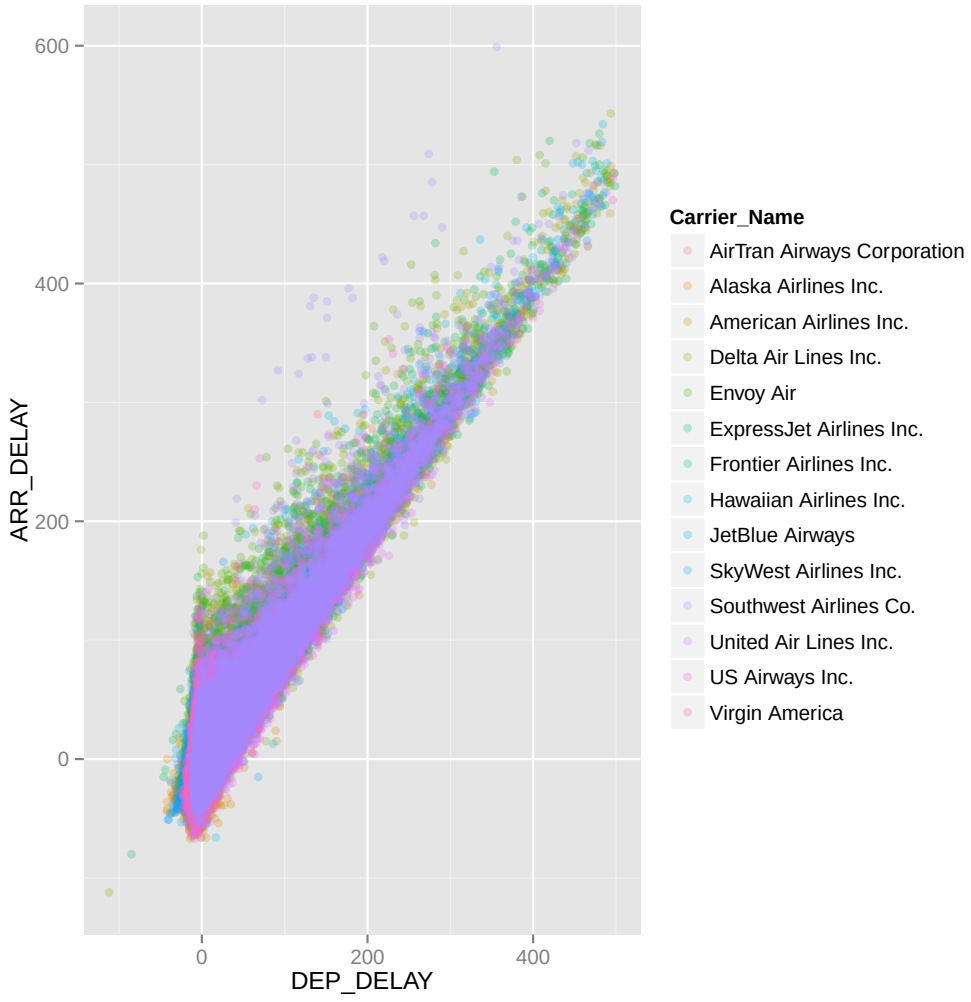
```
summary(as.numeric(flights$NAS_DELAY))
```

```
##    Min. 1st Qu.  Median
##       0       0       2
##    Mean 3rd Qu.    Max.
##      12      17     799
##    NA's
##  351955
```

The following graph is a scatterplot showing the relationship between departure and arrival delays. As expected, there is a linear relationship between the two. The later the departure delay, the later you will arrive.

```
carrier_table <- as.data.frame(table(flights$Carrier_Name))
colnames(carrier_table) <- c("airline",
    "freq")

carrier_table <- carrier_table[order(-carrier_table$freq),
    ]
```

```
# Get rid of airlines with no
# records
carrier_table <- carrier_table[carrier_table$freq >
    0, ]
# Plot carrier departure vs
# arrival delays
qplot(data = flights[flights$DEP_DELAY <
    500, ], x = DEP_DELAY, y = ARR_DELAY,
    colour = Carrier_Name, alpha = I(1/4))
```



5 Part1:Clustering

5.1 Introduction

We attempt to use Cluster Analysis to group airlines based on their delay times and the number of records for each airline in the data set. We will use the following libraries to perform cluster analysis.

```
library(sqldf)
library(cluster)
```

First, we create a data frame **df** with the columns we choose from the dataset i.e. **Carrier** and **Departure_DelayInMins**. Then we remove missing data with the **na.omit** function.

```
Carrier <- flights$Carrier_Name
Departure_DelayInMins <- flights$DEP_DELAY

df <- data.frame(Carrier, Departure_DelayInMins)
newdata <- na.omit(df)
```

Using the **sqldf** command we group the carriers based on the average delay in their departures and also extract the count of each carrier. Then we coerce the data into a numeric matrix **num** because if we use **command** for kmeans clustering, NAs are introduced by coercion.

```
command <- sqldf("SELECT Carrier, count(Carrier)as No_of_CarrierRecords, avg(Depar
```

```
## Loading required package:  tcltk
```

```
num <- data.matrix(command, rownames.force = NA)
```

5.2 Analysis

We know various clustering techniques that we can apply for our analysis. We pick K-means for our analysis.

5.2.1 K-means Clustering

The following code runs the k-means algorithm on our data and also aggregating the results to obtain the cluster means.

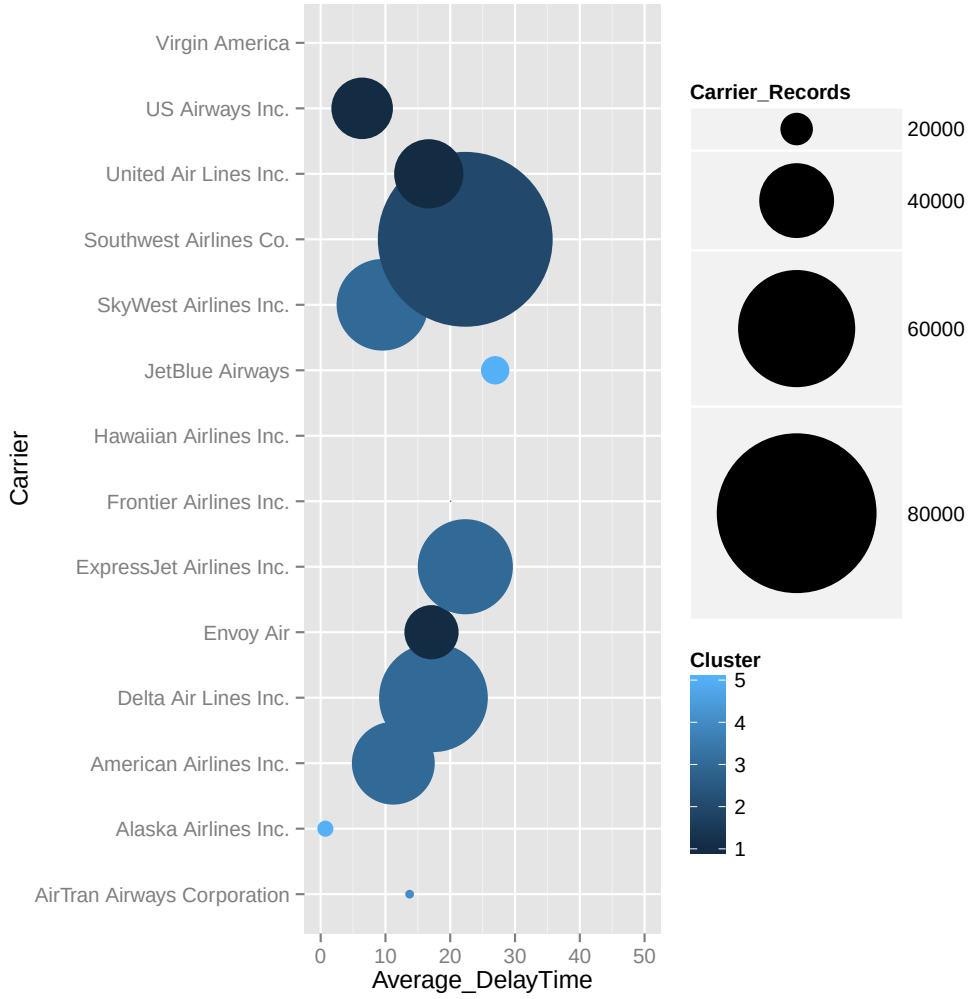
```
fit <- kmeans(num, 5)
aggregate(num, by = list(fit$cluster),
    FUN = mean)  #shows cluster means
```

```
##   Group.1 Carrier
## 1       1  1186.7
## 2       2  1316.0
## 3       3   686.8
## 4       4   791.5
## 5       5   536.5
##   No_of_CarrierRecords
## 1                33799
## 2                87161
## 3                49369
## 4                 6313
## 5                15169
##   Average_DelayTime
## 1            13.400
## 2            22.316
## 3            15.121
## 4             9.693
## 5            13.832
```

For plotting the graph, we take the **Average_DelayTime** and **Carrier** as the X and Y axis, respectively. The size of each bubble corresponds to **Carrier_Records**. Now using qplot, we visualize this data with the color of bubbles being the clusters that each of the Carriers belong to.

```
Carrier_Records <- command$No_of_CarrierRecords
Average_DelayTime <- command$Average_DelayTime
Carrier <- command$Carrier
Cluster <- fit$cluster

qplot(data = command, x = Average_DelayTime,
```

```
    y = Carrier, size = Carrier_Records,
    color = Cluster, xlim = c(0,
        50)) + scale_size(range = c(0,
    40))
```



5.3 Conclusion

K-means cluster analysis provided us with a decent distribution of clusters. Also we were able to extract meaningful information from this analysis. From

the graph above, we notice that the average delay of each Carrier does not depend on how popular it is.

6 PartII: Classification

6.1 Introduction

Classification is a data mining (machine learning) technique used to predict group membership for data instances.In this section we are going to analyze the data using association rule, decision tree and naive bayes algorithm.

6.2 Analysis

Before starting with the analysis, let us understand the data that we are going to use for it. We obtained weather data from **Accuweather.com** and integrated it into our main data using **merge** function.

6.2.1 Association Rules

```
library(arules)

flights.weather = merge(x = flights,
    y = weather, by.x = c("Origin_Airport_City",
        "DAY_OF_MONTH"), by.y = c("City",
        "Day"))
flights.weather = merge(x = flights.weather,
    y = weather, by.x = c("Dest_Airport_City",
        "DAY_OF_MONTH"), by.y = c("City",
        "Day"))

colnames(flights.weather)[30:37] <- c("origin.city.high.temp",
    "origin.city.low.temp", "origin.city.precip",
    "origin.city.snow", "dest.city.high.temp",
    "dest.city.low.temp", "dest.city.precip",
```

```
    "dest.city.snow")
```

The initial step here is to pick the attributes with which we are going to perform the analysis and apply **as.factor()** function over them to make them factors. We considered few weather attributes, categorical attributes and departure delay to work on the analysis. A data frame **updf** is created with attributes **depDelay**,**oCityName**,**depTime**,**orisnow**,**desSnow** and **cityName**.The **depDelay** attribute contains binary values where '0' represents "No Delay" and '1' represents "Delay".

The delay time attribute **depDelay** is set as the rhs value to find the combination of attributes causing the delay.

```
depDelay <- as.factor(flights.weather$DEP_DEL15)
# highTemp <-
# as.factor(flights.weather£dest.city.high.temp)
# lowTemp <-
# as.factor(flights.weather£dest.city.low.temp)
# snow <-
# as.factor(flights.weather£dest.city.snow)
# precip <-
# as.factor(flights.weather£dest.city.precip)
# cityName <-
# as.factor(flights.weather£DEST_CITY_NAME)

cityName <- flights.weather$DEST_CITY_NAME
ocityName <- flights.weather$ORIGIN_CITY_NAME
depTime <- as.factor(flights.weather$DEP_TIME)
desSnow <- as.factor(flights.weather$dest.city.snow)
oriSnow <- as.factor(flights.weather$origin.city.snow)

updf <- data.frame(depDelay, cityName,
    ocityName, depTime, desSnow,
    oriSnow)


apriori.appearance = list(rhs = c("depDelay=0"),
    default = "lhs")
```

```
apriori.appearance1 = list(rhs = c("depDelay=1"),
    default = "lhs")
```

The **support** and **confidence** level are pre-defined to get decent result.

```
apriori.parameter = list(support = 0.001,
    confidence = 0.001)
```

Now, the **apriori** function is applied over the obtained data to get the result. And, a subset is created and inspected by setting **confidence** and **support** level to reduce the number of combinations being printed. We created two rules to distinguish the combinations affecting the **depDealy** attribute.

```
rules = apriori(updf, parameter = apriori.parameter,
    appearance = apriori.appearance)
```

```
##
## parameter specification:
##  confidence minval smax arem
##       0.001    0.1    1 none
##   aval originalSupport
##  FALSE            TRUE
##  support minlen maxlen target
##    0.001      1     10  rules
##    ext
##  FALSE
##
## algorithmic control:
##  filter tree heap memopt load
##     0.1 TRUE TRUE  FALSE TRUE
##  sort verbose
##     2    TRUE
##
## apriori - find association rules with the apriori algorithm
## version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
## set item appearances ...[1 item(s)] done [0.00s].
## set transactions ...[1466 item(s), 119981 transaction(s)] done [0.07s].
## sorting and recoding items ... [489 item(s)] done [0.01s].
## creating transaction tree ... done [0.10s].
```

```
## checking subsets of size 1 2 3 4 5 done [0.04s].
## writing ... [1500 rule(s)] done [0.00s].
## creating S4 object  ... done [0.02s].
```

```
rules.subset = subset(rules, subset = lift >
    1 & confidence < 0.7)
```

The **rules** presents the combinations of attributes representing "on-time" flights.

```
inspect(head(sort(rules.subset,
    by = "lift"), 5))
```

```
##   lhs                              rhs           support confidence  lift
## 1 {cityName=Newark, NJ,
##    ocityName=Houston, TX,
##    oriSnow=0}                   => {depDelay=0} 0.001434     0.6992 1.011
## 2 {cityName=Denver, CO,
##    ocityName=Los Angeles, CA,
##    desSnow=0}                   => {depDelay=0} 0.002667     0.6987 1.010
## 3 {cityName=Denver, CO,
##    ocityName=Los Angeles, CA,
##    desSnow=0,
##    oriSnow=0}                   => {depDelay=0} 0.002667     0.6987 1.010
## 4 {cityName=Los Angeles, CA,
##    ocityName=Washington, DC}    => {depDelay=0} 0.002142     0.6984 1.010
## 5 {cityName=Los Angeles, CA,
##    ocityName=Washington, DC,
##    desSnow=0}                   => {depDelay=0} 0.002142     0.6984 1.010
```

The **rules1** presents the combination of attributes representing "delayed" flights.

```
rules1 = apriori(updf, parameter = apriori.parameter,
    appearance = apriori.appearance1)
```

```
##
## parameter specification:
##  confidence minval smax arem
```

```
##       0.001    0.1    1 none
##   aval originalSupport
##  FALSE            TRUE
##  support minlen maxlen target
##    0.001      1     10  rules
##    ext
##  FALSE
##
## algorithmic control:
##  filter tree heap memopt load
##     0.1 TRUE TRUE  FALSE TRUE
##  sort verbose
##     2    TRUE
##
## apriori - find association rules with the apriori algorithm
## version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
## set item appearances ...[1 item(s)] done [0.00s].
## set transactions ...[1466 item(s), 119981 transaction(s)] done [0.06s].
## sorting and recoding items ... [489 item(s)] done [0.01s].
## creating transaction tree ... done [0.09s].
## checking subsets of size 1 2 3 4 5 done [0.04s].
## writing ... [481 rule(s)] done [0.00s].
## creating S4 object  ... done [0.02s].
```

```
rules1.subset = subset(rules1,
    subset = lift > 1.4 & confidence <
        0.7)
inspect(head(sort(rules1.subset,
    by = "lift"), 5))
```

```
##   lhs                          rhs           support confidence  lift
## 1 {ocityName=Denver, CO,
##    oriSnow=0.21}          => {depDelay=1} 0.001334     0.6723 2.671
## 2 {desSnow=0,
##    oriSnow=0.21}          => {depDelay=1} 0.001350     0.6378 2.534
## 3 {desSnow=0.21,
##    oriSnow=0}             => {depDelay=1} 0.001325     0.6163 2.449
## 4 {ocityName=Chicago, IL,
```

```
##     desSnow=0,
##     oriSnow=0.2}           => {depDelay=1} 0.001292     0.6102 2.425
## 5 {ocityName=Chicago, IL,
##     oriSnow=0.2}           => {depDelay=1} 0.001542     0.6046 2.402
```

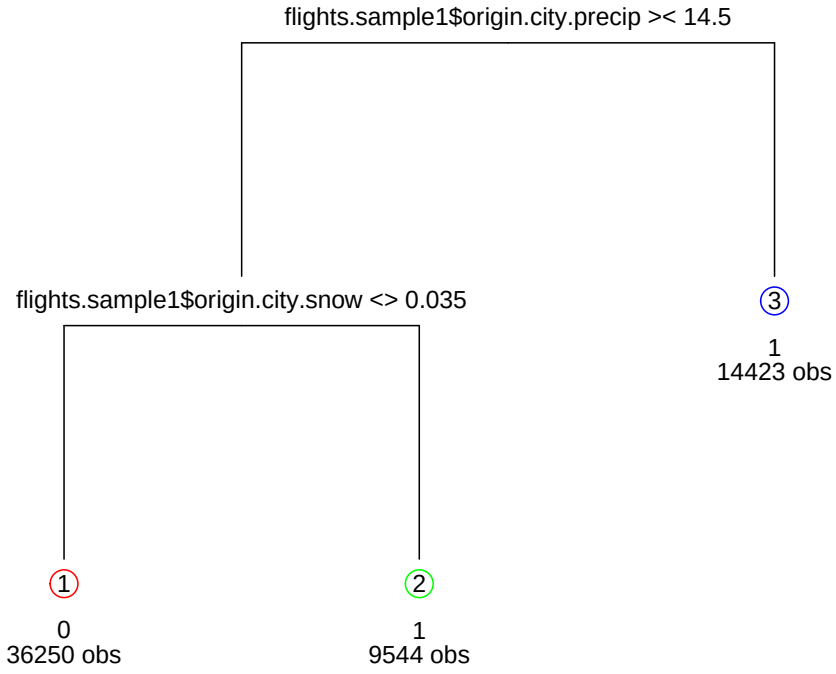
6.2.2 Decision Trees

We created the decision tree with the same attributes that we used in association rules. As our **depDelay** attribute contains just a binary value, it is easy to create a decision tree and check the factors influencing the result. The initial step to be done is to create a table with **depDelay** values in it. As the number of 0's in the attribute were way more than 1's, we got an error in using the attribute to create a tree. So, we obtained the number of 0's and 1's in the attribute and equalized the number of values by using **sample** function. Then, **rbind** is used to integrate the values and get the table.

```
library(rpart)
library(maptree)

# Sampling the dep.del15 to
# make 0's and 1's even.
flight.ontime <- flights.weather[which(flights.weather$DEP_DEL15 ==
    0), ]
flight.ontime <- flight.ontime[sample(1:nrow(flight.ontime),
    30019, replace = FALSE), ]
flight.delay <- flights.weather[which(flights.weather$DEP_DEL15 ==
    1), ]

flights.sample1 <- rbind(flight.ontime,
    flight.delay)
```

Basically, the tree is created by providing weather attributes like origin low temp, origin city snow and origin city precip to predict the **depDelay**. This is being mentioned in the **flight.formula**. Once we state a formula, **rpart** is applied over the given formula and the obtained data to get the tree.

```
flight.formula = as.factor(flights.sample1$DEP_DEL15) ~
    flights.sample1$origin.city.low.temp +
        flights.sample1$origin.city.snow +
        flights.sample1$origin.city.precip

flight.rpart = rpart(formula = flight.formula,
    data = flights.sample1)
draw.tree(flight.rpart)
```



The explanation for the above displayed tree is as follows:

if precip is greater than 14.5 **then**
  if snow is lesser than 0.035 **then**
    flights are on-time
  else
    flights gets delayed
  end if
else
    flights gets delayed
end if

Note that for the tree displayed above,we just predicted the factors influencing the departure delay alone. The same logic can be applied to find arrival delay too.

6.2.3 Naive Bayes

The third classification concept that we tried over the data is Naive Bayes. The intention here is to implement a classifier function with Naive Bayes classification and see how it predicts the delay based on the factors that we include in the formula.

When the algorithm was applied over]textbfflights.weather data, we could not obtain a decent result because the number of on-time flights were way higher than delayed flights. So we used the sampled version of **flights.weather** to work on the analysis.

We created a training set and a test set to split the analysis into two parts. We started the analysis by considering the attributes related to a flight departure. Then a data frame **newdf1** is created with the attributes **depDelay**,**desCityName**,**oriCityName**, **oriPrecip** and **uniqueCarrier**.

```
library(e1071)
index <- 1:nrow(flights.sample1)
trainindex <- sample(index, trunc(length(index)/2))
trainset <- flights.sample1[trainindex,
    ]
testset <- flights.sample1[-trainindex,
```

```
    ]

# With train data
newdf1 <- data.frame(trainset$DEP_DEL15,
    trainset$DEST_CITY_NAME, trainset$DEP_TIME,
    trainset$ORIGIN_CITY_NAME,
    trainset$origin.city.precip,
    trainset$UNIQUE_CARRIER)
```

Then **navieBayes** is applied over the data frame to check the predicted result. The result creates a matrix denoting the number of correct and wrong predictions done by the function. On interpreting the result, it is evident that the function has correctly predicted 9504 records of on-time flights and 9983 of delayed flights.

```
classifier <- naiveBayes(newdf1[,
    -1], as.factor(newdf1[, 1]))
table(predict(classifier, newdf1[,
    -1]), newdf1[, 1], dnn = list("predicted",
    "actual"))
```

```
##           actual
## predicted    0    1
##         0 9529 5126
##         1 5495 9958
```

The same logic is again followed on the test set to check the accuracy of the result. The function provides us with 64% accuracy.

```
newdf2 <- data.frame(trainset$DEP_DEL15,
    trainset$DEST_CITY_NAME, trainset$DEP_TIME,
    trainset$ORIGIN_CITY_NAME,
    trainset$origin.city.precip,
    trainset$UNIQUE_CARRIER)
classifiernew <- naiveBayes(newdf2[,
    2:6], as.factor(newdf2[, 1]))

table(predict(classifiernew, newdf2[,
    -1]), newdf2[, 1], dnn = list("predicted",
```

```
    "actual"))
```

```
##           actual
## predicted    0    1
##         0 9529 5126
##         1 5495 9958
```

The same concept is applied over the set of flights arriving to the airports.For this we consider the attributes associated with the destination.

```
df3 <- data.frame(trainset$ARR_DEL15,
    trainset$dest.city.snow, trainset$dest.city.precip,
    trainset$DEST_CITY_NAME, trainset$UNIQUE_CARRIER,
    trainset$ORIGIN_CITY_NAME,
    trainset$DIVERTED, trainset$ARR_TIME)

# With train data
classifier2 <- naiveBayes(df3[,
    2:8], as.factor(df3[, 1]))
table(predict(classifier2, df3[,
    -1]), df3[, 1], dnn = list("predicted",
    "actual"))
```

```
##           actual
## predicted     0     1
##         0 14406  8636
##         1  2481  4453
```

And again, the same attributes are being used to determine the effect on test set.

```
df4 <- data.frame(trainset$ARR_DEL15,
    trainset$dest.city.snow, trainset$dest.city.precip,
    trainset$DEST_CITY_NAME, trainset$UNIQUE_CARRIER,
    trainset$ORIGIN_CITY_NAME,
    trainset$DIVERTED, trainset$ARR_TIME)

# With test data
classifier3 <- naiveBayes(df4[,
    2:8], as.factor(df4[, 1]))
```

```
table(predict(classifier3, df4[,
    -1]), df4[, 1], dnn = list("predicted",
    "actual"))
```

```
##           actual
## predicted     0     1
##         0 14406  8636
##         1  2481  4453
```

6.3 Conclusion

The rpart function gives a good classification of result over any other decision tree functions. The Precipitation factor seems to have a good effect on the delay of flights. Also, we utilized few categorical variables to work on different concepts that in turn provided decent predictions about the delays.

7 Part III: Network Analysis

7.1 Introduction

Because our flight data includes an origin and destination airport, it lends itself perfectly to performing network analysis. To do so, we will structure our data set as a graph. Graphs are a relatively new concept when referring to data stores (especially in comparison to an RDBMS). They are particularly efficient at modeling connections (edges) between two entities (nodes), allowing your data model to be "white boarded". The data model for our flight data is straightforward; the airports represent nodes, and the flights between them represent edges. All other fields in our data set represent attributes of an edge.

7.2 Analysis

We split up our analysis into two parts; centrality analysis, and visualization.

7.2.1 Centrality Analysis

In order to get our centrality measures, we rearrange the the flight data frame so that the origin and destination cities are the first two fields. This is because the **graph.data.frame** function interprets the first two fields as the nodes and the rest of the fields as the attributes of each edge.

```
# Flight network data
library(igraph)
library(ggplot2)
library(ggmap)
library(maps)
library(geosphere)

flights_graph <- data.frame(origin.city.name = flights$ORIGIN_CITY_NAME,
    dest.city.name = flights$DEST_CITY_NAME,
    flights[, c(-4, -6)])

g <- graph.data.frame(flights_graph,
    directed = TRUE)
```

Now that the data is in a graph structure, we look at the vertex and edge count to make sure we have all of the data.

```
# Get vertex count
vcount(g)
```

```
## [1] 297
```

```
# Get edge count
ecount(g)
```

```
## [1] 471949
```

We look at the cities with the highest total degrees, as well as **in** and **out** degrees. **In** and **out** degrees signify direction in a directed graph. This means that an edge has a clear start node and end node, rather than just a generic connection between two nodes.

```
# Get cities with highest
# degree
```

```r
head(sort(degree(g), decreasing = TRUE))
```

```
##           Atlanta, GA
##                 61580
##           Chicago, IL
##                 56187
## Dallas/Fort Worth, TX
##                 46970
##           Houston, TX
##                 39285
##       Los Angeles, CA
##                 36955
##            Denver, CO
##                 35932
```

```r
head(sort(degree(g, mode = "out"),
    decreasing = TRUE))
```

```
##           Atlanta, GA
##                 30796
##           Chicago, IL
##                 28081
## Dallas/Fort Worth, TX
##                 23488
##           Houston, TX
##                 19649
##       Los Angeles, CA
##                 18481
##            Denver, CO
##                 17977
```

```r
head(sort(degree(g, mode = "in"),
    decreasing = TRUE))
```

```
##           Atlanta, GA
##                 30784
##           Chicago, IL
##                 28106
## Dallas/Fort Worth, TX
```

```
##                 23482
##           Houston, TX
##                 19636
##       Los Angeles, CA
##                 18474
##            Denver, CO
##                 17955
```

You can see that cities with the highest degree are major hubs. Next, we will start looking at various centrality measures to explore the graph further.

The diameter of a graph measures the width. The width of a graph tells you the longest connection string, i.e. the "longest shortest path". More formally, it is the largest number of vertices which must be traversed in order to travel from one vertex to another, excluding paths that backtrack, detour, or loop. In this case, the width of the graph is four. We can look to see the "longest shortest path" using the **farthest.nodes** and **get.shortest.paths** functions.

```
# Diameter of graph, i.e. how
# wide is it?
diameter(g)
```

```
## [1] 4
```

```
fn <- farthest.nodes(g)
fn
```

```
## [1]  2 92  4
```

```
sp <- get.shortest.paths(g, from = V(g)[fn[1]],
    to = V(g)[fn[2]])
V(g)[sp$vpath[[1]]]
```

```
## Vertex sequence:
## [1] "Raleigh/Durham, NC"
## [2] "Dallas/Fort Worth, TX"
## [3] "Seattle, WA"
## [4] "Juneau, AK"
## [5] "Sitka, AK"
```

We can see that if you want to get to Bethel, AK, you must start at Worcester and travel through Fort Lauderdale, Houston, and Anchorage.

An interesting aspect of a graph is a clique. Cliques identify groups of vertices where each vertex is connected to each other vertex. In our graph, we actually have 38 cliques.

```
# Can get the largest cliques
# of the graph. Cliques
# identify groups of vertices
# where each vertex is
# connected to each other
# vertex.
lc <- largest.cliques(g)
```

If we look at the structure of **lc**, we see that we have a list of 38 cliques of size 21. This means that there are 38 different sets of 21 airports where you can fly direct between them. Also, we looked at the vertex numbers of each clique and noticed that the first 8 elements are identical across all of them.

```
str(lc)

## List of 38
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 24 ...
```

```
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
##  $ : num [1:21] 5 13 1 59 42 4 35 58 9 6 ...
```

Curious about which these are, we looked at a subset of the first list.

```
lc[[1]]
```

```
##  [1]  5 13  1 59 42  4 35 58
##  [9]  9 24 12 31 44 57 16 17
## [17] 63 19 14 38 65
```

```
V(g)[lc[[1]][1:8]]
```

```
## Vertex sequence:
## [1] "Chicago, IL"
## [2] "Atlanta, GA"
## [3] "Dallas/Fort Worth, TX"
## [4] "Houston, TX"
## [5] "Denver, CO"
## [6] "Las Vegas, NV"
```

```
## [7] "Phoenix, AZ"
## [8] "Minneapolis, MN"
```

The results show us 8 major hubs in the US, the first 5 representing 5 out of the top 6 airports with regards to the degree (number of flights). Next, we look at the centrality measures to figure out which airports are the most connected.

We looks at closeness and farness first. Farness of a node is defined as the sum of its distances to all other nodes, and its closeness is defined as the inverse of the farness. Closeness can be regarded as a measure of how long it takes for information to spread from node s to all other nodes.

For this graph, the closeness can be regarded as which airport is most central to other airports such that if you wanted to travel from one city to another, this airport would usually yield the shortest distance.

```
# Get closeness and farness
# metrics
c <- closeness(g)
f <- 1/c

head(sort(c, decreasing = TRUE))
```

```
##           Chicago, IL
##              0.002242
##           Atlanta, GA
##              0.002198
## Dallas/Fort Worth, TX
##              0.002155
##            Denver, CO
##              0.002137
##           Houston, TX
##              0.002053
##       Minneapolis, MN
##              0.001988
```

```
head(sort(f))
```

```
##           Chicago, IL
```

```
##                   446
##           Atlanta, GA
##                   455
## Dallas/Fort Worth, TX
##                   464
##            Denver, CO
##                   468
##           Houston, TX
##                   487
##       Minneapolis, MN
##                   503
```

Not surprisingly, due to being the major mid-west hub in the US, Chicago has the highest closeness value.

Betweenness measures the number of times a node acts as a bridge along the shortest path of two other nodes. In this case, which airport is involved in the most connections?

```
# Get betweenness
b <- betweenness(g)
head(sort(b, decreasing = TRUE))
```

```
##           Chicago, IL
##                 20047
##           Atlanta, GA
##                 19702
## Dallas/Fort Worth, TX
##                 15574
##            Denver, CO
##                 11194
##       Minneapolis, MN
##                  6896
##     San Francisco, CA
##                  6230
```

We show Atlanta, not Chicago, has the highest betweenness centrality. Not completely surprising given that Atlanta had the second highest closeness value behind Chicago.

Transitivity measures the relative number of triangles in a graph, compared to the total number of connected triples of nodes. In layman's terms, how many friends of my friends am I also friends with? In this case, how many triples of airports are completely connected versus the total number of triples.

```
# Get transitivity
transitivity(g)
```

```
## [1] 0.3193
```

To give some context on the result, most social graphs have a transitivity between 0.3 and 0.6.

Eigenvector centrality is similar to Google' Page Rank algorithm. It is the measure of the influence of a node in a network.It assigns relative scores to all nodes based on the concept that connections to high scoring nodes contribute more to the score than connections to low scoring nodes.

```
# Eigenvector centrality
head(sort(evcent(g)$vector, decreasing = TRUE))
```

```
##           Chicago, IL
##                1.0000
##           Atlanta, GA
##                0.9554
##       Los Angeles, CA
##                0.8478
##          New York, NY
##                0.8466
## Dallas/Fort Worth, TX
##                0.7483
##           Houston, TX
##                0.7409
```

The results show that, even though Atlanta has the highest betweenness centrality, Chicago has the highest eigenvector centrality, which says that Chicago is directly connected with other highly connected nodes.

7.2.2 Visualization

Some of the more appealing work that comes out of network analysis is the type of visualizations you can do to get a better understanding of what the data looks like. To reduce clutter, we take a random sample of 5000 flights without replacement and create a graph. Next, we got the latitude and longitude by geocoding the data using Google's API. We then created a data frame of flights to merge with the geocoded data.

The code is commented out because we output the geocoded data to a file so that the geocode doesn't happen on every PDF compilation. The code below simply reads the file into a data frame for us to use.

```
# Get sample for network
# visualization flights_sample
# <-
# flights_graph[sample(1:nrow(flights_graph),5000,replace=FALSE),]
# g.sample <-
# graph.data.frame(flights_sample,directed
# = TRUE)
edge.delays <- read.table("edge_delays_v2.txt",
    sep = "|", header = TRUE)
```

To visualize the sample network, we create a color palette for our edges and set limits for the latitude and longitude of our map.

```
# Map connections library(maps)
# library(geosphere)

# Create color palette
pal <- colorRampPalette(c("#FFFFFF",
    "#FFBFBF", "#FF8080", "#FF4040",
    "#FF0000"))
colors <- pal(100)

# set x and y limits for map
# xlim <- c(-171.738281,
# -56.601563) ylim <-
# c(12.039321, 71.856229)
```

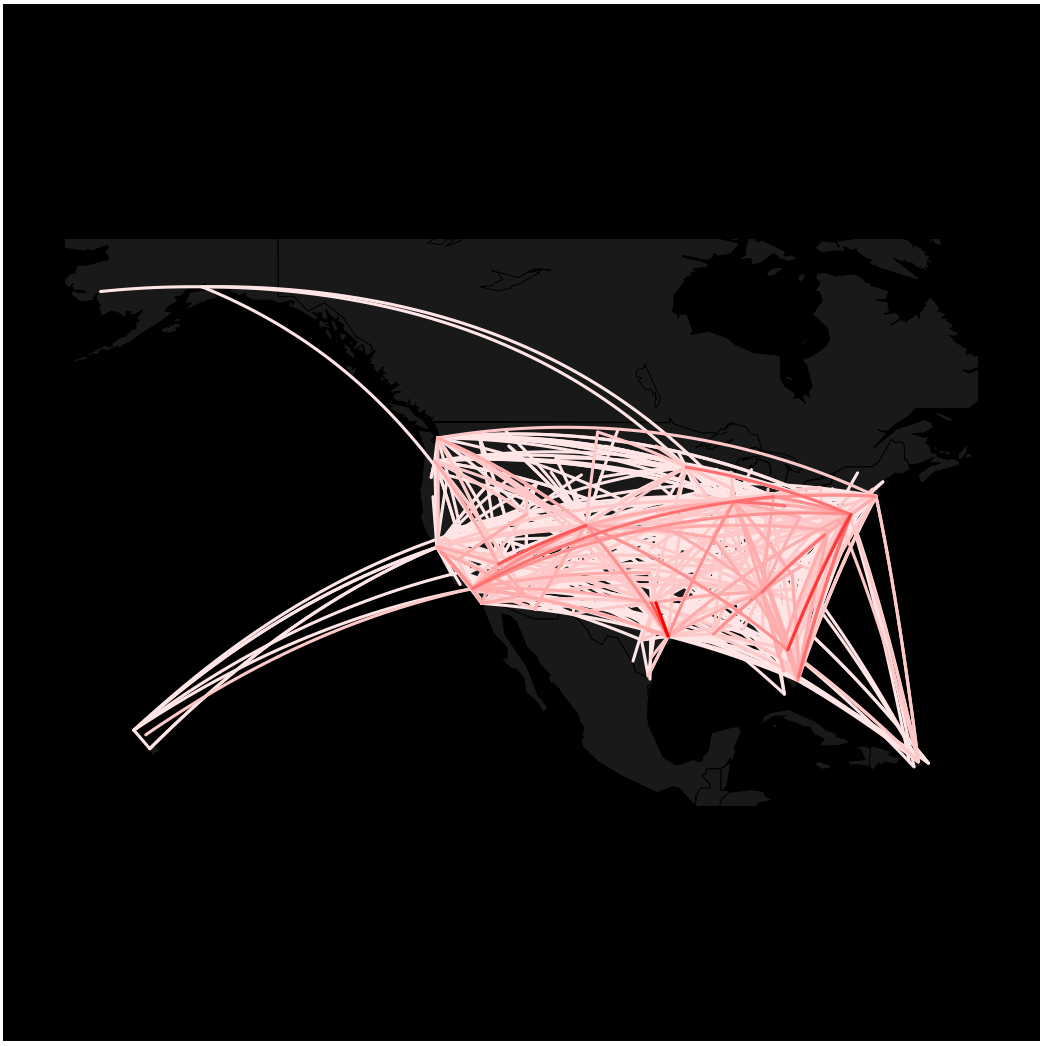
```
xlim <- c(-165, -60)
ylim <- c(15, 65)
```

Next, we order the connections based on delay count so that the most delayed connections get plotted last.

```
edge.delays <- edge.delays[order(edge.delays$delays),
    ]

maxcnt <- max(edge.delays$delays)
```

We now plot each line one by one on our map and color the lines based on delay count.

```
# plot lines on the map
plot.new()

map("world", col = "#191919", fill = TRUE,
    bg = "#000000", lwd = 0.05,
    xlim = xlim, ylim = ylim)

for (j in 1:nrow(edge.delays)) {
    inter <- gcIntermediate(c(edge.delays[j,
        ]$orig.lon, edge.delays[j,
        ]$orig.lat), c(edge.delays[j,
        ]$dest.lon, edge.delays[j,
        ]$dest.lat), n = 100, addStartEnd = TRUE)
    colindex <- round((edge.delays[j,
        ]$delays/maxcnt) * length(colors))
    lines(inter, col = colors[colindex],
        lwd = 2)
}
```



From looking at the map, we can see that it is flooded by a lot of "white" lines. These are most likely less trafficked routes. To clear up the map, we filter the lines that get plotted by setting a lower bound of the colindex variable.

```
# Create new color palette to
# account for less lines
pal <- colorRampPalette(c("#FFFFFF",
    "#FFBFBF", "#FF4040", "#FF0000"))
colors <- pal(100)
```

```
xlim <- c(-165, -60)
ylim <- c(15, 55)

# Create world map
map("world", col = "#191919", fill = TRUE,
    bg = "#000000", lwd = 0.05,
    xlim = xlim, ylim = ylim, )

# Plot lines if colindex meets
# requirements

for (j in 1:nrow(edge.delays)) {
    inter <- gcIntermediate(c(edge.delays[j,
        ]$orig.lon, edge.delays[j,
        ]$orig.lat), c(edge.delays[j,
        ]$dest.lon, edge.delays[j,
        ]$dest.lat), n = 100, addStartEnd = TRUE)
    colindex <- round((edge.delays[j,
        ]$delays/maxcnt) * length(colors))
    # After checking the
    if (colindex > 15) {
        lines(inter, col = colors[colindex],
            lwd = 2)
    }
}
```

The resulting visualization is much less cluttered and more aesthetically pleasing. It also more clearly shows where most of the delays are.

7.3 Conclusion

From the centrality measures, we essentially confirm where the hubs are in the US. What is interesting is where the hubs are and how that ties into the number of delays. Unfortunately, as people we can't really avoid these large

airports that are the biggest offenders with regards to delays. Chicago and Atlanta being the biggest culprits.