Uber Data Challenge

Chris Dailey

As Chapter 2 of the *Chris Dailey Applies for a Job* straight-to-DVD miniseries, our hero must demonstrate worth by executing some quality data science to prove he isn't making stuff up.

Part 1 consists of some basic data manipulation and time series analysis. Part 2 is about experimental design. Finally, Part 3 is about predictive modeling. We'll switch between two programming languages, R and Python. R has much better and cleaner statistical functions and the regression functions are much more concise. Python has machine learning libraries that are much more consistent and convenient. R has them as well, but they're something of a patchwork of independently developed solutions. In short, we'll use R for data exploration and regression, and we'll use Python for advanced machine learning.

We begin with Part 1, our hero already neck-deep in science...

Part 1: Time Series Analysis

The task: given a dataset of timestamps representing login times, discover the underlying patterns and discuss any data issues.

We'll be working in R for the entirety of this question. First, we set up our environment and load the data.

```
library(rjson)
library(ggplot2)
library(grid)
library(gridExtra)
library(scales)

raw_data = fromJSON(file = "logins (3).json")
str(raw_data)

## List of 1
##  $ login_time: chr [1:93142] "1970-01-01 20:13:18" "1970-01-01 20:16:10" "
1970-01-01 20:16:37" "1970-01-01 20:16:36" ...
```

Modify the data a bit to make it more convenient. Since nothing is nested there's no need to work in json.

```
timestamps = as.POSIXlt(raw_data$login_time)
length(timestamps)

## [1] 93142
```

```
max(timestamps) - min(timestamps)
```

```
## Time difference of 101.9482 days
```

```
head(timestamps, n=20)
```

```
##  [1] "1970-01-01 20:13:18 EST" "1970-01-01 20:16:10 EST"
##  [3] "1970-01-01 20:16:37 EST" "1970-01-01 20:16:36 EST"
##  [5] "1970-01-01 20:26:21 EST" "1970-01-01 20:21:41 EST"
##  [7] "1970-01-01 20:12:16 EST" "1970-01-01 20:35:47 EST"
##  [9] "1970-01-01 20:35:38 EST" "1970-01-01 20:47:52 EST"
## [11] "1970-01-01 20:26:05 EST" "1970-01-01 20:31:03 EST"
## [13] "1970-01-01 20:34:46 EST" "1970-01-01 20:36:34 EST"
## [15] "1970-01-01 20:39:25 EST" "1970-01-01 20:40:37 EST"
## [17] "1970-01-01 20:34:58 EST" "1970-01-01 20:43:59 EST"
## [19] "1970-01-01 20:47:46 EST" "1970-01-01 20:46:34 EST"
```

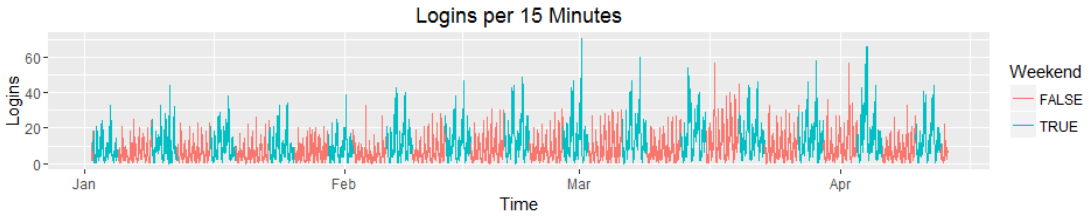
The data speaks. A little over 93k datapoints covering about 102 days in a standard format. The months, dates, and times all seem fine, but the year is suspicious. This is common for systems that don't record the year; in most cases, *nix systems will default to the epoch, Jan 1 1970, when parts of a date are missing. This means the year portion of the data is meaningless and will be something of an issue later as it will limit our ability to figure out which day of the week a timestamp is from. The prompt specifies that the timestamps are all taken from a specific location, so the time zone can also be ignored.

We'll bucket the data to allow for time series analysis. We'll also chop it up a bit and format it some to make it easier to work with.

```
#Mark out the range of coverage
start_time = min(timestamps)
end_time = max(timestamps)
#define the bins
buckets = seq.POSIXt(from = start_time, to = end_time, by = "15 min")
#Bin, tabulate, and convert to a data frame
hist1 = data.frame(table(cut.POSIXt(timestamps, buckets)))
names(hist1) = c("Bucket", "Logins")
#Format the time
hist1$Bucket = as.POSIXlt(hist1$Bucket)
#Mark weekends (Fri, Sat, Sun) for easier distinction
hist1$Weekend = hist1$Bucket$wday %in% c(5,6,0)
```

The first step is always to look at the data.

```
ggplot(hist1, aes(Bucket, Logins, color = Weekend, group = 1), size = 1) + ge
om_line() + scale_x_datetime() + labs(x="Time", title="Logins per 15 Minutes"
)
```



We see some trends that shouldn't surprise. A spike each day with higher spikes during the weekends. It also appears that usage is increasing in general over the covered period. We'll quantify the three trends: daily, weekly and long term. First, though, we'll review the theoretical basis for our analysis.

Time Series Basics

Feel free to skip this part if you'd rather avoid the mathematical side of life.

We represent a time series of length n by $\{x_t: t = 1, \ldots, n\} = \{x_1, x_2, \ldots, x_n\}$. It consists of n values sampled at discrete times $1,2, \ldots, n$. The notation will be abbreviated to $\{x_t\}$ when the length n of a series doesn't need to be specified.

We expect this series to be dominated by trends and seasonal effects. We can isolate each component: the trend, the seasonal effect, and the underlying stochastic process (the "random"). We can decompose this in two ways. An additive decomposition is expressed mathematically as

$$x_t = m_t + s_t + z_t$$

where, at time t, x_t is the observed series, m_t is the trend, s_t is the seasonal effect, and z_t is the error term representing the stochastic component (a series of random variables with mean zero).

If the seasonal effect depends on the trend, a multiplicative decomposition is appropriate:

$$x_t = m_t * s_t + z_t$$

Finding the seasonal effect and separating it from the trend depends on the period (k)of the phenomenon we expect to find. We simply calculate a moving average over a span of k observations[1]. For example, if we expect to find a weekly trend across daily observations, we could estimate the trend as:

$$\hat{m}_t = \frac{x_{t-3} + x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2} + x_{t+3}}{7}$$

Note the ^ above the m indicating an estimation. Once we have the estimated trend, the estimated additive seasonal effect can be derived by subtracting the trend:

[1] There's a special case when *k* is even since there's no center observation, but it's an easy fix; increase the width by one and half the weight of the outer observations.

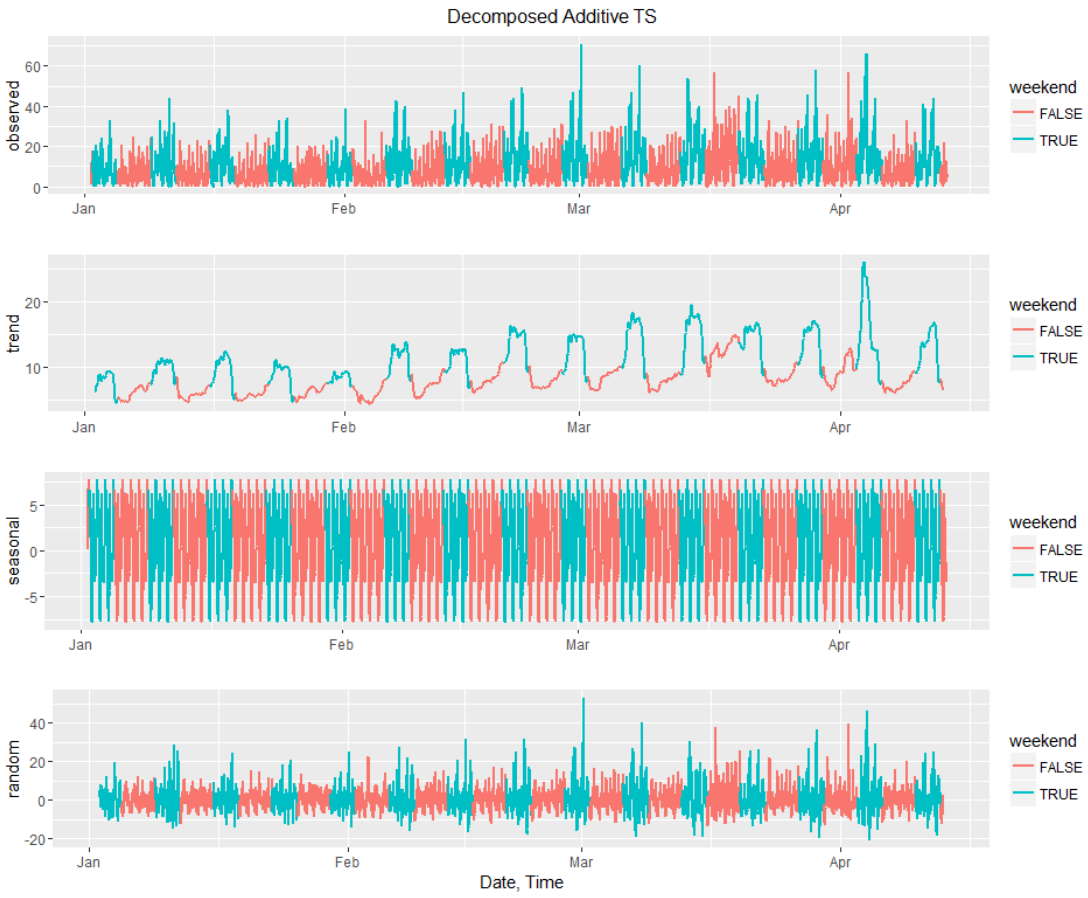
$$\hat{s}_t = x_t - \hat{m}_t$$

Finally, the stochastic component is all that's left when the trend and seasonal effects are subtracted from the observed series. Ideally, this remaining component will be white noise, i.e, a series of random values with mean zero and constant variance with no dependence on previous observations. If it's not, some variance is not captured by our model. That won't stop us from deriving meaningful insights, but it will limit our ability to make predictions from our model. We should also be aware of the unaccounted for variance when assigning causality. There are procedures and models to account for this, but beyond the basic insights of the model we won't need them to reach our goals.

Daily Trends

To look at daily trends, we assign our time series a period of $4 * 24$, since we have 4 observations per hour and there are 24 hours in a day and we want to recover daily trends. We decompose the time series and take a look at our trends.

```
#build a time series
ts1 = ts(data = hist1$Logins, start = c(1,1), frequency = (4 * 24))
#this is a function defined at the end of the report
decompose_timeseries(ts1, hist1)
```



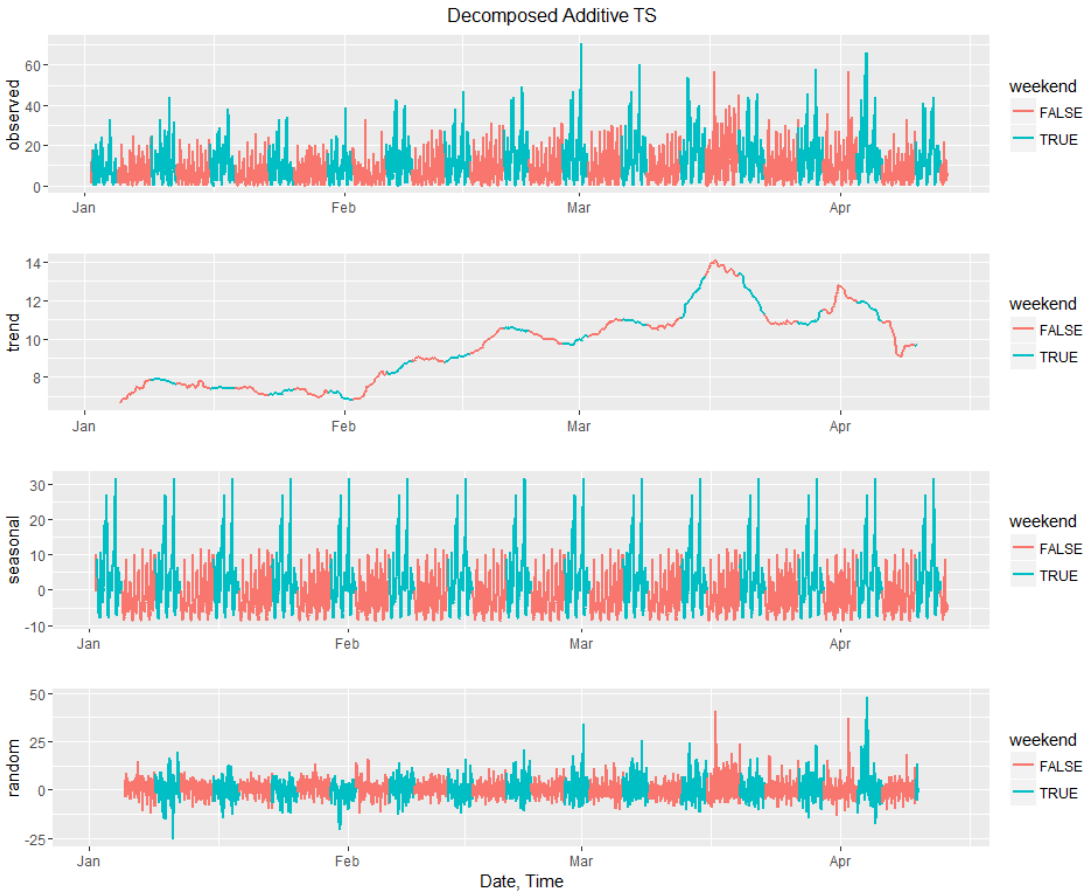
When viewed at a daily scale, the results make sense. The seasonal trend indicates two spikes a day, one larger than the other. The trend shows that weekends are much busier as expected, and that the spikes on weekend days are bigger than the spikes during the week. There are two major concerns with this model. First, the seasonal effects don't differentiate between weekdays and weekends which are intuitively and empirically different, so trends that might be specific to each are lost. Second, the random component, while nicely centered at zero, does not resemble white noise; there is unaccounted for variance indicated by a changing range of variance in the random component. Further, the unaccounted for variance is quite large; the random component is much larger than the seasonal effects which limits our ability to make predictions, empirical or intuitive.

We will leave the second largely unaddressed. We'll see that it becomes less significant when we account for specific trends and while we'll briefly demonstrate a means of correcting it, we won't pursue it much. Also, it could simply be that there is a non-stochastic variable driving change; for example, spring break might induce an influx of tourists; the seasonal effect of the day of the week would not account for this.

Easily addressable, though, is the distinction between weekdays and weekends. There are several ways to approach this, but we'll use the most straightforward for now: widening the period to a week will allow our seasonal effects to account for the distinction between weekdays and weekends.

Weekly Trends

```
#note the increased frequency
ts2 = ts(data = hist1$Logins, start = c(1,1), frequency = (4 * 24 * 7))
decompose_timeseries(ts2, hist1)
```

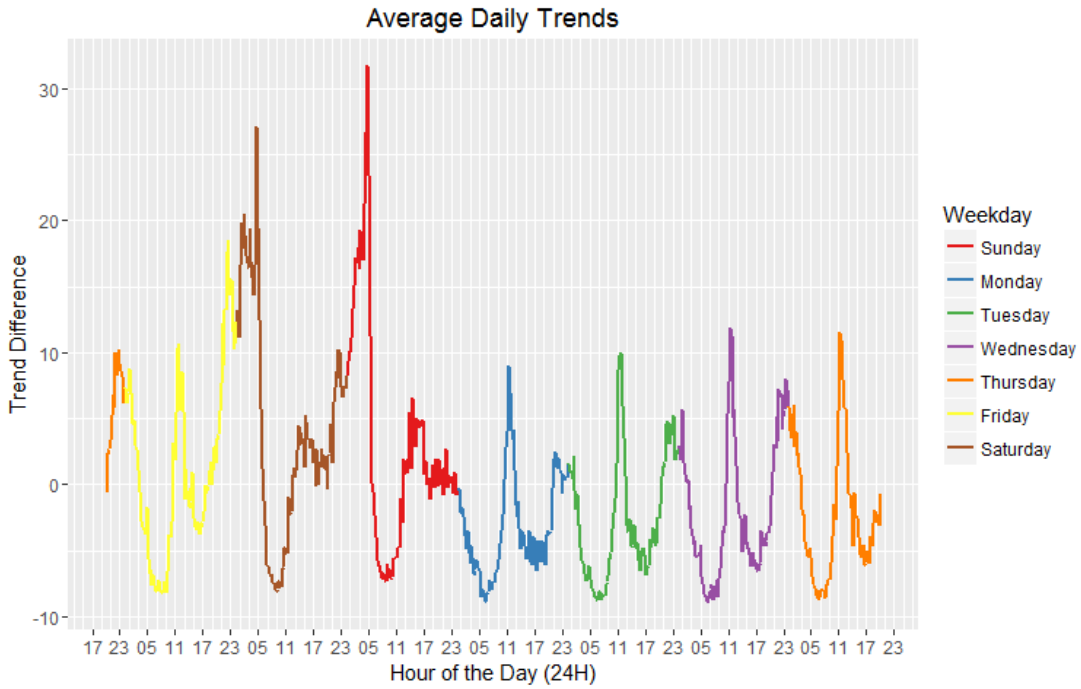


Now we see some things previously hidden in the data! First, we notice that in the long term, usage is indeed increasing, and by quite a lot. Second, and here's what we came for, our seasonal effects capture the difference between weekdays and weekends. This will allow us to identify typical usage patterns on weekdays and weekends, which we'll attend to in a moment. First, we must address the random component; it is still not white noise and the variance doesn't seem constant. If we were to build a model for this, we would first prove to ourselves that the random component wasn't white noise by reviewing its partial self-correlation. The moving trend implies a non-stationary series and the changing variance plus the self-correlation we'd certainly find implies a GARCH model would make the most sense. In this case, though, we needn't build a model, just notice trends.

Let's look more closely at the weekly usage patterns on a day-to-day basis

```
#break down the series into components
ts2.decomp.add = decompose(ts2)
#format it for easy use
ts2.seasonal = data.frame(bucket=hist1$Bucket, seasonal = ts2.decomp.add$seas
onal)
#extract just one week and format it
one_week = ts2.seasonal[0:(7*24*4),]
one_week$bucket = as.POSIXlt(one_week$bucket)
one_week$Weekday = weekdays(one_week$bucket)
```

```
one_week$Weekday = factor(one_week$Weekday, levels = c("Sunday", "Monday", "T
uesday", "Wednesday", "Thursday", "Friday", "Saturday"))

#Plot the data
ggplot(one_week) +
  geom_line(aes(bucket, seasonal, color = Weekday, group=1), size=1) +
  scale_color_brewer(palette="Set1") +
  scale_x_datetime(breaks= date_breaks("6 hours"), minor_breaks = date_breaks
("2 hour"), labels = date_format("%H", tz = "EST")) +
  labs(title="Average Daily Trends", x="Hour of the Day (24H)", y="Trend Diff
erence")
```



As we saw earlier, weekends and weekdays exhibit different trends. Weekdays consistently spike at lunch (with a mini spike after, perhaps for the trip back to the office) and from 9pm to 1am. Weekends show a predictable spike in line with nightlife activity: higher ridership on Friday night around 10:30pm with sustained usage until the main spike when the bars and clubs close around 5am.

We mentioned earlier about the ambiguity of using a year we know is wrong to determine the day of the week, but the patterns line up in a sensible way, so it's safe to assume we have the days of the week correct.

Part 2: Experimental Design

1. What would you choose as the key measure of success of this experiment and why?

I would look at the percentage of rides in each city per driver as the key metric. Drivers exclusive to each city will have a percentage close to 100% in their main city. After introducing the treatment, seeing the change in percent for drivers in each city would be a strong indicator of whether or not the treatment made a difference to each driver and to each distinct population. This also has the advantage of measuring the cross-city tendencies of drivers; using amount of available drivers in each city, for example, might just reflect an increased interest in driving in general during the treatment rather than an increased willingness to cross cities. It'd be important to track a driver's "home city" to allow for random assignment to groups within each city.

2. Describe a practical experiment including implementation details, statistical tests, and result interpretation.

Fortunately, the Uber platform provides an easy means of experimentation. Four groups makes the most sense: a control group and a treatment group for each city (Gotham Control, Gotham Treatment, Metropolis Control, Metropolis Treatment). A power calculation would give us a starting point for selecting a sample size. To perform this calculation we'd need some data up front to estimate the standard deviation of percentages pre-treatment. Given that, if we decided a 10% change in percentage is "significant enough," we could calculate our sample size. For now, I'll just say each group has 100 members. Additionally, it'd be best to get the groups as similar as possible in terms of demography, car value, typical tendency distributions, etc., in order to isolate the treatment variable more readily. It's good enough to choose a genuinely random group from each city, and within each city randomly assign the drivers to control and treatment groups.

The Uber platform allows for precise delivery of the treatment with extremely limited potential for spillover. The treatment driver groups in each city would be notified via the app that for the next two weeks days all tolls of the intercity bridge will be waived. Two weeks will account for some variance that's possible week to week, so that's a reasonable duration to detect weekly trends and defeat unusual fluctuations. The control group would be told nothing, and proceed as normal. Ethically speaking, it'd be defensible to inform both groups that they were part of an experiment and add nothing further for the control group. After a week, inform all groups that the experiment has concluded.

Once the data has been gathered, a simple one-sided t-test between the two groups in each city will determine whether the treatment induces a positive effect. The data would also allow weekend/weekday, daytime/nighttime comparisons in a similar fashion. The t-test assumes normality and constant variance; normality can be tested using the Shapiro-Wilk test and constant variance can be assessed with an F test or by reviewing a QQ plot. If either of those assumption are violated, a Mann-Whitney U test can also assess the effect significance without those assumptions.

Results would be easily interpreted with some room for further exploration. Completely aggregated, the result is simple and easy to interpret ("The treatment did not increase cross

city drivership"), but digging deeper might reveal hidden trends ("The treatment increased cross-city drivership by 18% during morning and afternoon rush hour, but not during other times or during the weekend"). Recommended changes to operations would reflect that deeper analysis ("Reimburse toll fees during peak hours when the surge pricing is higher than 1.5").

Part 3: Predictive Modeling

The task: To build a predictive model of whether a rider will be "active" or not given a dataset of riders.

We'll be switching between R and Python for this question, starting with R. The steps we'll take will be to clean and adapt our data to fit the analysis, decide on model priorities, build and compare models, and choose the best model.

We establish our environment and load the data. An easy issue with the data is that null values or values resulting from math errors (like dividing by zero) were listed as NaN without quotes. This is an issue, as R expects NULL or NA values. I used the search and replace functionality of a text editor to switch the NaN values to "NA" so R would load the data with no problems.

```
library(jsonlite)
library(ggplot2)
library(reshape2)
library(class)

raw_data = fromJSON(txt = "uber_data_challenge v2.json")
#format the dates as dates
raw_data$signup_date = as.POSIXct(raw_data$signup_date)
raw_data$last_trip_date = as.POSIXct(raw_data$last_trip_date)
```

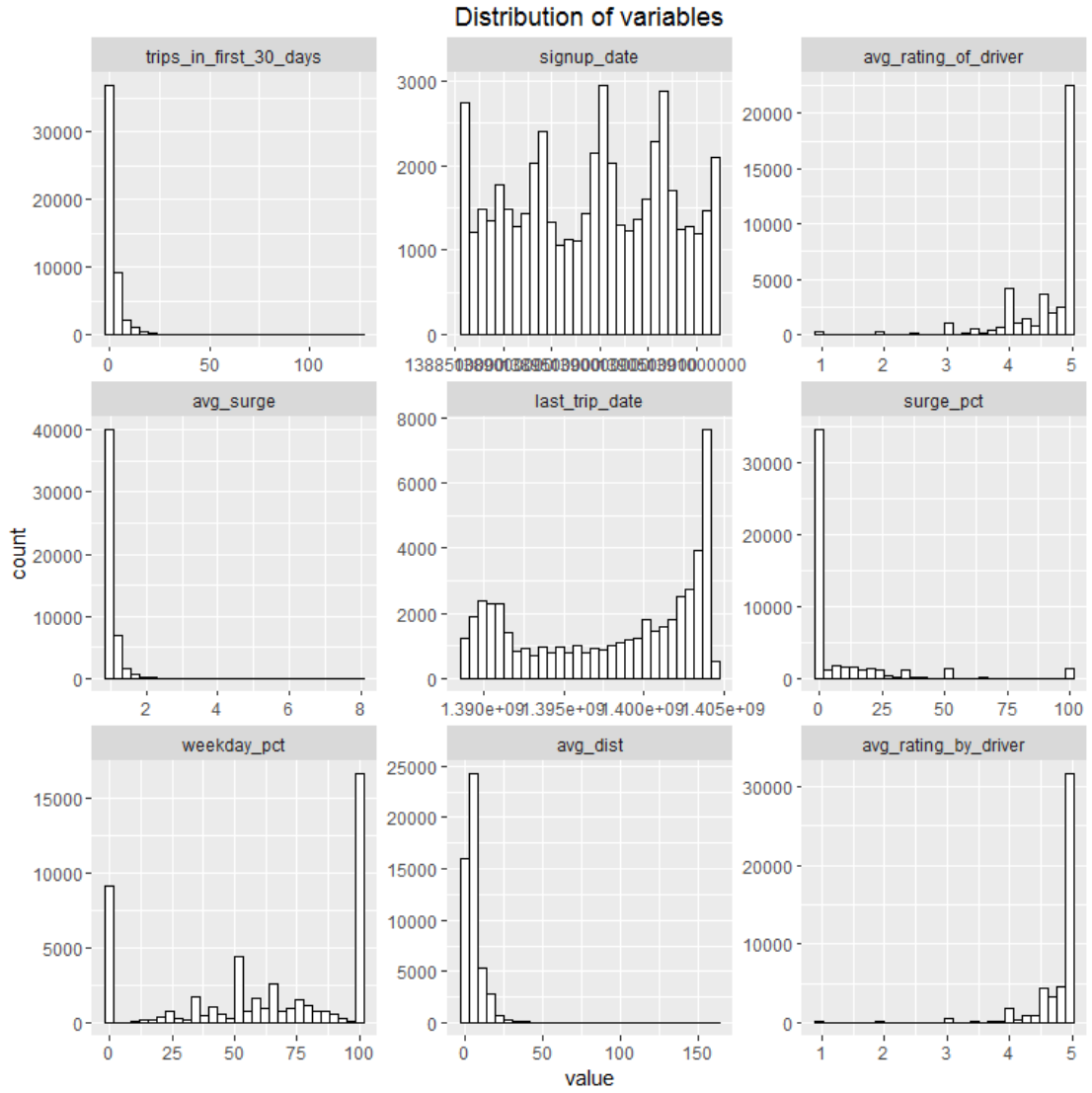
Next we establish some useful variables. Since the prompt defines "active" as having ridden in the last 30 days, we need to know when the data was pulled. We'll assume the most recent ride was when the data was current and derive activeness from that.

```
current_date = max(raw_data$last_trip_date)
record_count = nrow(raw_data)

#establish which users are "active"
raw_data$active = difftime(current_date, raw_data$last_trip_date, units = "da
ys") < 30
```

Now that our data is ready to go, we can have a look at it. We'll check some basic structures, numbers, and distributions.

```
str(raw_data)

## 'data.frame':    50000 obs. of  13 variables:
##  $ city                  : chr  "King's Landing" "Astapor" "Astapor" "King
```

```
's Landing" ...
##  $ trips_in_first_30_days: int  4 0 3 9 14 2 1 2 2 1 ...
##  $ signup_date           : POSIXct, format: "2014-01-25" "2014-01-29" ...
##  $ avg_rating_of_driver  : num  4.7 5 4.3 4.6 4.4 3.5 NA 5 4.5 NA ...
##  $ avg_surge             : num  1.1 1 1 1.14 1.19 1 1 1 1 1 ...
##  $ last_trip_date        : POSIXct, format: "2014-06-17" "2014-05-05" ...
##  $ phone                 : chr  "iPhone" "Android" "iPhone" "iPhone" ...
##  $ surge_pct             : num  15.4 0 0 20 11.8 0 0 0 0 0 ...
##  $ uber_black_user       : logi  TRUE FALSE FALSE TRUE FALSE TRUE ...
##  $ weekday_pct           : num  46.2 50 100 80 82.4 100 100 100 100 0 ...
##  $ avg_dist              : num  3.67 8.26 0.77 2.36 3.13 ...
##  $ avg_rating_by_driver  : num  5 5 5 4.9 4.9 5 4 5 5 5 ...
##  $ active                : logi  TRUE FALSE FALSE TRUE FALSE TRUE ...

colSums(is.na(raw_data))

##                   city trips_in_first_30_days            signup_date
##                      0                      0                      0
##   avg_rating_of_driver             avg_surge         last_trip_date
##                   8122                      0                      0
##                  phone             surge_pct        uber_black_user
##                    396                      0                      0
##            weekday_pct              avg_dist   avg_rating_by_driver
##                      0                      0                    201
##                 active
##                      0

ggplot(melt(raw_data), aes(value)) + geom_histogram(color = "black", fill = "
white") + facet_wrap(~variable, scales = "free") + labs(title = "Distribution
of variables")
```



We're actually missing quite a lot of data, from one column in particular: average rating of driver. It'd best to avoid using this in our models, so let's just drop that data.

```
raw_data = raw_data[-4]
```

Next we must establish a means of comparing models. There are several ways to do this; accuracy is a simple and intuitive way to measure. Accuracy is simply the percentage of correct classifications. This can be troublesome when building models to detect rare occurrences, since a model can be extremely accurate by predicting that these rare events never happen, obviously defeating the point of the model. Depending on whether it's worse to predict an event when there is none, or to predict no event when there is one, other measures can be used to derive maximum usefulness of the model. For our purposes, though, we'll stick with simple accuracy.

Let's look at the percentage of active riders. This will give us our prior belief that a rider will be retained and give us a lower bound for our model.

```
sum(raw_data$active)/length(raw_data$active)
```

```
## [1] 0.3662
```

So we're only retaining about 37% of riders. If the model guessed "non-retained" for every datapoint regardless of the data, it'd perform at, on average, 63% accuracy, but wouldn't be very useful, which illustrates the limitations of accuracy as a metric. . This is our effective lower bound, so we'll want models that perform better than 63%. Next we'll build a simple non-trivial model for our "practical" lower bound.

Data Preparation and Baseline Model: Logistic Regression

We'll pick a few reasonable features and train a logistic regression model on the data and see how it performs. Logistic regression is a good starting point in classification efforts because it's a very well-known and well-understood technique; many scientists understand the math driving it and its properties. The same cannot be said for, say, support vector machines or neural networks.

We'll shuffle the data, split it into training and test sets, and develop a baseline model by assessing accuracy and deriving a confusion matrix.

```
#shuffle the data
shuffled_index = sample(seq(1, nrow(raw_data)), replace = FALSE)
shuffled = raw_data[shuffled_index,]

#split the data into training and test sets for validation
training_data = shuffled[seq(1, .75*record_count),]
test_data = shuffled[seq(.75*record_count+1, record_count),]
nrow(training_data) + nrow(test_data)
```

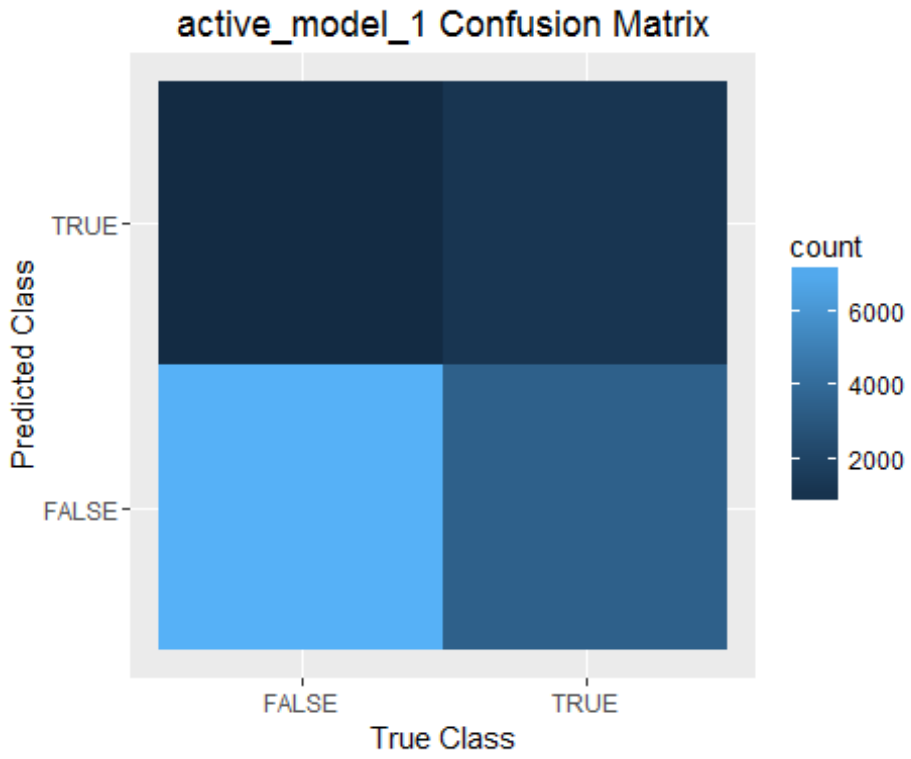
```
## [1] 50000
```

```
#train a model on the training set
active_model_1 = glm(data = training_data, formula = active ~ trips_in_first_
30_days + weekday_pct + avg_dist + avg_rating_by_driver + uber_black_user)

#make predictions against the test set
preds = predict.glm(active_model_1, newdata = test_data) > .5
matches = (predict.glm(active_model_1, newdata = test_data) > .5) == test_dat
a$active
sum(matches, na.rm = TRUE)/length(matches)
```

```
## [1] 0.67816
```

```
#derive the confusion matrix
confusion_matrix_combinations = data.frame(true_class = test_data$active, pre
d_class = preds)
confusion_matrix = table(confusion_matrix_combinations$true_class, confusion_
```

```
matrix_combinations$pred_class)
confusion_matrix

##
##          FALSE TRUE
##    FALSE  7352  546
##    TRUE   3429 1125

ggplot(confusion_matrix_combinations) + geom_bin2d(aes(true_class, pred_class
)) + labs(title="active_model_1 Confusion Matrix", x="True Class", y="Predict
ed Class")

## Warning: Removed 48 rows containing non-finite values (stat_bin2d).
```



Our logistic regression model performs at around 67% accuracy. This is better than our lower bound; we'll compare future models against this baseline. Presumably, we want to find riders who would not be retained so we can take steps to retain them. In this case it's more important that the model doesn't miss rider who would not be retained. It's possible to "tune" the model to a more lenient standard in order to catch more of the non-retained riders, but this comes at a cost. Let's look at the tendencies of the model.

```
confusion_matrix

##
##          FALSE TRUE
##    FALSE  7352  546
##    TRUE   3429 1125
```

```
#Percentage of non-retained rider correctly identified
100*confusion_matrix[1,1]/(confusion_matrix[1,1] + confusion_matrix[1,2])
```

```
## [1] 93.08686
```

```
#Percentage of riders identified as non-retained who are actually not retaine
d
100*confusion_matrix[1,1]/(confusion_matrix[1,1] + confusion_matrix[2,1])
```

```
## [1] 68.19405
```

The model is quite good at recognizing non-retained riders, only missing about 7% of them, but it also has a tendency to identify retained riders as non-retained.

This is a well-known statistical phenomena that makes comparing models difficult, since the same model can perform differently depending on how it's tuned and what's more important. Fortunately, we can use a diagram called a receiver operating characteristic (ROC) curve to judge how good a model is across a range of tunings.

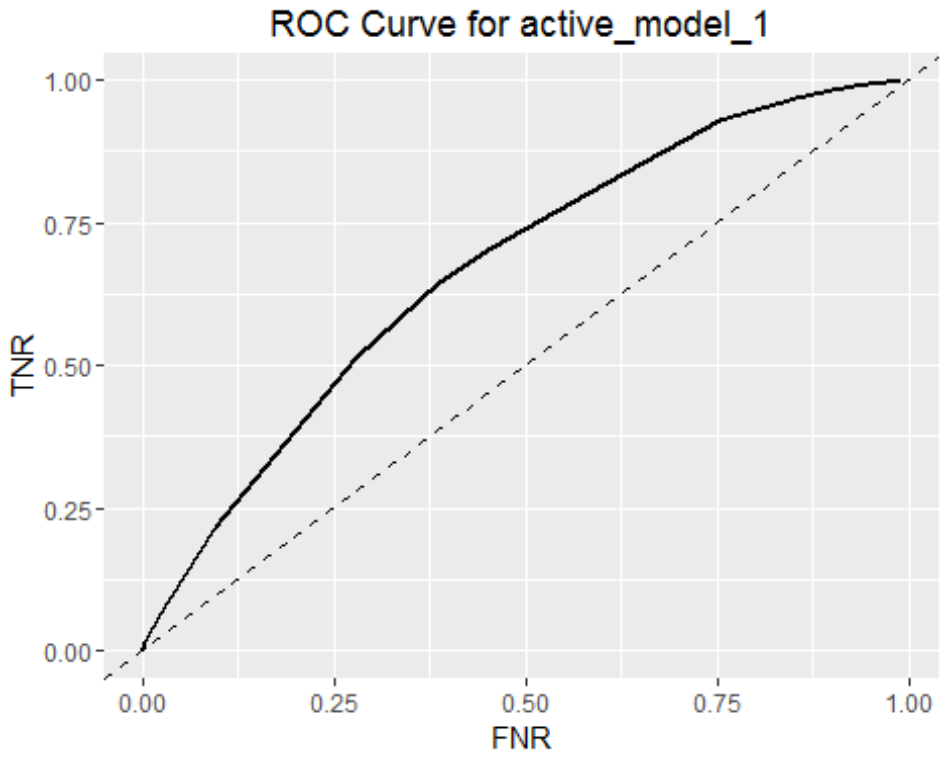
```
thresholds = seq(0, 1, .05)

results = data.frame(FNR = c(), TNR = c(), TH = c())
start = proc.time()
for (threshold in thresholds){
  preds = predict.glm(active_model_1, newdata = test_data) > threshold
  matches = (predict.glm(active_model_1, newdata = test_data) > threshold) ==
test_data$active

  confusion_matrix_combinations = data.frame(true_class = test_data$active, p
red_class = preds)
  confusion_matrix = table(confusion_matrix_combinations$true_class, confusio
n_matrix_combinations$pred_class)

  TNR = confusion_matrix[1,1]/sum(confusion_matrix[1,])
  if (dim(confusion_matrix)[2] == 1){
    FNR = 1
  } else {
    FNR = 1 - (confusion_matrix[2,2]/sum(confusion_matrix[2,]))
  }
  result = data.frame(FNR = c(FNR), TNR = c(TNR), TH = c(threshold))
  results = rbind(results, result)
}
print("Calculation duration:")
```

```
## [1] "Calculation duration:"
```

```
proc.time() - start
```

```
##    user  system elapsed
##    0.63    0.00    0.62
```

```
ggplot(results) + geom_line(aes(FNR, TNR), size = 1) + geom_abline(slope = 1,
intercept = 0, linetype="dashed") + labs(title = "ROC Curve for active_model_
1")
```



The dashed line represents a random chance classifier. We'd like to see this curve peak sharply and early; the farther the curve is away from the random chance line, the better. We can also see where the optimal sensitivity is based on our needs from the shape of the curve.

Additional Model: k Nearest Neighbors

Another model we can use is a simple memory-based model called K nearest neighbors (knn). We can generate a similar curve for it, but knn isn't a probabilistic model, so we won't see a nice curve like with the logistic model. We can still compare the ranges across which the model classifies based on k, the main parameter.

```
solid_train_data = na.omit(training_data)
solid_test_data = na.omit(test_data)
train_cl = factor(solid_train_data$active)
test_cl = factor(solid_test_data$active)
solid_train_data = solid_train_data[c('trips_in_first_30_days','avg_surge','s
urge_pct','weekday_pct','avg_dist')]
solid_test_data = solid_test_data[c('trips_in_first_30_days','avg_surge','sur
ge_pct','weekday_pct','avg_dist')]
```

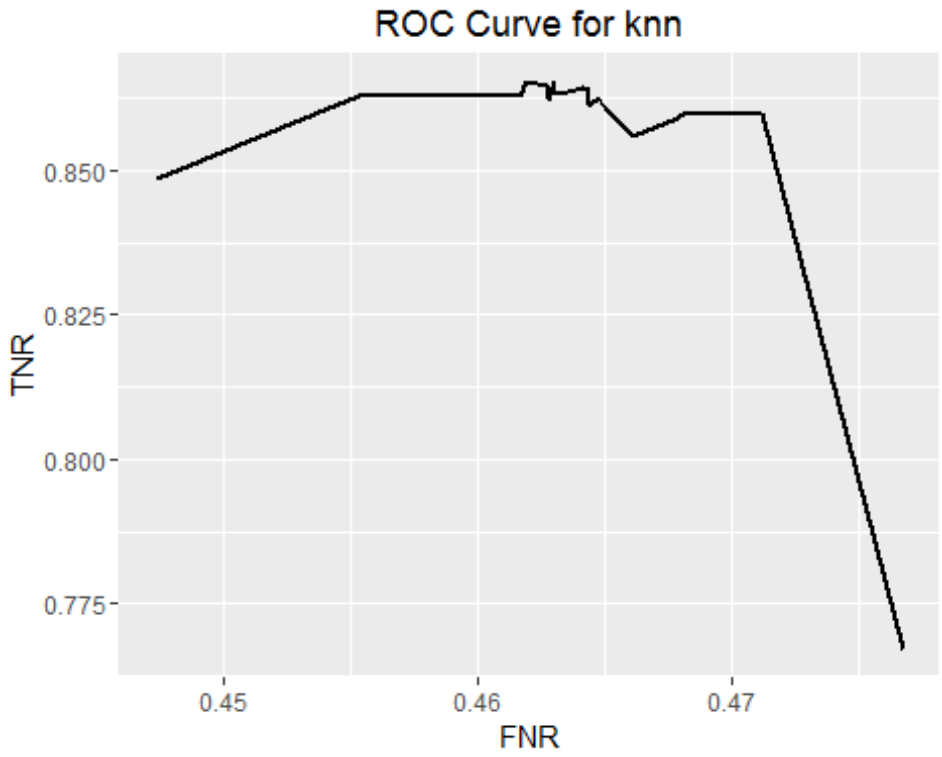
```r
thresholds = seq(1, 200,10)
results_knn = data.frame(FNR = c(), TNR = c(), TH = c())
start = proc.time()
for (threshold in thresholds){
  preds = knn(solid_train_data, solid_test_data, cl = train_cl, k=threshold)

  confusion_matrix_combinations = data.frame(true_class = test_cl, pred_class
= factor(preds))
  confusion_matrix = table(confusion_matrix_combinations$true_class,  confusi
on_matrix_combinations$pred_class)

  TNR = confusion_matrix[1,1]/sum(confusion_matrix[1,])
  FNR = 1 - (confusion_matrix[2,2]/sum(confusion_matrix[2,]))

  result = data.frame(FNR = c(FNR), TNR = c(TNR), TH = c(threshold))
  results_knn = rbind(results_knn, result)
}
print("Calculation duration:")
```

```
## [1] "Calculation duration:"
```

```r
proc.time() - start
```

```
##    user  system elapsed
##  107.53    0.00  107.59
```

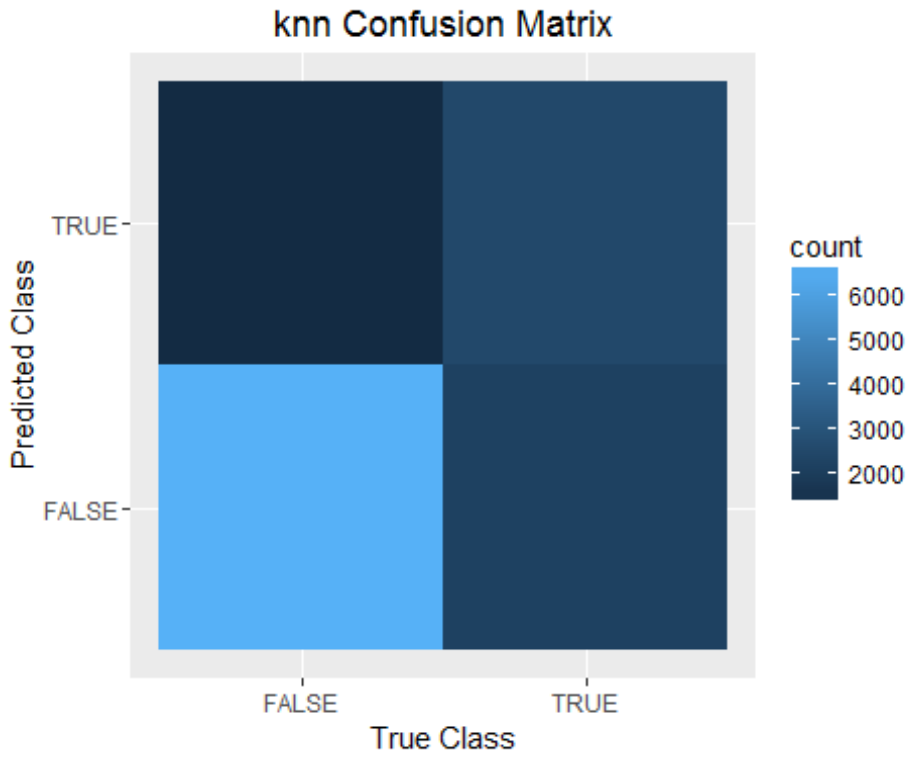
```r
ggplot(results_knn) + geom_line(aes(FNR, TNR), size = 1) + geom_abline(slope
= 1, intercept = 0, linetype="dashed") + labs(title = "ROC Curve for knn")
```



Some things jump out from the code. First, knn requires complete entries; any entry in the training or test set with a missing datapoint must either be filled with dummy data or thrown out. In this case, we choose to throw them out. Second, the nature of the model is that you don't really "train" it; there's no model being built. The entire dataset must be fit into memory and datapoints to classify must be compared to every single point in the training set. This is not a naively scalable model to use, nor a particularly fast one (this model took over a minute to assess whereas the logistic model took less than a second). The curve is hideous, but it does imply a strong ability to classify at certain points. When k is small, it's quite accurate, around 80%. To be clear, adjusting k is not comparable to adjusting the sensitivity of our logistic model, but it's the only significant way to tune a knn classifier. Let's look at the confusion matrix at one of the better settings.

```
  preds = knn(solid_train_data, solid_test_data, cl = train_cl, k=75)

  confusion_matrix_combinations = data.frame(true_class = test_cl, pred_class
= factor(preds))
  confusion_matrix = table(confusion_matrix_combinations$true_class,  confusi
on_matrix_combinations$pred_class)

confusion_matrix

##
##          FALSE TRUE
##   FALSE  6770 1064
##   TRUE   2106 2423
```

```
#Percentage of non-retained rider correctly identified
100*confusion_matrix[1,1]/(confusion_matrix[1,1] + confusion_matrix[1,2])

## [1] 86.41818

#Percentage of riders identified as non-retained who are actually not retaine
d
100*confusion_matrix[1,1]/(confusion_matrix[1,1] + confusion_matrix[2,1])

## [1] 76.2731

ggplot(confusion_matrix_combinations) + geom_bin2d(aes(true_class, pred_class
)) + labs(title="knn Confusion Matrix", x="True Class", y="Predicted Class")
```



So the knn model catches about 86% of non-retained riders (worse than the logistic model), but 76% of the riders classified as non-retained actually were (better than the logistic model).

It's clear that we must prioritize our expectations of our classifier. Which is more important? If we had a prefect classifier, what would we do with it? We've implicitly assumed that we're actually looking to find non-retained riders, but is it better to overestimate how many non-retains there are or is it better to underestimate?

We'll try a few more models and see what we can do.

Additional Model: Decision Trees

We'll try decision trees next. For this, we'll switch from R to Python because the Python machine learning libraries are much more concise and consistent. Also, those libraries will save a lot of the manual coding we've done so far (again, there are R libraries for this but it's good to be able to write the basics manually for clarity).

```
%matplotlib inline
import matplotlib.pyplot as plt

import sklearn
import pandas as pd
import numpy as np
import json
from pprint import pprint
import time

from sklearn.decomposition import  PCA
from sklearn.svm import SVC
from sklearn.cross_validation import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.feature_extraction import DictVectorizer

raw_data = pd.read_json("./uber_data_challenge v2.json")

raw_data.dtypes

avg_dist                  float64
avg_rating_by_driver       object
avg_rating_of_driver       object
avg_surge                 float64
city                       object
last_trip_date             object
phone                      object
signup_date                object
surge_pct                 float64
trips_in_first_30_days      int64
uber_black_user              bool
weekday_pct               float64
dtype: object

raw_data['last_trip_date'] = pd.to_datetime(raw_data['last_trip_date'])
raw_data['signup_date'] = pd.to_datetime(raw_data['signup_date'])

current_date = max(raw_data['last_trip_date'])
```

```
diffs = current_date - raw_data['last_trip_date']

raw_data['days_since_last_trip'] = [x.days for x in diffs]

labels = [x < 30 for x in raw_data['days_since_last_trip']]
labels[:5]

[True, False, False, True, False]
```

A lot of the advanced classification methods have trouble with, or require preprocessing to handle, ordinal and categorical data. Before we get that far, let's look at how the categorical variables, City and Phone, reflect rider retention.

```
for_grouping = raw_data
for_grouping['active'] = labels
grouped = raw_data.groupby(['city','phone'])
by_city_and_phone = grouped['active'].mean()
by_city_and_phone

city            phone
Astapor         Android    0.106026
                iPhone     0.307727
King's Landing  Android    0.427942
                iPhone     0.687104
Winterfell      Android    0.180907
                iPhone     0.416598
Name: active, dtype: float64
```

Normalize the results...

```
by_city_and_phone/min(by_city_and_phone)

city            phone
Astapor         Android    1.000000
                iPhone     2.902372
King's Landing  Android    4.036205
                iPhone     6.480524
Winterfell      Android    1.706249
                iPhone     3.929211
Name: active, dtype: float64
```

This is a Stark (har!) distribution. One might expect a generally similar average retention across cities and phones but this doesn't appear to be the case. iPhone users are across the board *much* more likely to retain than Android users, and user in King's Landing are *much* more likely to retain than riders in other cities. This makes sense, since King's Landing is a densely populated city more suited to taxi-type service than remote Winterfell (much smaller than King's Landing, although the show makes King's Landing look much smaller than it is in the books) or Astapor (where slave labor would undercut any commercial offerings). The books provide little indication of the popularity of different brands of smartphones, unfortunately. That's getting into domain-specific knowledge, though.

Let's take a quick look at the counts of each subsection to make sure our classes aren't wildly imbalanced or that our sample sizes are too low:

```
print("Class counts for each subcategory:")
grouped.count()['active']
```

```
Class counts for each subcategory:

city            phone
Astapor         Android     5244
                iPhone     11169
King's Landing  Android     2498
                iPhone      7568
Winterfell      Android     7280
                iPhone     15845
Name: active, dtype: int64
```

Normalize the results...

```
print("Normalized class counts for each subcategory:")
grouped.count()['active']/min(grouped.count()['active'])
```

```
Normalized class counts for each subcategory:

city            phone
Astapor         Android    2.099279
                iPhone     4.471177
King's Landing  Android    1.000000
                iPhone     3.029624
Winterfell      Android    2.914331
                iPhone     6.343074
Name: active, dtype: float64
```

As an aside, we can multiply these two to figure out a sort of relative Bayesian ROI for riders in each subsection. A bit tangential, but a neat thing to check out.

```
print("Relative Bayesian effective riders per subcategory:")
B_ROI = (by_city_and_phone/min(by_city_and_phone)) * (grouped.count()['active
']/min(grouped.count()['active']))
B_ROI/min(B_ROI)
```

```
Relative Bayesian effective riders per subcategory:

city            phone
Astapor         Android     1.000000
                iPhone      6.181655
King's Landing  Android     1.922662
                iPhone      9.352518
Winterfell      Android     2.368705
                iPhone     11.872302
Name: active, dtype: float64
```

Looks like there are 12x more effective retained riders in Winterfell on iPhones than in Astapor on Androids. This apparently high importance of a few categorical variables implies a decision tree or a well-formulated logistic regression would work well. We've already tried logistic regression, but it might work better if we emphasize the newfound importance of the city and phone variables. First, let's try our next technique: decision trees.

Tangentially, the classes aren't terribly balanced so we might have to handle that later, but the sample sizes are fine.

First, we'll have to modify our data. We'll use a technique called one-hot encoding to enable generalized classifiers to learn on our data. There are automated ways to do this but I can never get them to work, and since there are only a few options, I'll manually convert the variables.

```
no_dates = raw_data[['avg_dist','avg_surge','surge_pct','trips_in_first_30_da
ys','weekday_pct']]
no_dates['iphone_enc'] = [x == "iPhone" for x in raw_data['phone']]
no_dates['android_enc'] = [x == "Android" for x in raw_data['phone']]
no_dates['astapor_enc'] = [x == "Astapor" for x in raw_data['city']]
no_dates['kingslanding_enc'] = [x == "King's Landing" for x in raw_data['city
']]
no_dates['winterfell_enc'] = [x == "Winterfell" for x in raw_data['city']]
```

Next we'll split our dataset. We'll remove the dates to make the decision tree work.

```
data_train, data_test, labels_train, labels_test = train_test_split(no_dates,
labels, test_size=.25, random_state = 2008)
```

A basic decision tree will be our starting point. From there, we'll train a random forest and see if we get a better decision tree.

```
clf_dt1 = DecisionTreeClassifier().fit(data_train, labels_train)
clf_dt1.score(data_test, labels_test)

0.70104

clf_rf1 = RandomForestClassifier(n_estimators=25).fit(data_train, labels_trai
n)
clf_rf1.score(data_test, labels_test)

0.72631999999999997
```

We get a pretty good model out of the random forest. This is quite common; decision trees have the common weakness of requiring manual feature engineering while random forests effective learn their features stochastically. Let's check the confusion matrix of our random forest model.

```
confusion_matrix(labels_test, clf_rf1.predict(data_test))

array([[6204, 1708],
       [1713, 2875]])
```

This is actually quite good. It's good at catching riders who would not be retained and doesn't overguess too much. Let's try our last technique: support vector machines.

Additional Model: Support Vector Machine

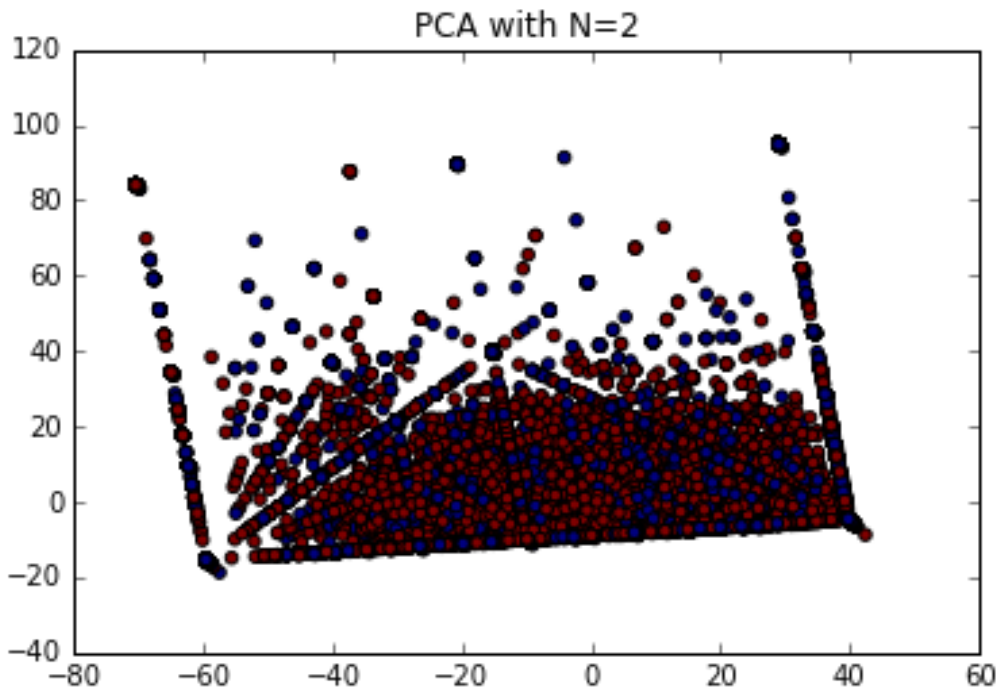
In principle, support vector machines (SVMs) work a bit like logistic regression insofar as it tries to discover the most effective decision boundary between classes in whatever dimensionality the data exists. We'll just use numerical variables for now, although one-hot encoding would allow for categorical variables as before. We start by subsetting the data down to just continuous variables.

```
data_train, data_test, labels_train, labels_test = train_test_split(raw_data,
labels, test_size=.25, random_state = 2008)

train_grouped = data_train.groupby(['city','phone'])

just_numbers = raw_data[['avg_dist','avg_surge','surge_pct','trips_in_first_3
0_days','weekday_pct']]
print(just_numbers.head())
```

```
   avg_dist  avg_surge  surge_pct  trips_in_first_30_days  weekday_pct
0      3.67       1.10       15.4                       4         46.2
1      8.26       1.00        0.0                       0         50.0
2      0.77       1.00        0.0                       3        100.0
3      2.36       1.14       20.0                       9         80.0
4      3.13       1.19       11.8                      14         82.4
```

Let's start by projecting our data into fewer dimensions to get a read on how easily a decision boundary of just the numerical variables might work. We'll use principal component analysis (PCA) to project our data into two dimension for easy visualization.

```
pca = PCA(n_components=2)
reduced = pca.fit_transform(just_numbers)
pca.explained_variance_ratio_

array([ 0.76197171,  0.21259258])

plt.scatter(reduced[:,0], reduced[:,1], c=labels)
plt.title("PCA with N=2")
plt.show()
```



That doesn't bode well. Doesn't seem likely we'll be able to build a highly accurate classifier, but this can be deceiving; we're throwing away some information projecting the data into two dimensions, though the explained variance calculation above implies we're capturing the vast majority of the variance. There are also some weird patterns here that imply some strange correlations and, to me, look artificially generated.

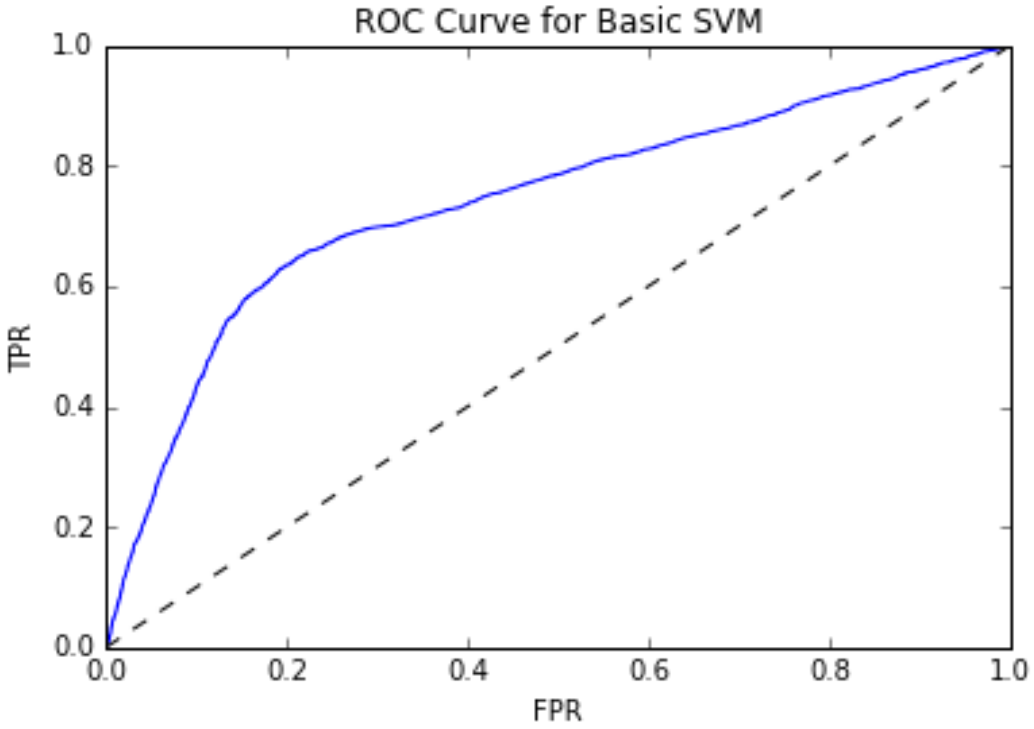
We'll again split our data and train a classifier, checking the accuracy and confusion matrix.

```
data_train, data_test, labels_train, labels_test = train_test_split(just_numb
ers, labels, test_size=.25, random_state = 2008)

clf_svm = SVC().fit(data_train, labels_train)
print(clf_svm.score(data_test, labels_test))

0.74616

confusion_matrix(labels_test, clf_svm.predict(data_test))

array([[6797, 1115],
       [2058, 2530]])
```

The SVM model is better at catching non-retained riders than the decision tree, but is less accurate with retained riders. Let's take a look at its ROC curve.

```
y_score = clf_svm.decision_function(data_test)
fpr, tpr, thresholds = roc_curve(labels_test, y_score)
plt.plot(fpr, tpr)
plt.plot([0,1], [0,1], 'k--')
```

```
plt.title("ROC Curve for Basic SVM")
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.show()
```



This looks better than our baseline logistic classifier. It's possible to compute the area under the curves (AUC) to reduce the comparison to a single number, but the shape of the curve is more important than the AUC; the point is to express the tradeoff between sensitivity and accuracy which can't be expressed in a single number. Since it's rather misleading, I'll leave it out. For example, we couldn't calculate the AUC for KNN, but we can see that KNN gives us the best tradeoff so far (time and memory constraints notwithstanding).

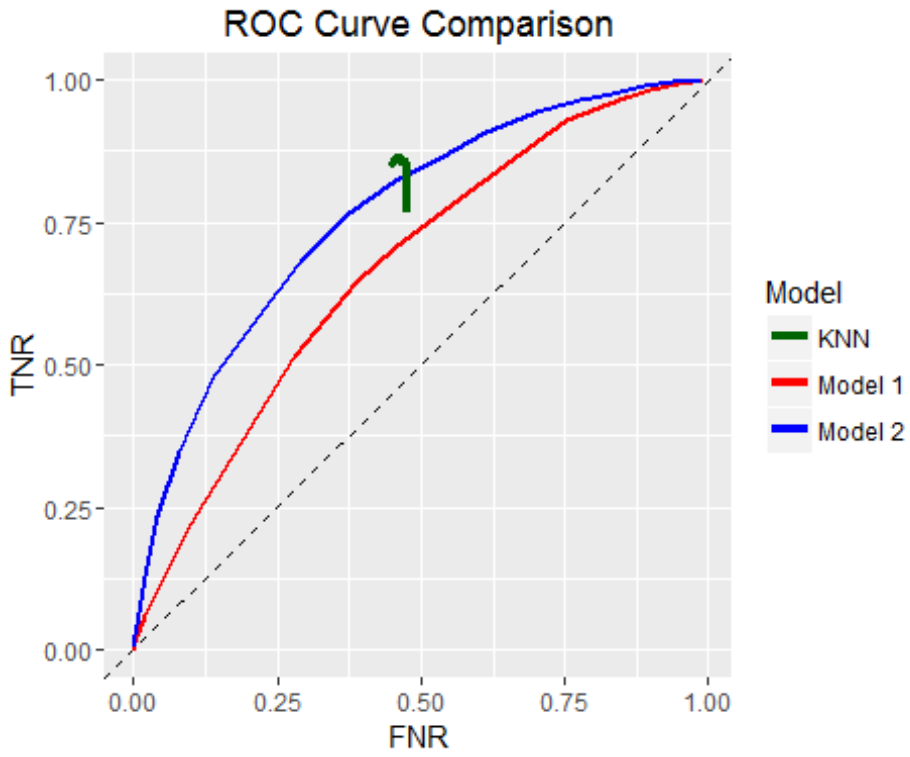
It looks like the SVM model is performing best. Before we call it, let's give logistic regression another change. We've found that city and phone are strong predictors of retention; let's isolate the regression results by city and phone and see if it performs well.

Additional Model: Logistic Regress Revisited

We saw from poking around in Python that the user's phone and city were strong indicators of retention. Let's go back to R and work with logistic regression to factor that in.

```
active_model_2 = glm(data = training_data, formula = (active ~ phone*city*(we
ekday_pct + trips_in_first_30_days + avg_surge + surge_pct + uber_black_user
+ avg_dist)))

thresholds = seq(0, 1, .05)
```

```
results2 = data.frame(FNR = c(), TNR = c(), TH = c())
start = proc.time()
for (threshold in thresholds){
  preds = predict.glm(active_model_2, newdata = test_data) > threshold
  matches = (predict.glm(active_model_2, newdata = test_data) > threshold) ==
test_data$active

  confusion_matrix_combinations = data.frame(true_class = test_data$active, p
red_class = preds)
  confusion_matrix = table(confusion_matrix_combinations$true_class, confusio
n_matrix_combinations$pred_class)

    TNR = confusion_matrix[1,1]/sum(confusion_matrix[1,])
  if (dim(confusion_matrix)[2] == 1){
    FNR = 1
  } else {
    FNR = 1 - (confusion_matrix[2,2]/sum(confusion_matrix[2,]))
  }


  result = data.frame(FNR = c(FNR), TNR = c(TNR), TH = c(threshold))
  results2 = rbind(results2, result)
}
print("Calculation duration:")

## [1] "Calculation duration:"

proc.time() - start

##    user  system elapsed
##    1.11    0.08    1.19

cols = c("Model 1"="red","Model 2"="blue", "KNN"="darkgreen")

ggplot() + geom_abline(slope = 1, intercept = 0, linetype="dashed") +
  geom_line(data = results, aes(FNR, TNR, color="Model 1"), size=1) +
  geom_line(data = results2, aes(FNR, TNR, color="Model 2"), size = 1) +
  geom_line(data = results_knn, aes(FNR, TNR, color="KNN"), size = 1.5) +
  labs(title = "ROC Curve Comparison") +
  scale_color_manual(name = "Model", values = cols)
```



Model 2, our updated logistic model, looks pretty good. Let's check the confusion matrix:

```
preds = predict.glm(active_model_2, newdata = test_data) > .5
matches = (predict.glm(active_model_2, newdata = test_data) > .5) == test_dat
a$active

confusion_matrix_combinations = data.frame(true_class = test_data$active, pre
d_class = preds)
confusion_matrix = table(confusion_matrix_combinations$true_class, confusion_
matrix_combinations$pred_class)

confusion_matrix
```

```
##
##          FALSE TRUE
##   FALSE  6816 1052
##   TRUE   2442 2099
```

So the model is about 50/50 on the riders who are actually retained, but correctly finds about 85% of the riders who are not. I would assume that the goal is to find riders who would not retain and act on those riders to reduce churn, so this is a good balance. KNN actually beats both models in a certain range, but it's not that much better and takes much more time and memory. Despite all the fancy algorithms out there, logistic regression seems to work fine. Our best model, though, was a random forest: while the accuracy is lower than the SVM model and KNN, it strikes the best balance between scalability, speed, and catching the best proportion of non-retains to retains.

Functions

```
decompose_timeseries = function(ts, histobj){
  ts.decomp.add = decompose(ts)
  ts.decomp.add.df = data.frame(
    bucket = histobj$Bucket,
    observed = ts.decomp.add$x,
    seasonal = ts.decomp.add$seasonal,
    trend = ts.decomp.add$trend,
    random = ts.decomp.add$random,
    weekend = histobj$Weekend)
  ts.graph.observed = ggplot(ts.decomp.add.df, aes(bucket, observed, color =
weekend, group = 1)) + geom_line(size = 1) + scale_x_datetime() + labs(x="")
  ts.graph.trend = ggplot(ts.decomp.add.df, aes(bucket, trend, color = weeken
d, group = 1)) + geom_line(size = 1) + scale_x_datetime() + labs(x="")
  ts.graph.seasonal = ggplot(ts.decomp.add.df, aes(bucket, seasonal, color =
weekend, group = 1)) + geom_line(size = 1) + scale_x_datetime() + labs(x="")
  ts.graph.random = ggplot(ts.decomp.add.df, aes(bucket, random, color = week
end, group = 1)) + geom_line(size = 1) + scale_x_datetime() + labs(x="Date, T
ime")
  return(grid.arrange(ts.graph.observed, ts.graph.trend, ts.graph.seasonal, t
s.graph.random, ncol=1, top = "Decomposed Additive TS"))
}
```