R/DATA ANALYTICS TASTER SESSION

Dr Andrew Whiter - UCL School of Management

Quick overview of some key basics in R
- Vectors
- Functions
- Loops

Walkthrough of a kaggle competition - to do some exploratory data analysis and then make some predictions
- Data Preparation
- Plotting
- Prediction

About R

- R is a programming language developed by and for statisticians, and has quickly become very popular amongst statisticians and data scientists
- R and R Studio are free and accessible to companies and individuals
- It's both user friendly and powerful, with lots of libraries available to do complex tasks
- There is **lots** of help online, it is very likely someone has asked your question before on stackoverflow.

Calculations

R is able, like most programming languages, to perform basic calculations, e.g:

```
2 + 2

> [1] 4
```

You can perform basic operations in R including:

Addition (+), Subtraction (-), Multiplication(*), Exponents (^), Modulus (%%).

Objects

You can create objects and assign outputs of functions to them, using the assignment operator "<-"

```
d1 <- 2
d1

> [1] 2
```

```
d2 <- 2 + 2
d2

> [1] 4
```

Objects

You can create objects of different types, the three key ones are:

```
obj1 <- 2
obj2 <- TRUE
obj3 <- "hello"
class(obj1)

> [1] "numeric"
```

```
class(obj2)

> [1] "logical"
```

```
class(obj3)

> [1] "character"
```

Vectors

These objects that we just created are actually stored in R as vectors of length one. To create a longer vector we can use the concatenation function, "c":

```
g <- c(2 + 2, 7, 1234, 34, 78000000)
g

> [1]  4  7  1234  34  78000000
```

```
class(g)

> [1] "numeric"
```

We can see the class of this vector is numeric, as all of its elements are numeric.

Vectors

Vectors can contain logical, character and numeric elements, but will store them as all one type, the "highest" that will store all elements

```
h <- c(1 + 1 == 3, "should be false", 2.22, 5)
h

> [1] "FALSE"           "should be false" "2.22"            "5"
```

```
class(h)

> [1] "character"
```

You can play around with the combinations of these to see that the order is: logical < numeric < character.

Vector Subsetting

You can access an element of a vector using the subset "[]" function.

```
i <- c(1,5,8,11,12)
i[1]

> [1] 1
```

```
i[c(1,4)]

> [1]  1  11
```

```
i[2:4]

> [1]  5  8  11
```

Vector Operations

You can perform operations on a vector - either on all elements of the vector, or as an aggregation

```
i <- c(1,5,8,11,12)
j <- c(2,6,9,12,13)
i+j

> [1]  3 11 17 23 25
```

```
i^2

> [1]   1  25  64  121  144
```

Vector Operations

You can perform operations on a vector - either on all elements of the vector, or as an aggregation

```
i <- c(1,5,8,11,12)
mean(i)

> [1] 7.4
```

```
sum(i)

> [1] 37
```

Data Frames

If you're using R for analysing data, you are likely to be making, loading, and manipulating data frames a **lot**. Data frames are objects that include columns (vectors) that can have different types, e.g.:

```
df <- data.frame(colour=c("blue","red","yellow","purple"),
                 number=c(1,2,3,4),
                 iseven=((1:4)%%2 ==0))
df

> colour number iseven
1   blue      1  FALSE
2    red      2   TRUE
3 yellow      3  FALSE
4 purple      4   TRUE
```

Data Frames

You can access an element of a data frame either in the same way that you would a matrix:

```
df[2,3]

> [1] TRUE
```

or using the column names

```
df[3,"number"]

> [1] 3
```

Functions

You can create your own functions in R

```
exponentplusone <- function(a, b){
      c <- a^(b+1)
      return(c)}

exponentplusone(2,3)

> [1] 16
```

This is useful if you want to something multiple times.

Conditionals

The "if" conditional can be used to check >, <, !=, == and other boolean operators (e.g. isnull())

```
x <- 23
if (x%%2 == 0) {print("even")} else {print("odd")}

> [1] "odd"
```

If statements can also be nested:

```
x <- "Does this string contain the word apple?"

if (grepl("apple", x))
  {print("Yes")} else
  if (grepl("orange", x))
    {print("No, but it contains the word orange")} else
    {print("No")}

> [1] "Yes"
```

Loops

Both for loops and while loops are supported in R, with additional "looping" functions over vectors and data frames, (e.g. apply, sapply)

```
for (i in 1:5)
{print(i^2)}

>
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Loops

Both for loops and while loops are supported in R, with additional "looping" functions over vectors and data frames, (e.g. apply, sapply)

```
x <- 0

while(x < 5)
    {x <- x+1
     print(x^2)}

>
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Loops

Both for loops and while loops are supported in R, with additional "looping" functions over vectors and data frames, (e.g. apply, sapply)

```
sapply(1:5, function(x) x^2)

> [1]  1  4  9 16 25
```

Kaggle

Platform that runs data science and machine learning competitions in a crowdsourced way - companies or organisations can post their problems and their data, and data scientists around the world compete to get the highest accuracy on their results.

Kaggle has become quite popular in the data science community, the forums are very highly used and different approaches discussed, kaggle competition wins can be highly prestigious, and recruiters for data science roles may ask whether you have entered any competitions etc.

Problem

The competition we will focus on can be found here: https://www.kaggle.com/c/bike-sharing-demand

This is an old (finished) competition, and is to predict the # of bicycles that will be rented per hour, based on information about the season, month, day, hour and weather.

Accessing Data

Step one is to retrieve the data. We download the training dataset from here: https://www.kaggle.com/c/bike-sharing-demand/data, and make sure that it is in our working directory.

The "train" dataset is what we will use to explore the data and build our models. The "test" dataset does not have the outcome variables, and is used for us to predict the outcome and then submit to kaggle.

Loading Data

To load a csv, R has an aptly named function read.csv:

```
bikes <- read.csv("train.csv")
```

In R studio once this is done you can see in the Environment tab the number of observations and variables. If you double click you can also look into the data and visually sort.

Exploring Data

One of the first things to do when given a dataset is to understand what data is inside, this can be done by looking at the top few rows:

```
head(bikes,4)

>
  datetime            season holiday workingday weather temp atemp
1 2011-01-01 00:00:00      1       0          0       1 9.84 14.395
2 2011-01-01 01:00:00      1       0          0       1 9.02 13.635
3 2011-01-01 02:00:00      1       0          0       1 9.02 13.635
4 2011-01-01 03:00:00      1       0          0       1 9.84 14.395

  humidity windspeed casual registered count
1       81         0      3         13    16
2       80         0      8         32    40
3       80         0      5         27    32
4       75         0      3         10    13
```

Exploring Data

Or by looking at the structure

```
str(bikes)

>
data.frame:    10886 obs. of  12 variables:
 $ datetime  : Factor w/ 10886 levels "2011-01-01 00:00:00",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ season    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ holiday   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ workingday: int  0 0 0 0 0 0 0 0 0 0 ...
 $ weather   : int  1 1 1 1 1 2 1 1 1 1 ...
 $ temp      : num  9.84 9.02 9.02 9.84 9.84 ...
 $ atemp     : num  14.4 13.6 13.6 14.4 14.4 ...
 $ humidity  : int  81 80 80 75 75 80 86 75 76 ...
 $ windspeed : num  0 0 0 0 0 ...
 $ casual    : int  3 8 5 3 0 0 2 1 1 8 ...
 $ registered: int  13 32 27 10 1 1 0 2 7 6 ...
 $ count     : int  16 40 32 13 1 1 2 3 8 14 ...
```

Exploring Data

To drill into a specific variable, we can also look at the summary:

```
summary(bikes$temp)

>   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.82   13.94   20.50   20.23   26.24   41.00
```

The importance of cleaning data

Experts give varying estimates for the % of time that is spent cleaning data, but the basic point is that although it is not the most glamorous of tasks, it's one of the most important ones to master as a data scientist, and will make the final graphs and analysis more meaningful and useful.

Dates? Factors?

We can see that our data variable has been loaded in as a "factor" variable. A "factor" is a categorical variable, where the order and value is not important (an example could be colour).

The order clearly does matter for dates so we will have to cast this into the right format.

```
# YYYY-MM-DD as date class
bikes$date <- as.Date(bikes$datetime)
```

```
# HH as integer
bikes$hour <- hour(ymd_hms(bikes$datetime))
```

How do I know this transformation? (http://stackoverflow.com/questions/19292438/split-date-time)

Check

```
head(data.frame(datetime=bikes$datetime,
           date=bikes$date,
           time=bikes$hour))

>             datetime       date time
1 2011-01-01 00:00:00 2011-01-01    0
2 2011-01-01 01:00:00 2011-01-01    1
3 2011-01-01 02:00:00 2011-01-01    2
4 2011-01-01 03:00:00 2011-01-01    3
5 2011-01-01 04:00:00 2011-01-01    4
6 2011-01-01 05:00:00 2011-01-01    5
```

Cleaning Data - Seasons

Let's have a look at the "season" variable by checking its unique values.

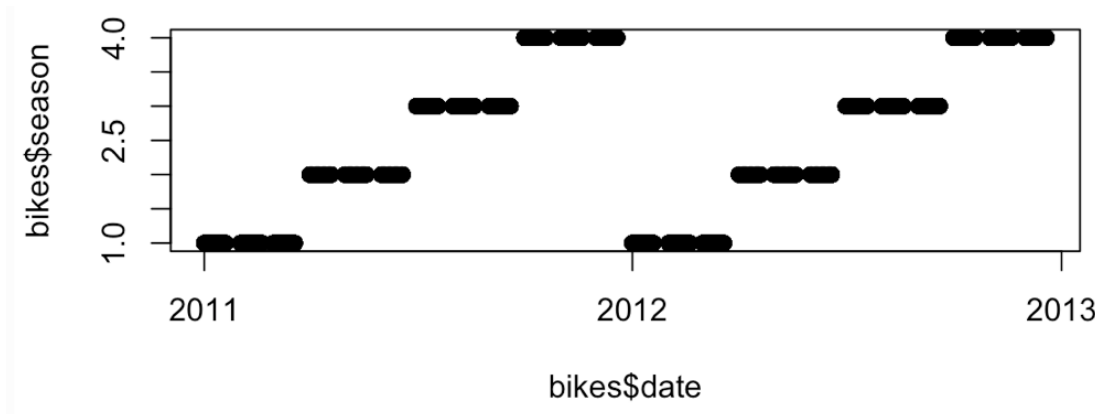
```
unique(bikes$season)

> [1] 1 2 3 4
```

Looks like this is a categorical variable for the four seasons. Let's see if we can figure out which number corresponds to which season by doing a plot.

Cleaning Data - Season

The regular way of plotting in R is plot()

```
plot(bikes$date,bikes$season)
```



There's much to be improved graphically, but this gives us some insight.

Cleaning Data - Season

To double check, let's figure out the possible seasons where the month is January

```
unique(bikes$season[format.Date(bikes$date, "%m")=="01"])

> [1] 1
```

Cleaning Data - Season

If we include season as it is, as an integer 1, 2, 3 or 4, then the model will include it as a numerical variable where winter (4) is worth twice as much as summer (2), etc.

If you're familiar with regression it might be clear why this is a problem. Instead it would be much better if R understood that this data is categorical, so we will create a new variable for season as a factor, and give it its meaningful name.

```r
bikes$seasonfactor <- as.factor(bikes$season)

levels(bikes$seasonfactor) <- c("Spring","Summer",
                                 "Autumn","Winter")
```

Cleaning Data - Weather

The kaggle website for the data also tells us that we have the following options for "weather":

1: Clear, Few clouds, Partly cloudy, Partly cloudy

2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist

3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds

4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog

```r
bikes$weatherfactor <- as.factor(bikes$weather)
```

Cleaning Data

```
> 'data.frame':    10886 obs. of  16 variables:
 $ datetime      : Factor w/ 10886 levels "2011-01-01 00:00:00",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ season        : int  1 1 1 1 1 1 1 1 1 1 ...
 $ holiday       : int  0 0 0 0 0 0 0 0 0 0 ...
 $ workingday    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ weather       : int  1 1 1 1 1 2 1 1 1 1 ...
 $ temp          : num  9.84 9.02 9.02 9.84 9.84 ...
 $ atemp         : num  14.4 13.6 13.6 14.4 14.4 ...
 $ humidity      : int  81 80 80 75 75 75 80 86 75 76 ...
 $ windspeed     : num  0 0 0 0 0 ...
 $ casual        : int  3 8 5 3 0 0 2 1 1 8 ...
 $ registered    : int  13 32 27 10 1 1 0 2 7 6 ...
 $ count         : int  16 40 32 13 1 1 2 3 8 14 ...
 $ date          : Date, format: "2011-01-01" "2011-01-01" ...
 $ hour          : int  0 1 2 3 4 5 6 7 8 9 ...
 $ seasonfactor  : Factor w/ 4 levels "Spring","Summer",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ weatherfactor: Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 2 1 1 1 1 ...
```

Plotting & ggplot

The next step in a data analysis is usually to start looking at some plots. As we saw earlier the base package for plotting in R is not very pretty, it's also not particularly easy to use.

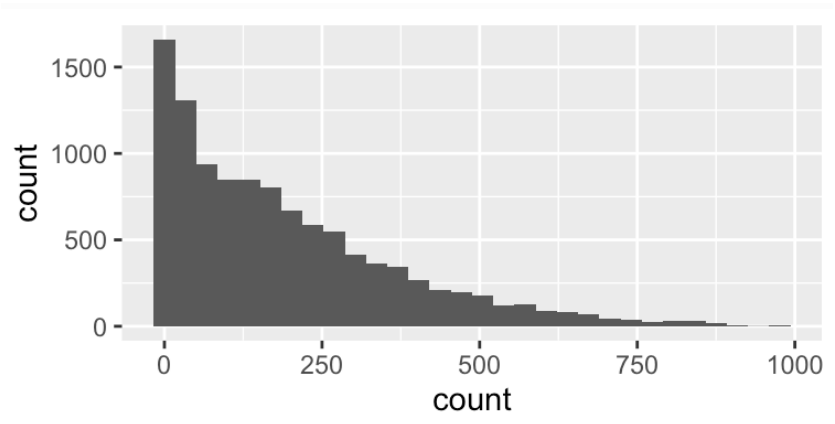
Most R users use the package ggplot, you can get this by installing the package and loading it:

```
install.packages("ggplot2")
library(ggplot2)
```

Exploring Data - Count

We are interested in how many people rent bikes in a certain hour, let's first look at the distribution of # of bikes rented per hour, as a histogram.

```r
ggplot(data=bikes, aes(x=count)) + geom_histogram()
```

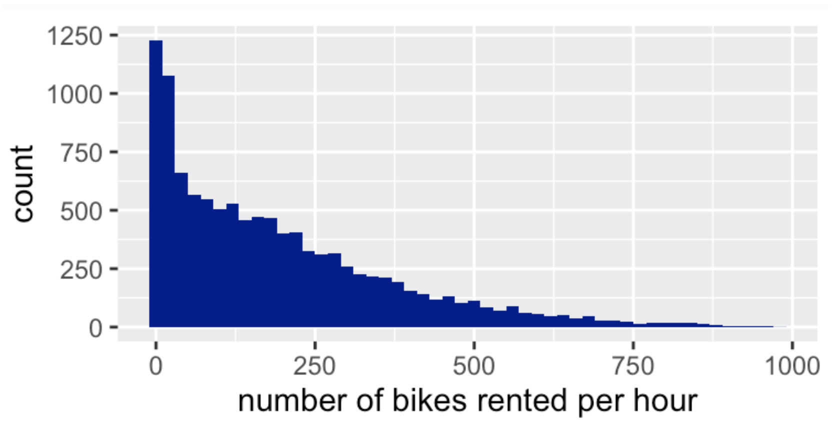


Exploring Data - Count

You will get a warning message while running this saying that the number of "bins" is being chosen automatically by R.

Also, because the histogram default y axis is count, this graph is slightly confusing, let's make it prettier.

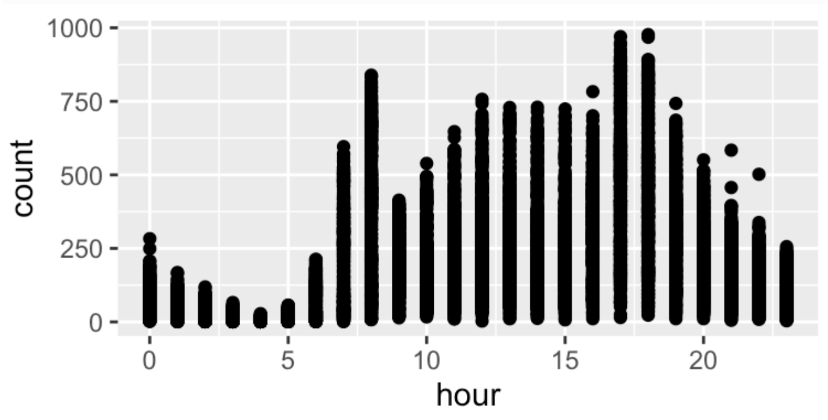
```r
ggplot(data=bikes, aes(x=count)) +
        geom_histogram(binwidth = 20, fill="darkblue") +
        xlab("number of bikes rented per hour")
```



Exploring Data - Hour

Let's look at the interaction between hour and number of bikes rented, hopefully we will be able to see a daily pattern.

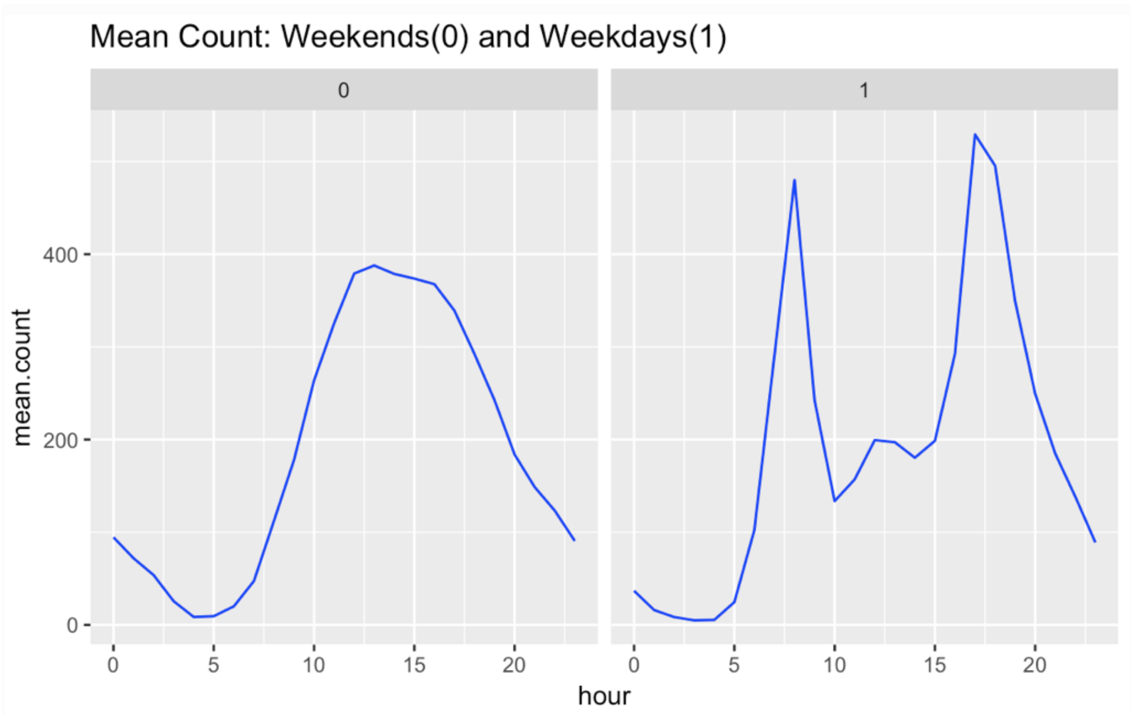
```
ggplot(data=bikes, aes(x=hour, y=count)) +
       geom_point()
```



Exploring Data - Mean Count By Hour

The **aggregate** function is useful for calculating summarising statistics such as means

```
meandata <- aggregate(bikes$count, list(bikes$hour, bikes$workingday),mean)
names(meandata)<-c("hour","workingday","mean.count")

ggplot(data=meandata, aes(x=hour, y=mean.count)) +
  geom_line(color='blue') + facet_grid(. ~ workingday) +
   ggtitle("Mean Count: Weekends(0) and Weekdays(1)")
```

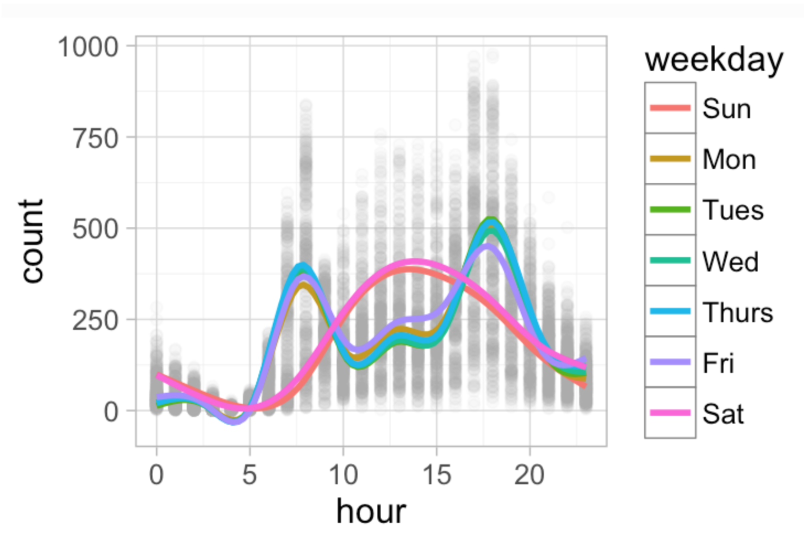


Exploring Data - Day of Week

Somewhat, but it might be better for us to also look at day of week:

```
bikes$weekday <- wday(as.Date(bikes$date), label=TRUE)
```

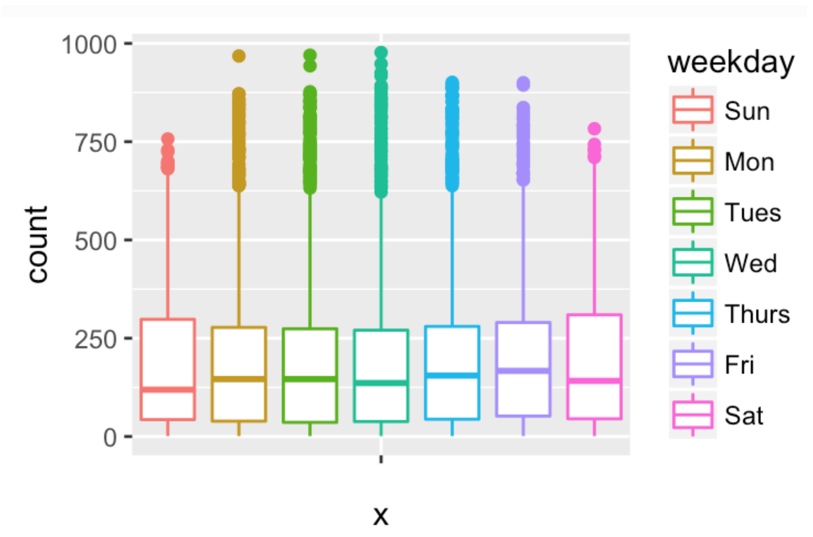
```
ggplot(data=bikes, aes(x=hour, y=count, color=weekday)) +
       geom_point(alpha=0.05, color="darkgray")+
       geom_smooth(fill=NA) +
       theme_light()
```



Exploring Data - Day of Week

Let's take look also at the overall counts by number of days per week:

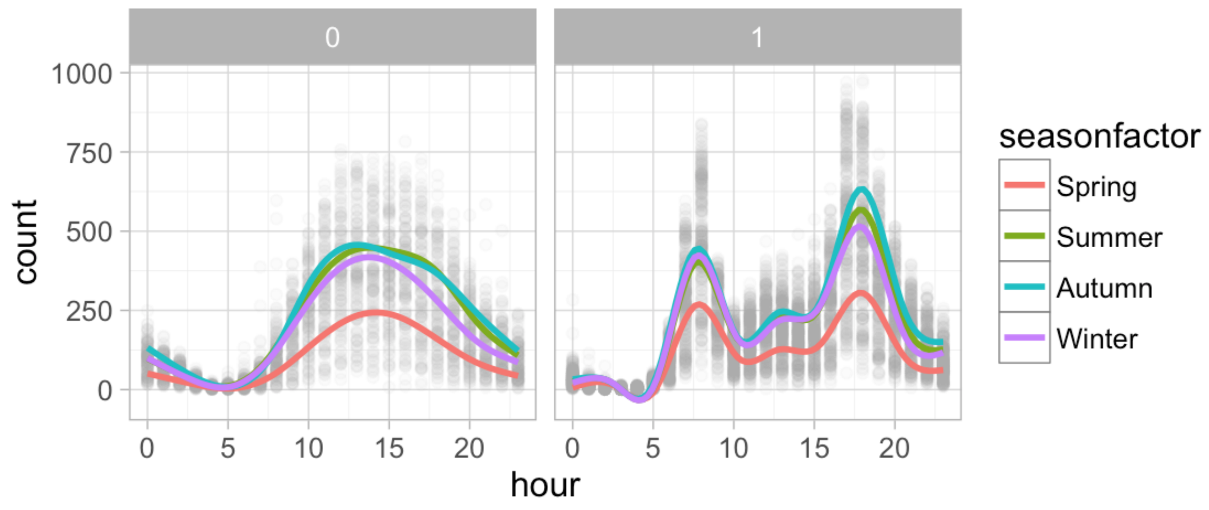
```
ggplot(data=bikes, aes(x="", y=count, color=weekday)) +
        geom_boxplot(position=position_dodge(1))
```



Exploring Data - Seasons

Let's check whether the seasons make a difference to this:

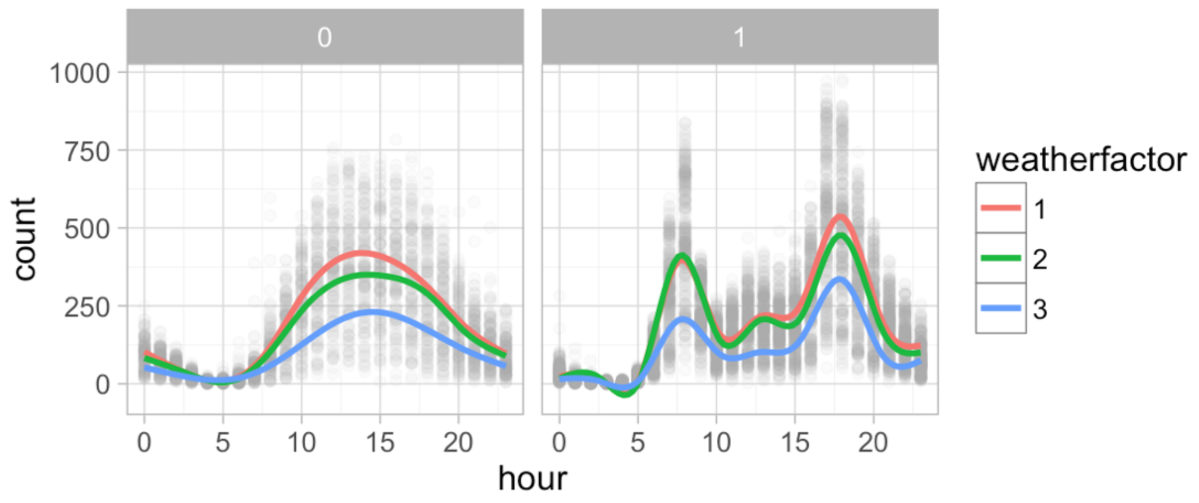
```
ggplot(data=bikes, aes(x=hour, y=count, color=seasonfactor)) +
       geom_point(alpha=0.05, color="darkgray") +
       geom_smooth(fill=NA) +
       facet_grid(. ~ workingday) + theme_light()
```



Exploring Data - Weather

Let's check whether the weather makes a difference to this (note weather = 4 has been auto-removed as there is only 1 point):

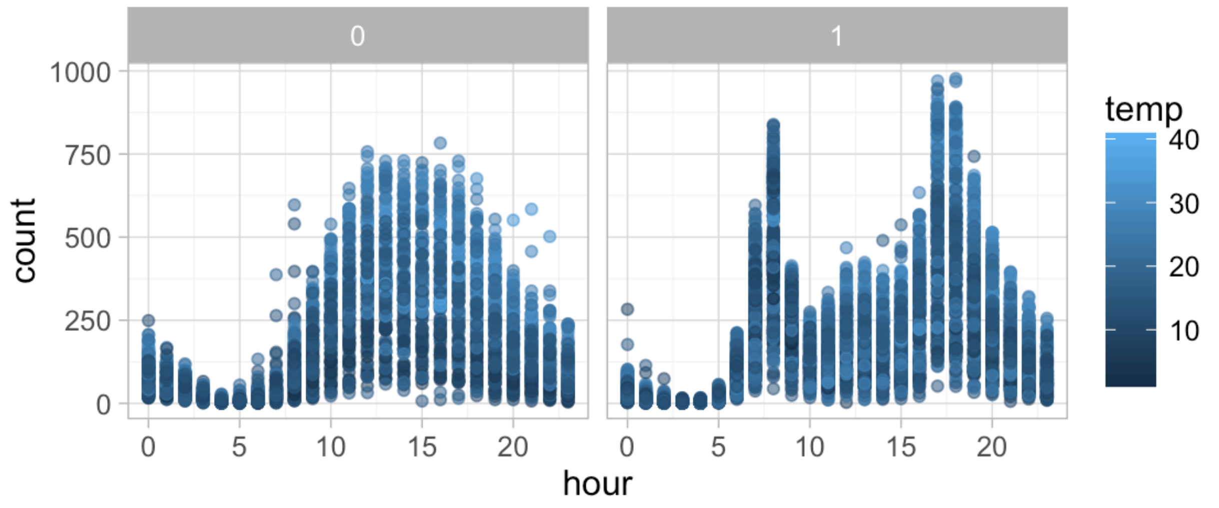
```
ggplot(data=bikes, aes(x=hour, y=count,color=weatherfactor))
+
        geom_point(alpha=0.05, color="darkgray") +
        geom_smooth(fill=NA) +
        facet_grid(. ~ workingday) + theme_light()
```



Exploring Data - Temperature

R can also include a colour scale for continuous variables:

```
ggplot(data=bikes, aes(x=hour, y=count, color=temp)) +
        geom_point(alpha=0.6) +
        facet_grid( ~ workingday) + theme_light()
```

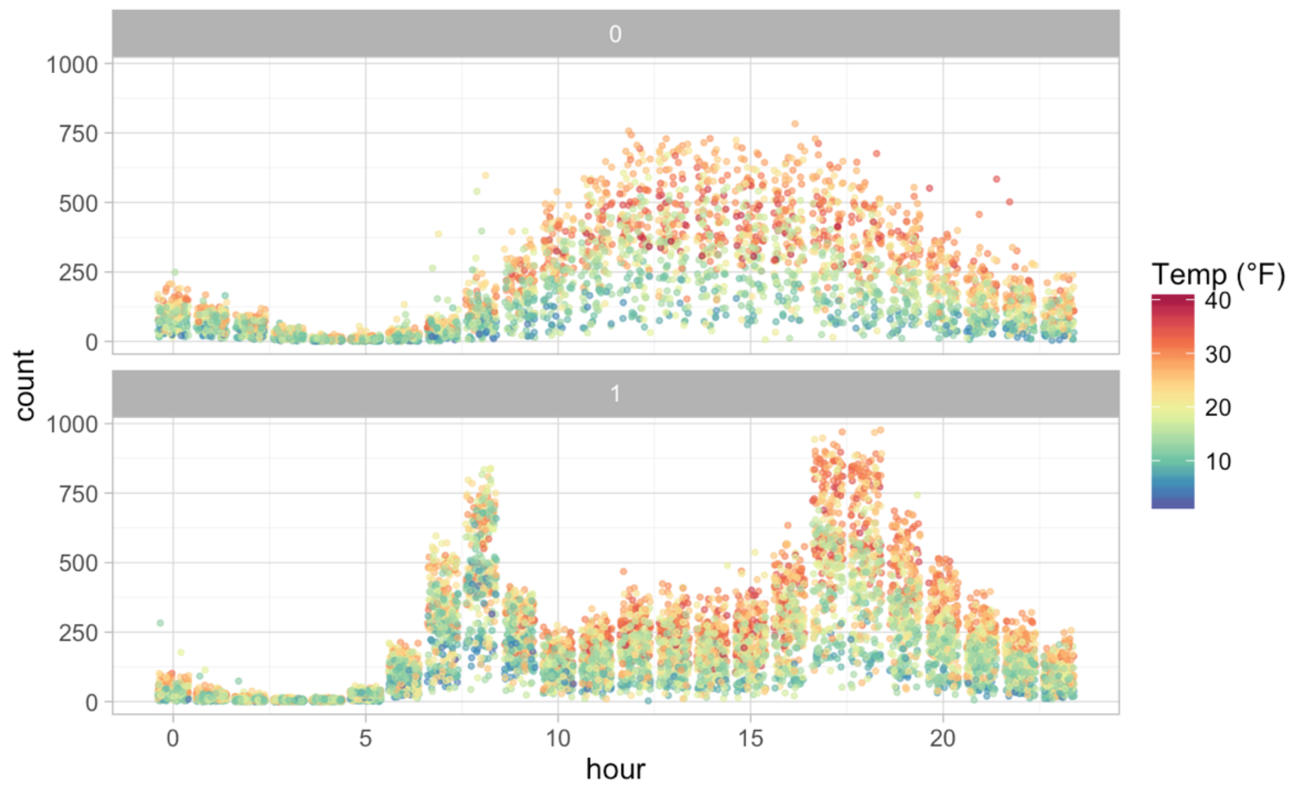


Exploring Data - Temperature

As from here, a couple of good ideas are to include a temperature colour scale, and add some "jitter" to the x axis.

```
ggplot(data=bikes, aes(x=hour, y=count, color=temp)) +
       geom_point(alpha=0.6,
                  size=0.75,
                  position=position_jitter(w=1,h=0)) +
      scale_colour_gradientn("Temp (°F)",
                             colours=c("#5e4fa2", "#3288bd",
                                       "#66c2a5", "#abdda4",
                                       "#e6f598", "#fee08b",
                                       "#fdae61", "#f46d43",
                                       "#d53e4f", "#9e0142")) +
       facet_wrap( ~ workingday, ncol=1) + theme_light()
```

Exploring Data - Temperature

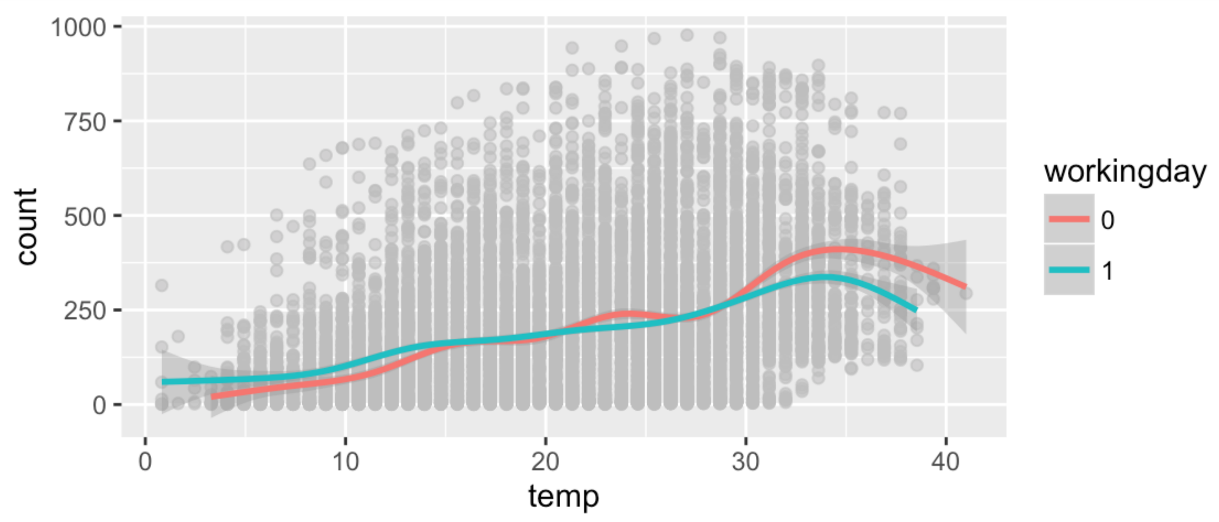


Exploring Data - Temperature

UCL SCHOOL OF MANAGEMENT

Let's visually look at a basic assumption that higher temperature causes higher numbers of bikes to be rented:

```
ggplot(data=bikes, aes(x=temp, y=count,
                                color=as.factor(workingday))) +
       geom_point(alpha=0.6, color="gray") +
       geom_smooth() +
       guides(color=guide_legend(title = "workingday"))
```

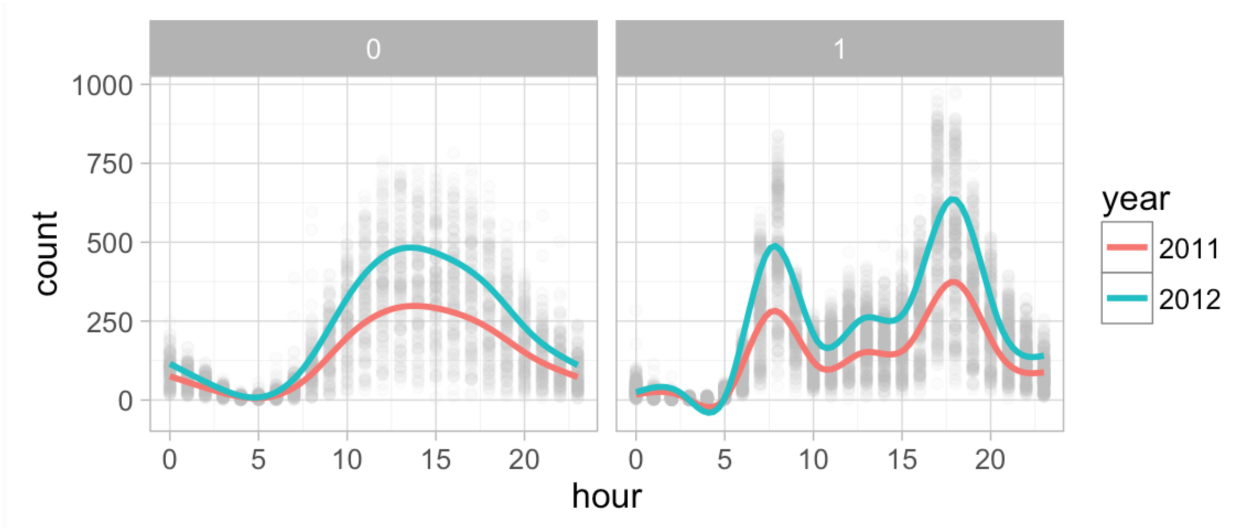


Exploring Data - Year

So far we have been looking at the data over both years (2011 and 2012), but there may have been a different level for each of those years, let's have a look:

```
bikes$year <- as.factor(year(ymd_hms(bikes$datetime)))
```

```
ggplot(data=bikes, aes(x=hour, y=count, color=year)) +
       geom_point(alpha=0.05, color="gray") + geom_smooth(fill=NA) +
       facet_grid(. ~ workingday) + theme_light()
```



Exploring Data - Summary

Using ggplot we have produced:

- boxplots
- histograms
- scatter diagrams
- smoothed trendlines
- facet grids

ggplot is very flexible and powerful, there are some excellent examples here.

Modeling and prediction

So now we have some understanding of how some of these factors impact the target variable (count), the next step is to build and train a model with the information that we have, and see whether we are able to predict the target variables of the test set.

Test and training sets

Seeing as Kaggle does not provide us with the target variables of the test set, we will only be able to know the predictive accuracy once we submit our results to Kaggle.

So it may be a useful idea for us to create our own test set within our data, so that we can internally test using that.

```r
#library(caret)

set.seed(1)
intrain <- createDataPartition(y = bikes$count, p = 0.9, list = FALSE)

bikestrain <- bikes[intrain, ]
bikestest <- bikes[-intrain, ]
```

bikes	bikestrain	bikestest
10886	9799	1087

Simple Linear Modelling

UCL SCHOOL OF MANAGEMENT

To start simply, let's build a model that tries to explain the # of people renting a bike based only on temperature.

```
linearmodel.temperatureonly <- lm(count ~ temp, bikestrain)
```

```
summary(linearmodel.temperatureonly)
```

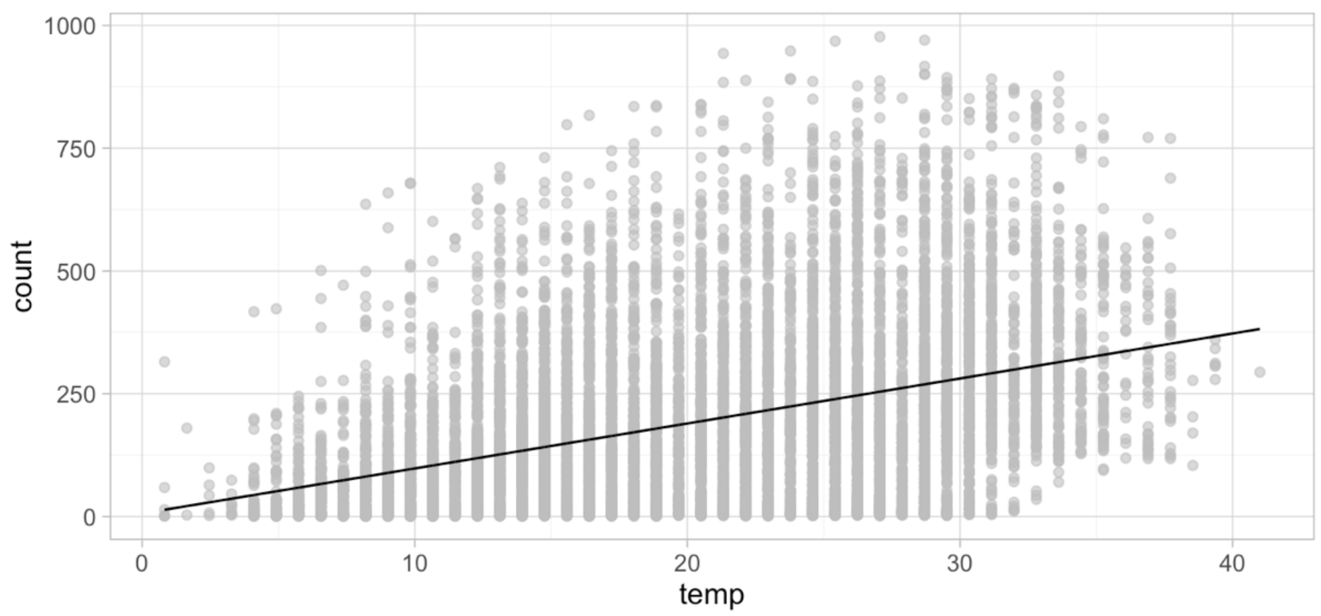
Simple Linear Modelling

UCL SCHOOL OF MANAGEMENT

```
Call:
lm(formula = count ~ temp, data = bikestrain)

Residuals:
    Min      1Q  Median      3Q     Max 
-288.23 -112.58  -33.09   78.71  741.42 

Coefficients:
            Estimate Std. Error t value Pr(&gt;|t|)    
(Intercept)   6.2745     4.6807   1.341     0.18    
temp          9.1607     0.2159  42.430   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '' 1

Residual standard error: 166.5 on 9797 degrees of freedom
Multiple R-squared:  0.1552, Adjusted R-squared:  0.1551 
F-statistic:  1800 on 1 and 9797 DF,  p-value: < 2.2e-16
```

Visually

```
ggplot(data=bikestrain, aes(x=temp, y=count)) +
       geom_point(alpha=0.6, color="gray") + theme_light() +
       geom_line(data=data.frame(temp=bikestrain$temp, count=predict(linearmodel.temperatureonly,bikestrain)))
```



Goodness of fit

Adjusted (R^2) is one of the usual measures for how good the fit of a model is, in our case this is 0.1551, which roughly means that 15% of the variance in the # of bikes rented per hour can be explained by the temperature.

It depends in what field you're working in as to whether this is a good or bad fit.

Goodness of fit / Accuracy

The way that Kaggle, for this competition, assessed accuracy is by using the **Root Mean Squared Logarithmic Error** of the predictions, so this means:

- Use the model to make predictions
- Calculate the below where p_i is the prediction and a_i is the actual value.

RMSLE $= \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\log(p_i + 1) - \log(a_i + 1))^2}$

This penalises predictions that are too low more heavily than those that are too high.

Define our own RMSLE function

```
rmsle <- function(testvalues,predvalues)
    {sqrt(1/length(testvalues)*
       sum((log(predvalues +1)-log(testvalues +1))^2))}
```

Accuracy on our test set

Let's calculate the RMSLE of our model on our internal test set:

```
testvalues <- bikestest$count

predvalues <- predict(linearmodel.temperatureonly, bikestest)

rmsle(testvalues,predvalues)

> [1] 1.441061
```

This number is quite hard to interpret on its own, but if the predicted values were exactly the same as the test values, then it would be =0, so we want to improve on this by making our RMSLE smaller.

More complex linear modelling

```
linearmodel.full <- lm(count ~ temp + I(temp^2) +
                              workingday + holiday +
                              windspeed + weatherfactor +
                              seasonfactor + hour:temp +
                              workingday:temp + hour:year
                              + temp:year + year:workingday
                              + weather:hour + hour:workingday,
                             bikestrain)
```

Including most of the variables (you can see in the summary that almost all are statistically significant). A quadratic form of temperature has been included (because we saw a slight drop off at high temperatures), and only one of the temperature variables has been included.

Accuracy on our test set

Let's calculate the RMSLE of our model on our internal test set:

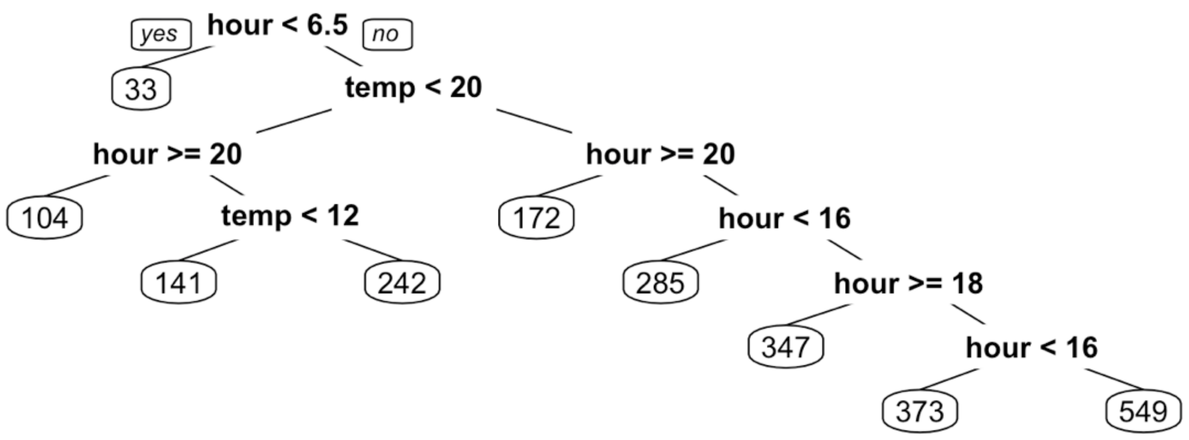
```
testvalues <- bikestest$count

predvalues <- predict(linearmodel.full,bikestest)
predvalues[predvalues<=0] <- 0

rmsle(testvalues,predvalues)

> [1] 1.211355
```

This is an improvement on our previous more simple model, but is still not very highly predictive.

Classification Trees

Let's look at a simple classification tree for our target outcome:

```
#library(rpart) and library(rpart.plot)

treemodel.basic = rpart(count ~ temp + hour, method="anova",
                        data=bikestrain)

prp(treemodel.basic)
```



Classification Trees

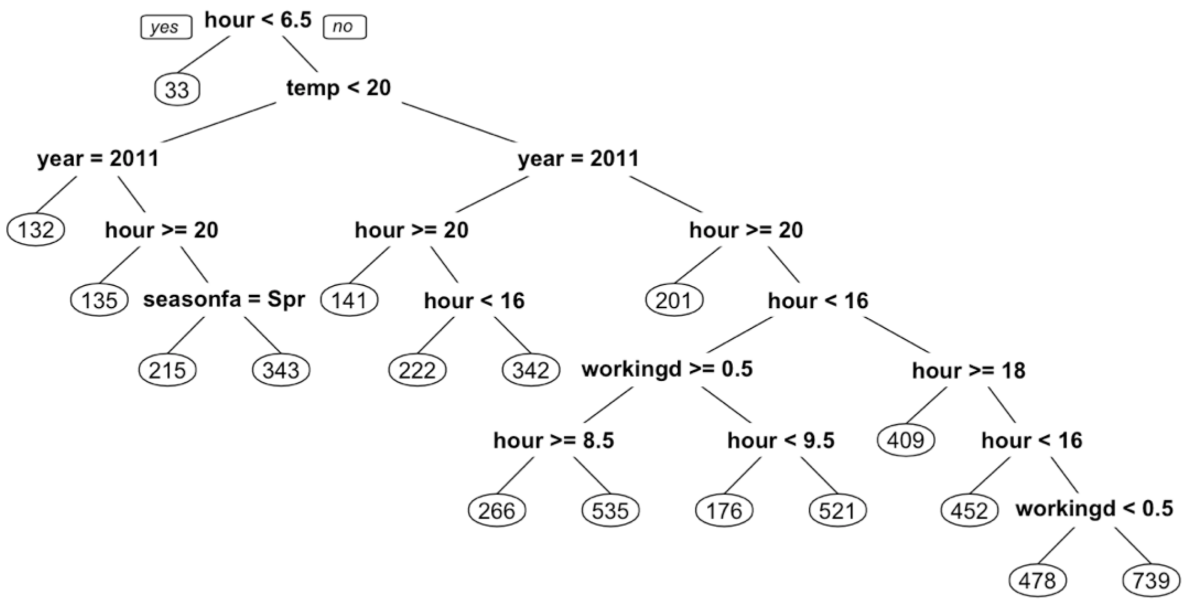
We can make classification trees more complicated by adding in further factors:

```
treemodel.full = rpart(count ~ temp + I(temp^2) +
                               hour + year +
                               workingday + holiday +
                               windspeed + weatherfactor +
                               seasonfactor, method="anova",
            data=bikestrain)
```

Classification Trees

We can make classification trees more complicated by adding in further factors:

```
prp(treemodel.full)
```



Classification Trees

The rpart package calculates splits that reduce the variance within groups, but sometimes do not include all of the variables, e.g.:

```
printcp(treemodel.full)

>
Regression tree:
rpart(formula = count ~ temp + I(temp^2) + hour + year + workingday + holiday + windspeed + weatherfactor + seasonfactor,
    method = "anova")

Variables actually used in tree construction:
[1] hour  seasonfactor  temp   workingday   year

Root node error: 321489246/9799 = 32808

n= 9799

         CP nsplit rel error  xerror      xstd
1  0.310462      0   1.00000 1.00030 0.0182851
2  0.089875      1   0.68954 0.68983 0.0144929
3  0.051388      2   0.59966 0.60009 0.0127472
4  0.038177      3   0.54828 0.54867 0.0111318
5  0.036638      4   0.51010 0.51692 0.0105358
6  0.030492      5   0.47346 0.47398 0.0095966
7  0.018216      6   0.44297 0.44354 0.0086683
8  0.017767      7   0.42475 0.42148 0.0083086
9  0.015326     10   0.37145 0.37265 0.0075809
10 0.014589     11   0.35613 0.35663 0.0071867
11 0.014461     12   0.34154 0.34913 0.0070331
12 0.011666     14   0.31262 0.31495 0.0063529
13 0.010000     16   0.28928 0.29166 0.0061378
```

Accuracy on our test set

Let's calculate the RMSLE of our model on our internal test set:

```
testvalues <- bikestest$count
predvalues <- predict(treemodel.full,bikestest)
predvalues[predvalues<=0] <- 0

rmsle(testvalues,predvalues)

> [1] 0.8690004
```

This is a significant improvement on our full linear model, this is likely because trees can find combinations and patterns that would have to be explicitly stated in a linear regression model. Also, because (at least given our formulation) our data does not look to meet the standard OLS assumptions of normally distributed errors.

Random Forest

Classification trees can be useful on their own, usually for categorical variables, but often they are ensembled into "random forests" to improve performance.

```
#library(randomForest)

randomforestmodel <- randomForest(count ~ .,
                        bikestrain[,c(2:9,12:14,18)],ntree=100)
```

Removed casual and registered (columns 10 and 11), and also removed the factor variables (note on this later).

Some caution on the choice of ntree, larger ntree are likely to increase accuracy, but at the cost of algorithm running time.

Accuracy on our test set

Let's calculate the RMSLE of our model on our internal test set:

```
testvalues <- bikestest$count

predvalues <- predict(randomforestmodel,bikestest)
predvalues[predvalues<=0] <- 0

rmsle(testvalues,predvalues)

> [1] 0.5064699
```

This is a significant improvement over the previous models.

Notes

This actual competition had an additional rule of "Your model should only use information which was available prior to the time for which it is forecasting."

So for each month, you could only train using the data you had before the 19th of that month, and then you had to predict the rest of the month. So for example you could not use June 2012s data to build the model for March 2011, which we have done in our example.

To do this you would have to loop through the months and make individual models, an example can be seen (here)[https://www.kaggle.com/benhamner/bike-sharing-demand/random-forest-benchmark/code].

To submit results to kaggle

To submit your results to kaggle you need to make a submission file and load it here https://www.kaggle.com/c/bike-sharing-demand/submissions/attach. The code at the end of the demo file will help you do this, all you need to do is change the model that is passed to *tempmodel*.

Note: there are some issues with factors, because some packages do not like to have a variable with one factor, and when you run through groups of test data for only January 2011, then the season would only be Winter. There are surely ways around this, but for ease you may want to use these variables as integers.

Our results

If we submitted the basic temperature model then we would have been 3042/3252, with a RMSLE of 1.36143. (Interestingly, approximately 6.5% of people who submitted to this competition had predictions less accurate than this).

The classification tree would have brought us to 2642/3252 (81%), with a RMSLE of 0.80674.

The random forest model: 2325/3252 (71%), with a RMSLE of 0.62657.

Next Steps

There are a few key ways to improve performance:

- Predict Casual and Registered users separately and add the predictions together
- Use more complicated models, e.g. using boosting or neural networks
- Create new variables e.g. combine temperature and hour in some way (can use principal components analysis)

Some reference points

To review what we've covered, there are good introductions to R for free online including here and with more detail here: here.

There are also some excellent cheat sheets for R that can be found here.