Basics of Probability

Dublin R

November 7, 2013

Contents

1 Overview

- Overview
- Basics of Probability (some definitions, the `prob` package)
- Dice Rolls and the Birthday Distribution (histograms)

- Gambler's Ruin (plotting functions)
- The Monty Hall Problem
- Probability Distributions (continuous and discrete distributions)

Book: ***Introduction to Probability and Statistics using R*** by G Kerns (downloadable for free online) We will use chapters 4,5 and 6.
Packages: The `prob` package (for the first part only).

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

2 Some Probability Definitions

Random Experiment A random experiment is one whose outcome is determined by chance, with an outcome that may not be predicted with certainty beforehand. Common examples are coin tosses and dice rolls.

Sample Space For a random experiment E, the set of all possible outcomes of E is called the sample space and is denoted by the letter S . For a coin-toss experiment, S would be the results “Head” and “Tail”, for a single roll of a die it is the numbers 1 to 6.

Events An event A is merely a collection of outcomes, or in other words, a subset of the sample space.

3 The Birthday Paradox

3.1 The Birthday function

The `R` command `pbirthday()` computes the probability of a coincidence of a number of randomly chosen people sharing a birthday, given that there are n people to choose from. Suppose there are four people in a room. The probability of two of them sharing a birthday can be computed as follows: (Answer: about 1.6 %)

```
> pbirthday(4)
[1] 0.01635591
```

How many people do you need for a greater than 50% chance of a shared birthday for n people? Many would make the guess 183 people. Let us use the `sapply()` command. The first arguments is the data set (here a sequence of integers from 2 to 60) and then the command or function (here `pbirthday()`, but specified without the brackets. (starting from 2 – so the 5th element corresponds to 6 people, etc)

```
 > sapply(2:60,pbirthday)
 [1] 0.002739726 0.008204166 0.016355912 0.027135574
 [5] 0.040462484 0.056235703 0.074335292 0.094623834
 [9] 0.116948178 0.141141378 0.167024789 0.194410275
[13] 0.223102512 0.252901320 0.283604005 0.315007665
[17] 0.346911418 0.379118526 0.411438384 0.443688335
[21] 0.475695308 0.507297234 0.538344258 0.568699704
[25] 0.598240820 0.626859282 0.654461472 0.680968537
[29] 0.706316243 0.730454634 0.753347528 0.774971854
[33] 0.795316865 0.814383239 0.832182106 0.848734008
[37] 0.864067821 0.878219664 0.891231810 0.903151611
[41] 0.914030472 0.923922856 0.932885369 0.940975899
[45] 0.948252843 0.954774403 0.960597973 0.965779609
[49] 0.970373580 0.974431993 0.978004509 0.981138113
[53] 0.983876963 0.986262289 0.988332355 0.990122459
[57] 0.991664979 0.992989448 0.994122661
```

The answer is 23 people (see entry 22)- probably much less than you thought!!

4 Playing with Dice

Sequence of Integers:
A sequence of integers can be created using the ":" operator, specifying the lowest value and highest value on either side. This is very useful for *Dice* experiments. Importantly this sequence is constructed as a vector.

```
Dice = 1:6
```

The `sample` function :
Another important command is the `sample()` command. This command instructs `R` to select n items from the specified vector. We can use it to simulate one roll of a die.

```
Dice = 1:6
sample(Dice,1)
```

```
> sample(Dice,1)
[1] 2
>
> sample(Dice,1)
[1] 4
>
> sample(Dice,1)
[1] 1
>
> sample(Dice,1)
[1] 2
```

Suppose we wish to simulate two rolls of a die. Surely we just specify 2 in the argument of the `sample()` command. In fact we do, but this is not enough. The `sample()` command works on the default basis of ***sampling without replacement***. That is to say: once a number has been selected, it can not be selected again. In realiy you can roll a "two" several times in succession.

```
> sample(Dice,6)
[1] 4 3 6 1 2 5
> sample(Dice,7)
Error in sample(Dice, 7) :
  cannot take a sample larger than the population when 'replace = FALSE'
```

What we do here is to additionally specify "`replace=TRUE`". This specifies an experiment where there is a ***sampling with replacement*** scheme. We can also use the `table()` function to study the outcomes of the simulation.

```
> sample(Dice,100,replace=TRUE)
 [1] 4 1 3 2 5 4 4 4 6 5 6 6 5 6 6 5 1 1 1 3
 .....
>
> X <- sample(Dice,100,replace=TRUE)
> table(X)
X
 1  2  3  4  5  6
17 19 22 12 17 13
> mean(X)
[1] 3.32
```

In this last example, we have 22 Threes and only 12 sixes. There is nothing particularly unusual about these ***Sampling fluctuations***

From now ow, let's work on the basis of 100 dice rolls, and for the sake of simplicity let us consider the **sum** of those 100 rolls. Importantly, we are simulating a **fair** die. We expect to get a value of approximately 350, but each time we perform the experiment there will always be slight fluctuations. We will never get 350 each time (in fact, rarely). Try the last line of the code below a few times. How many times do you get a figure less than 340 or greater than 360?

```
X <- sample(Dice,100,replace=TRUE)
sum(X)

#Equivalently
sum(sample(Dice,100,replace=TRUE))
```

4.1 Dice Experiment : Simulation Study

Lets consider this ***Dice Roll*** Summation experiment. We will perform the experiment 100000 times, and see what sort of distribution of summations we get. We will save the results in a vector called `sums`. We will use a FOR loop, used to repeat runs a specifed block of code, for the specified number of times.

```
Sums=numeric()    # Initialize An Empty Vector
M=100000  # Number of Iterations

#FOR loop
for (i in 1:M)
    {
     #Generate a New Value
     NewSum=sum(sample(Dice,100,replace=TRUE))

     #Add it to the record
     Sums = c(Sums, NewSum)
     }
```

We can perform some basic statistical operations to study this vector. In particular we are interested in the extremes: How many times was there a summation less than 300, and how many times was there a summation greated than 400? (around 1.5% probability in each case)

```
> length(Sums[Sums<300])
[1] 144
> length(Sums[Sums>400])
[1] 160
```

Lets us look at a histogram (a type of bar chart) of the `Sums` vector (Use around "`breaks =100`" to specify more intervals). What sort of shape is this histogram?

```
hist(Sums, breaks=100)
```

4.2 Central Limit Theorem

This is a very crude introduction to the ***Central Limit Theorem***. Even though the Dice Rolls are not normally distributed, the distribution of summations, are described in this experiment, are from a normally distributed sampling population. Also consider the probability of getting a sum more than 400.

Recalling that dice simulation is for fair dice, the probability of getting a score more extreme than 400 is 1.5% approximately. This provides (again crudely) an introduction to the idea of $p-$values , which are used a lot in statistical inference procedures.

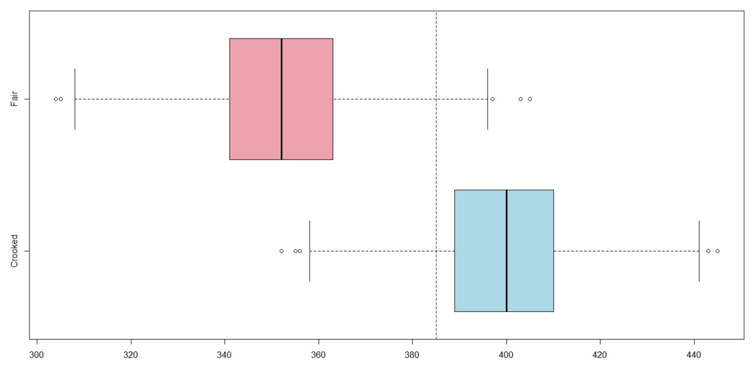

Figure 1

Crooked Dice

Let us consider a crooked dice, where 4,5 and 6 are twice a likely to appear. Try out the following code.

```
# Let 4,5, and  6 come up twice as often as 1,2 and 3

CrookedDice=c(1,2,3,4,4,5,5,6,6)

sum(sample(CrookedDice,100,replace=TRUE))
```

Suppose it was not certain whether a die was fair or crooked favouring higher values such as 4,5 and 6.

The 100 roll experiment was performed and the score turned out to be 400. It would be a very unusual outcome for a fair die, but not impossible. For crooked dice, larger summations would be expected and a score of approximately 400 would be fairly common. Would you think the die was fair or crooked?

```
FairDice <- 1:6
CrookedDice<-c(1,2,3,4,4,5,5,6,6)

######################################################



Fair=numeric()
Crook=numeric()    # Initialize An Empty Vector
M=1000  # Number of Iterations

#FOR loop
for (i in 1:M)
    {
     #Generate a New Value
     FairSum=sum(sample(FairDice,100,replace=TRUE))
     CrookSum=sum(sample(CrookedDice,100,replace=TRUE))

     #Add it to the record
     Fair = c(Fair, FairSum)
     Crook = c(Crook,CrookSum)

     }

######################################################


DiceSums = data.frame(
     Sums=c(Fair,Crook),
     DiceType=as.factor(c(rep("Fair",1000),
          rep("Crooked",1000) ))
     )


######################################################
boxplot(DiceSums$Sums~DiceSums$DiceType,
   horizontal=T,col=c("lightblue","lightpink2"))

abline(v=385,col="blue",lty=2)
```

5 The `prob` package

The sample space of a coin toss experiment can be written out using the `tosscoin()` function on the `prob` package, specifying the number of tosses. The size of the sample space is 2^n for n coin tosses, for binary outcome experiments. For 8 coin tosses, the sample space contains 256 possible outcomes.

```
> tosscoin(1)
  toss1
1     H
2     T
> tosscoin(2)
  toss1 toss2
1     H     H
2     T     H
3     H     T
4     T     T
```

There is a similar command for dice roll : `rolldie()`. Again, specify the number of rolls. For n dice rolls, there are 6^n outcomes in the sample space. (It gets large very quickly).

```
> rolldie(1)
  X1
1  1
2  2
3  3
4  4
5  5
6  6
```

The `cards()` command describes each card from a deck of cards. The `roulette()` commands describes each possible spin from a roulette wheel.

5.1 The prob package : computing probabilities

We can evaluate the probability associated with each sample point using the `makespace` argument.

```
> rolldie(1,makespace=TRUE)
  X1     probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
>
> tosscoin(3,makespace=TRUE)
  toss1 toss2 toss3 probs
1     H     H     H 0.125
2     T     H     H 0.125
3     H     T     H 0.125
4     T     T     H 0.125
5     H     H     T 0.125
6     T     H     T 0.125
7     H     T     T 0.125
8     T     T     T 0.125
```

We can use this to compute the probability of certain events. Suppose we wish to compute the probability of a sum of 28 or more from five dice rolls. Importantly, each column of the output has a name. X1, X2 etc. Lets subset the sample space such that the sum of the 5 X variables is greater than or equal to 28.

```
subset(rolldie(5,makespace=TRUE), X1 + X2 + X3 + X4 + X5 >= 28)
X = subset(rolldie(5,makespace=TRUE), X1 + X2 + X3 + X4 + X5 >= 28)
names(X)
X$prob
sum(X$prob)
```

```
> X = subset(rolldie(5,makespace=TRUE), X1 + X2 + X3 + X4 + X5 >= 28)
> names(X)
[1] "X1"    "X2"    "X3"    "X4"    "X5"    "probs"
>
> X$prob
 [1] 0.0001286008 0.0001286008 0.0001286008 0.0001286008
 [5] 0.0001286008 0.0001286008 0.0001286008 0.0001286008
 [9] 0.0001286008 0.0001286008 0.0001286008 0.0001286008
[13] 0.0001286008 0.0001286008 0.0001286008 0.0001286008
[17] 0.0001286008 0.0001286008 0.0001286008 0.0001286008
[21] 0.0001286008
>
>
 sum(X$prob)
[1] 0.002700617
```

5.2 The prob package ; Cards example

Compute the probability of a King or Queen.

```
S <- cards(,makespace=TRUE)
subset(S, rank %in% c("Q","K"))
```

```
> subset(S, rank %in% c("Q","K"))
   rank    suit      probs
11    Q    Club 0.01923077
12    K    Club 0.01923077
24    Q Diamond 0.01923077
25    K Diamond 0.01923077
37    Q   Heart 0.01923077
38    K   Heart 0.01923077
50    Q   Spade 0.01923077
51    K   Spade 0.01923077

> X = subset(S, rank %in% c("Q","K"))
> sum(X$probs)
[1] 0.1538462
```

6 Gambler's Fallacy

Figure 2

The ***Gambler's fallacy***, also known as the ***Monte Carlo fallacy*** (because its most famous example happened in a Monte Carlo Casino in 1913),and also referred to as the fallacy of the maturity of chances, is the belief that if deviations from expected behaviour are observed in repeated independent trials of some random process, future deviations in the opposite direction are then more likely. *(Wikipedia)*

6.1 Monte Carlo Casino

The most famous example of the gambler's fallacy occurred in a game of roulette at the Monte Carlo Casino on August 18, 1913, when the ball fell in black 26 times in a row. This was an extremely uncommon occurrence, although no more nor less common than any of the other 67,108,863 sequences of 26 red or black. Gamblers lost millions of francs betting against black, reasoning incorrectly that the streak was causing an "imbalance" in the randomness of the wheel, and that it had to be followed by a long streak of red. *(Wikipedia)*

Figure 3

6.2 Implementation with R

Firstly let simulate the outcomes of a Roulette Wheel.

- For the sake of simplicity, we will disregard ”Green” and let “Black” be signified by an outcome of 1 and “Red” signified by an outcome of 2.
- For this we will use the `runif()` command, as well as the `ceiling()` command, which rounds a value up to the next highest integer.

```
runif(5)
2*runif(5)
ceiling(2*runif(5)) #Ones and Twos
```

```
> runif(5)
[1] 0.02646220 0.90602044 0.45596144 0.25390162 0.06416899
>
>
> 2*runif(5)
[1] 1.4458583 0.7452968 0.7861305 0.4930401 1.9711546
> ceiling(2*runif(5))
[1] 2 2 2 1 1
```

In this last code segment, we get “Red” three times in a row, and then two ”Blacks”. Try it for a larger number of trials.(e.g. 100)

```
ceiling(2*runif(1000))
```

- What is of interest is the number of repeated colours. What we could do is to construct a `For` loop so as to monitor how often a colour repeats.
- Each time a new colour comes up, the sequence counter gets set to 1. If the next spin results in the same colour, the sequence number is set to 2, if it happens again, the next sequence number is 3, and so on.
- Firstly let set up a basic `FOR` loop to generate the colours. Ths code is more elaborate than the approach we used already, but it is easy to use this for studying repetitions.

```
M=100

#First Spin
Colour=ceiling(2*runif(1))

for(i in 2:M)
  {
  # Next Colour
  NextCol =  ceiling(2*runif(1))
  Colour = c(Colour,NextCol)
  }
```

```
> Colour
  [1] 2 2 2 2 1 1 2 2 1 1 2 1 1 1 2 1 2 2 1 2 1 2 2 1 2 1 2 1 2 1
 [31] 1 1 2 2 2 1 2 2 1 2 2 1 2 2 2 1 1 1 1 2 2 1 2 2 2 2 1 1 2 2
 [61] 1 2 1 2 1 2 2 2 1 1 2 2 1 2 1 1 2 1 2 1 2 2 2 1 1 2 2 1 1 2
 [91] 1 1 2 2 1 2 1 1 2 2
```

We are going to do something similar here, but record the sequence lengths.

```
M=100

#First Spin
Colour=ceiling(2*runif(1))



# Start a vector with a single value of 1.
SeqNo=c(1)

for(i in 2:M)
  {
  # Next Colour
  NextCol =  ceiling(2*runif(1))
  Colour = c(Colour,NextCol)



  #If the current colour is the same as the last, then the current
  #value in the sequence number vector  is 1 more than the last.
  #
  #Otherwise the current sequence number is reset to 1.
  if (Colour[i] == Colour[i-1])
    {
    SeqNo[i] = SeqNo[i-1]+1
    }else SeqNo[i]=1
  }
```

```
> max(SeqNo)
[1] 5
>
> cbind(Colour,SeqNo)
      Colour SeqNo
 [1,]      2     1
 [2,]      2     2
 [3,]      2     3
 [4,]      1     1
 [5,]      2     1
 [6,]      2     2
 [7,]      1     1
 [8,]      1     2
 [9,]      2     1
[10,]      1     1
```

To reduce data that needs to be collected, we will look at a *Sequence Maximum.*

- If there is a change of colour, the last sequenc number is added to a special vector: `SeqMax`.
- For the sake of brevity, Any values lower than 3 in `SeqMax` will be discarded afterwards.

```
M=100

#First Spin
Colour=ceiling(2*runif(1))
SeqNo=c(1)

SeqMax=numeric()

for(i in 2:M)
  {
  # Next Colour
  NextCol =  ceiling(2*runif(1))
  Colour = c(Colour,NextCol)


  if (Colour[i] == Colour[i-1])
    {
    SeqNo[i] = SeqNo[i-1]+1
    }else{SeqNo[i]=1;SeqMax=c(SeqMax,SeqNo[i-1])}
}
SeqMax = SeqMax[SeqMax>3]
```

Increase the number of iterations to a large number, say 100,000. Then see what turns up in the `SeqMax` vector. Use the `table()` command to determine the distribution.

```
> table(SeqMax[SeqMax>10])

11 12 13 14 15 18
21  6  2  3  1  1
```

7 Gambler's Ruin

Consider a gambler who starts with an initial fortune of A dollars and then on each successive gamble either wins or loses independent of the past with probabilities p and $q = 1 - p$ respectively. This gamblers places bets with the Banker, who has an initial fortune of B dollars.

(For the sake of simplicity, we will look at the game from the perspective of the gambler only. The Banker is, by convention, the richer of the two (i.e. A¡B) , and has a better chance of winning.)

- Probability of successful gamble for gambler : p
- Probability of unsuccessful gamble for gambler : q (where $q = 1 - p$)
- Ratio of success probability to failure success: $s = p/q$
- Conventionally the game is biased in favour of the Banker (i.e.$q > p$ and $s < 1$)

7.1 Simulating a Single Gamble

To simulate one single bet, compute a single random number between 0 and 1. For this, we use the `runif()` command, which uses generates a ***continuous uniform random variable***. The upper and lower limits can be specified. In the absence of particular specifications, the defaults are 0 and 1.

```
runif(1)
```

Lets assume that the game is biased in favour of the Banker with $p = 0.45$,$q = 0.55$. If the number is less than 0.45, the gamble wins. Otherwise the Banker wins.

```
> runif(1)
[1] 0.1251274
>#Gambler Wins
>
> runif(1)
[1] 0.754075
>#Gambler loses
>
> runif(1)
[1] 0.2132148
>#Gambler Wins
>
> runif(1)
```

```
[1] 0.8306269
>#Gambler Loses
```

7.2 Repeated Gambles

Let A be the Gambler's Kitty at the start of the gambling. Let B be the Banker's Wealth at the outset. (Therefore the total jackpot is A+ B) The probability of gambler winning a gamble is p.

- Let Rn denote the Gamblers total fortune after the $n-$th gamble. If the Gambler wins the first game, his wealth becomes $Rn = A + 1$.
- If he loses the first gamble, his wealth becomes $Rn = A - 1$. The entire sum of money at stake is the Jackpot i.e. $A + B$.
- The game ends when the Gambler wins the Jackpot ($Rn = A + B$) or loses everything ($Rn = 0$).
- The vector `Rec` records the gambler's worth on an ongoing basis, with the first element being the value of A. As the gambling progresses, the new value gets added to the vector.

Lets see how this plays out over a night of gambling. Let's start the gambler with 20 dollars, and the banker with 100. Unknown to the gambler is the fact that he or she has only a 47% chance of winning a single bet at each round.

Basic Structure

```
#initial values
A=20;B=100;p=0.47

#record the progress of the gambler - who starts with A wealth
Rec=c(A)

#generate a outcome for each gamble.
probval = runif(1)

if (probval < p)
{
A = A+1; B =B-1
}else{A=A-1;B=B+1}

#Save the values from each bet
Rec=c(Rec,A)
```

7.3 Using a while loop

So far we have managed to code for a single bet. Lets simulate the gambling in its entirety. For this, we will use a `while` loop. A `while` loop will repeat a set of commands as long as a particular logical condition is met. (Here, that logical condition is that the gambler has money to gamble with i.e. $A > 0$). Once he loses everything, the `while` loop (and the gambling) terminates.

```
A=20;B=100;p=0.47
Rec=c(A)

while(A>0)
{
    ProbVal=runif(1)
    if(ProbVal <= p)
      {
      A = A+1; B =B-1
      }else{A=A-1;B=B+1}
    Rec=c(Rec,A)

}
```

We will also include a `break` statement to stop the loop in the unlikely event that the Gambler wins the jackpot.

```
A=20;B=100;p=0.47
Rec=c(A)
Total=A+B  #total jackpot

while(A>0)
   {
   ProbVal=runif(1)
   if(ProbVal <= p)
   {
   A = A+1; B =B-1
   }else{A=A-1;B=B+1}
   Rec=c(Rec,A)
   if(A==Total){break}
   }
```

7.4 The Gambler's Performance

In all likelihood, the Gambler lost everything. But how long did he last in the game? how many gambles was he able to play? Also What was his highest level of wealth? We can look at the ***Rec*** vector to answer these questions.

```
length(Rec)
max(Rec)
```

We are going to write a function, called `GambRuinFunc()` to produce a duration value, for given values of A, B and p.

```
GambRuinFunc=function(A=20,B=100,p=0.47)
{
Rec=c(A)
Total=A+B  #total jackpot

while(A>0)
   {
   ProbVal=runif(1)
   if(ProbVal <= p)
   {
   A = A+1; B =B-1
   }else{A=A-1;B=B+1}
   Rec=c(Rec,A)
   if(A==Total){break}
   }
return(length(Rec))
}
```

We can construct a plot to depict the gambler's ongoing fortunes in the game.

```
# Line plot, colour red
plot(Rec,type="l",col="red")

#Axes
abline(h=0)
abline(v=0)

#Also the gamblers initial value and the total value of the game(optional)
abline(h=A,col="red")
abline(h=Total,col="green")
```

7.5 Simulation Study

Suppose we are interested in the probability distribution of durations. What is the probability of lasting more than 100 rounds? less than 200 etc? We can use a function called `GambRuinFunc()` to compute the durations from an number of nights of gambling (next page) . The function can easily be adjusted to consider the maximum value of the gambler's worth We can specify values for A , B and p different to that of the defaults. Here, they are picked so as to be quick to execute.

```
Durations= numeric()
M=1000
for(i in 1:M)
   {
   NextDuration = GambRuinFunc(A=10,B=30,p=0.40)
   Durations=c(Durations,NextDuration)
   }
```

We can study the Durations vector using simple statistical functions.

```
hist(Durations)
mean(Durations)
median(Durations)
IQR(Durations)
quantile(Durations,1:20/20)
```

```
GambRuinFunc=function(A=20,B=100,p=0.47)
{
Rec=c(A)
Total=A+B  #total jackpot

while(A>0)
   {
   ProbVal=runif(1)
   if(ProbVal <= p)
   {
   A = A+1; B =B-1
   }else{A=A-1;B=B+1}
   Rec=c(Rec,A)
   if(A==Total){break}
   }
Duration=length(Rec)
return(Duration)
}
```

We are going to keep track of the time it takes to carry out this simulation.

```
Time1=Sys.time()
DurDist=numeric()
M=150000
for(i in 1:M){

 DurDist=c(DurDist,GambRuinFunc())
 }

Time2=Sys.time()
#

hist(DurDist,main="Durations Distribution",breaks=100)
quantiles(DurDist)
```

```
hist(Durations)
mean(Durations)
median(Durations)
IQR(Durations)
quantile(Durations,1:20/20)
```

```
GambRuinFunc=function(A=20,B=100,p=0.47)
{
Rec=c(A)
Total=A+B  #total jackpot

while(A>0)
   {
   ProbVal=runif(1)
   if(ProbVal <= p)
   {
   A = A+1; B =B-1
   }else{A=A-1;B=B+1}
   Rec=c(Rec,A)
   if(A==Total){break}
   }
Duration=length(Rec)
return(Duration)
}
```

```
> quantile(DurDist)
  0%  25%  50%  75% 100%
  23  139  239  419 4167
>
> quantile(DurDist,1:10/10)
 10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
  91  123  157  195  239  295  369  481  691 4167
```

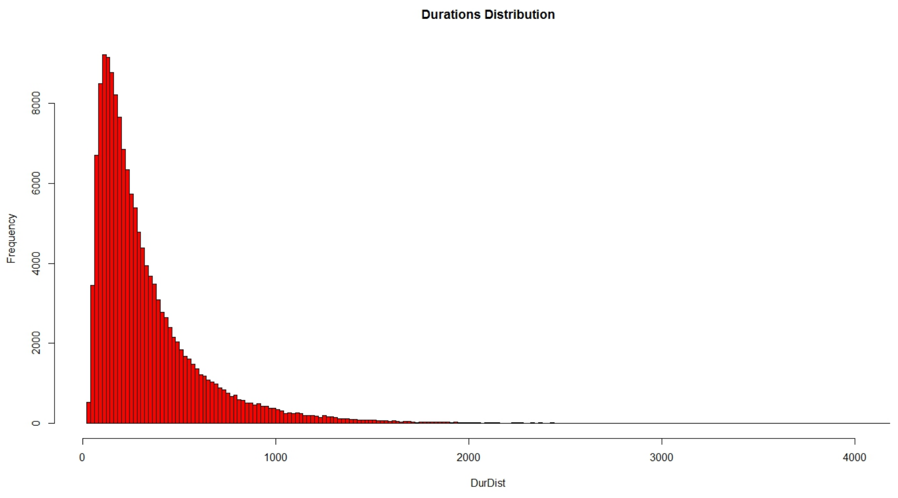

Figure 4

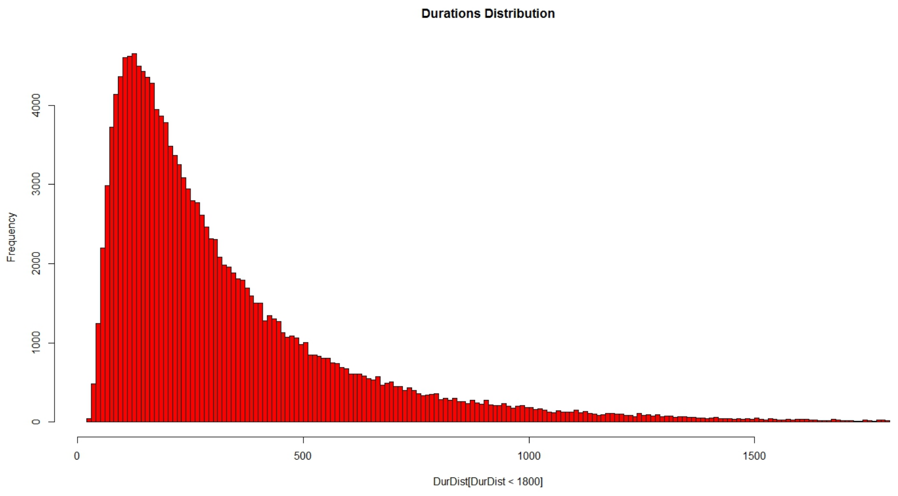

Figure 5