Statistical Computing with R: Masters in Data Sciences 503 (S28) Second Batch, SMS, TU, 2023

Shital Bhandary

Associate Professor

Statistics/Bio-statistics, Demography and Public Health Informatics

Patan Academy of Health Sciences, Lalitpur, Nepal

Faculty, Data Analysis and Decision Modeling, MBA, Pokhara University, Nepal

Faculty, FAIMER Fellowship in Health Professions Education, India/USA.

Review Preview:

- Monte Carlo simulations
  - Randomness
  - Random deviates
  - Resampling
- Use of Monte Carlo methods in Machine Learning
- Class imbalance problem
  - Statistical approach
  - Data science approach
- Missing data
  - Supervised learning
  - Unsupervised learning

Monte Carlo Simulations: https://bstaton1.github.io/au-r-workshop/ch4.html

- Simulation modeling is one of the primary reasons to move away from spreadsheet-type programs (like Microsoft Excel) and into a program like R.
- R allows us to replicate the same (possibly complex and detailed) calculations over and over with different random values.
- We can then summarize and plot the results of these replicated calculations all within the same program.
- Analyses of this type are called **Monte Carlo methods**: they randomly sample from a set of quantities for the purpose of generating and summarizing a distribution of some statistic related to the sampled quantities.

Randomness:

- A critical part of simulation modeling is the use of random processes.
- A **random process** is one that generates a different outcome according to some rules each time it is executed.
- They are tightly linked to the concept of **uncertainty**: you are unsure about the outcome the next time the process is executed.
- There are two basic ways to introduce randomness in R:
- **Random deviates**
- **Resampling**

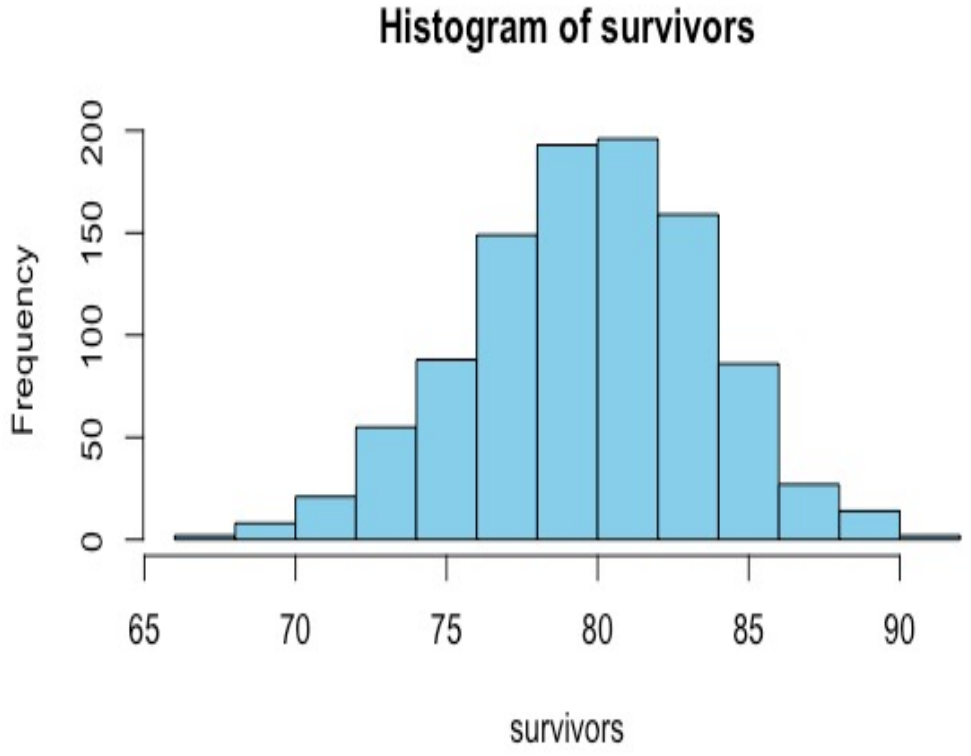
Random deviates:

- At the end of each year, each individual alive at the start can either live or die.
- There are **two outcomes here**, and suppose each individual has an 80% chance of surviving.
- The number of individuals that survive is the result of a **binomial random process** in which there were n individuals alive at the start of this year and p is the probability that any one individual survives to the next year.
- We can execute a binomial random process with p=0.8 and n=100 like this in R:
- **rbinom**(n = 1, size = 100, prob = 0.8)
- I got:
- [1] 83
- But you almost certainly get different number than this one!

We can also plot it with a bit of tweaking:

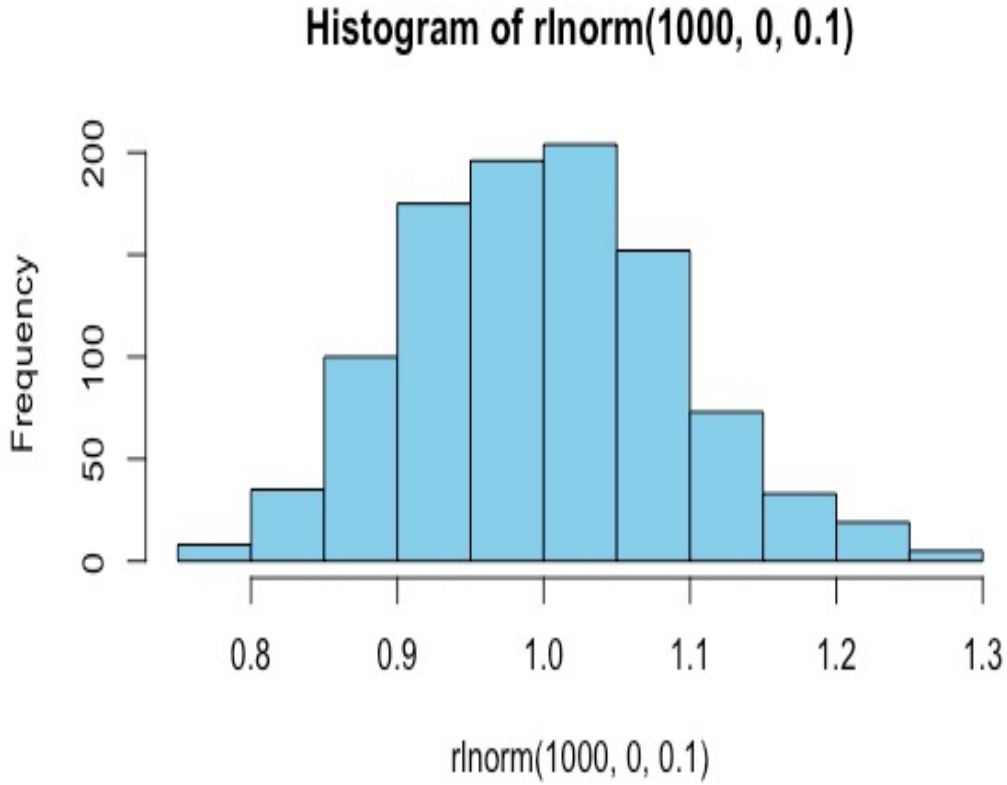
Histogram

survivors = **rbinom**(1000, 100, 0.8)
hist(survivors, col = "skyblue")



We could also use other processes like log normal:

- Another random process is the **lognormal process.**
- It generates random numbers such that the log of the values are normally-distributed with mean equal to **logmean** and standard deviation equal to **logsd**
- **hist**(**rlnorm**(1000, 0, 0.1), col = "skyblue")



Need for sampling:

https://machinelearningmastery.com/monte-carlo-sampling-for-probability/

- There are many problems in probability, and **more broadly in machine learning**, where we cannot calculate an analytical solution directly.
- **Class imbalance problem is such situation in Machine Learning!**
- In fact, there may be an argument that exact inference may be intractable for most practical probabilistic models.
- The desired calculation is typically a **sum of a discrete distribution or integral of a continuous distribution** and is intractable to calculate.
- The calculation may be intractable for many reasons, such as the **large number of random variables**, the stochastic nature of the domain, noise in the observations, the lack of observations, and more.

Resampling:

- Using random deviates works great for creating new random numbers, but what if we already have a set of numbers that we wish to introduce randomness to?
- For this, we can use **resampling techniques**.
- In R, the sample() function is used to sample size elements from the vector x.

#Resampling of 1 to 10:

- **sample**(x = 1:10, size = 5)

#Sample with replacement

- **sample**(x = **c**("a", "b", "c"), size = 10, replace = T)

#Sample with set probabilities

- **sample**(x = **c**("live", "die"), size = 10, replace = T, prob = **c**(0.8, 0.2))

We have used it:

- roll() function defining roll of a fair die twice
- Training and Testing sets definition, cross-validation

Reproducing randomness:

- For reproducibility purposes, we may wish to get the same exact random numbers each time we run our script.

- To do this, we need to set the **random seed**, which is the starting point of the random number generator our computer uses.

#Example:

- **set.seed**(1234)
- **rnorm**(1)
- [1] -1.207066

#Try without random seed

- rnorm(1)
- [1] 0.2774292

Replication:

- To use Monte Carlo methods, we need to be able to replicate some random process many times.
- There are two main ways this is commonly done: either with **replicate()** or with **for()** loops.
- The replicate() function executes some expression many times and returns the output from each execution.
- Say we have a vector x, which represents 30 observations of an animal length (mm):
- x = **rnorm**(30, 500, 30)

Replication in R:

- We wish to build the sampling distribution of the mean length "by hand".
- We can sample randomly from it, calculate the mean, then repeat this process many times.
- This can be done in R with:

```
#Code after x is defined:

means = replicate(n = 1000, expr = {
        x_i = sample(x, length(x),
        replace = T)
        mean(x_i)
})
```

Mean and SE same in x and 1000 replicated means of x? Unbiased estimate of x!

- If we take mean(means) and sd (means), that should be very similar to mean(x) and se(x).
- Create the se() function and prove this using R!
- se = **function**(x) **sd**(x)/**sqrt**(**length**(x))

#Check means first

- **mean**(means); **mean**(x)
- -[1] 492.5897
- [1] 492.6636

#Standard error of mean

sd(means); **se**(x)

[1] 5.130683

[1] 5.023584

Monte Carlo Simulations is based on **Law of Large Numbers**. It can also be used to prove **Regression to Mean** and **Central Limit Theorem**.

More on Law of Large Numbers here: https://machinelearningmastery.com/a-gentle-introduction-to-the-law-of-large-numbers-in-machine-learning/

Replication with “for” loop:

- In programming, a *loop* is a command that does something over and over until it reaches some point that you specify.
- R has a few types of loops: repeat(), while(), and for(), to name a few.
- for() loops are among the most common in simulation modeling.
- A for() loop repeats some action for however many times you tell it **for** each value in some vector.

#For loop syntax:

```
for (var in seq) {
        expression(var)
}
```

Examples:

```
#1
for (i in 1:5) {
    print(i^2)
}
#2
results=numeric(5)
for (i in 1:5) {
   results[i] = i^2 }
results
```

#Output 1

- [1] 1
- [1] 4
- [1] 9
- [1] 16
- [1] 25

#Output 2

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

More:

- nt = 100 *# number of years*
- N = NULL *# container for (fish) abundance*
- N[1] = 1000 *# first **end-of-year** abundance*

#Loop for replication

for (t **in** 2:nt) {

*#N this year is N last year * growth **

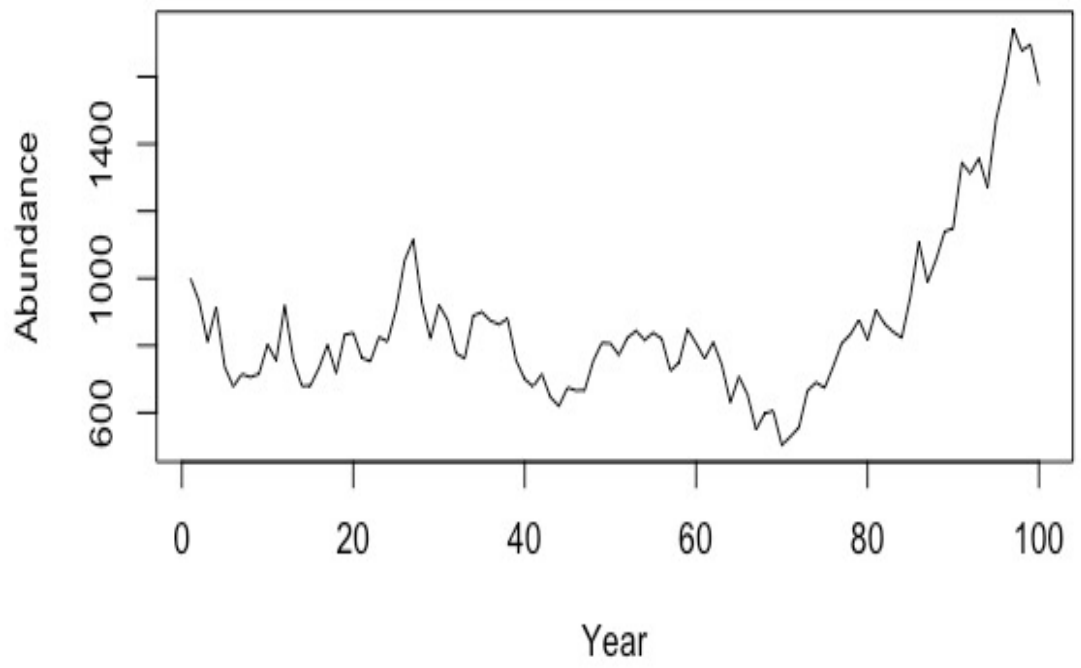
*# randomness * fraction that survive harvest*

N[t] = (N[t-1] * 1.1 * **rlnorm**(1, 0, 0.1)) * (1 - 0.08)

}

Let’s plot it:

- **plot**(N, type = "l", pch = 15, xlab = "Year", ylab = "Abundance")



Function writing for Monte Carlo simulation:

#In Monte Carlo analyses, it is often useful to wrap code into functions.

- This allows for easy replication and setting adjustment (e.g., if you wanted to compare the growth trajectories of two populations with differing growth rates).

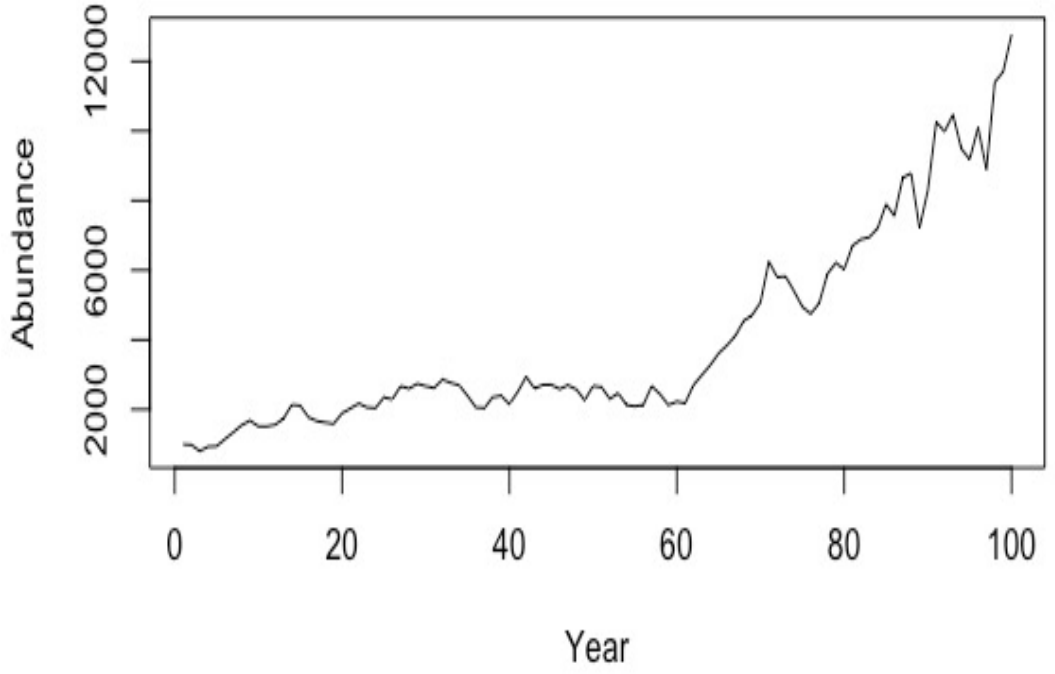
#Let's use five parameters to do so now:

- nt: the number of years,
- grow: the population growth rate,
- sd_grow: the amount of annual variability in the growth rate
- U: the annual exploitation rate
- plot: whether you wish to have a plot created.

```
pop_sim = function(nt, grow, sd_grow, U, plot = F) {
  N = NULL
  N[1] = 1000
  for (t in 2:nt) {
  N[t] = (N[t-1] * grow * rlnorm(1, 0, sd_grow)) * (1 - U)
}
   if (plot) { plot(N, type = "l", pch = 15, xlab  = "Year",
ylab = "Abundance")
}
N
}
```

Run: **pop_sim**(100, 1.1, 0.1, 0.08, T) to get

- [1] 1000.0000 982.3888 802.9221 930.8944 942.8799 1147.2425 1343.0696
- [8] 1547.2829 1679.2181 1514.6867 1513.1179 1560.9256 1736.7056 2135.8081
- [15] 2106.6725 1775.4615 1665.7489 1623.7020 1589.0171 1889.1755 2029.1288
- [22] 2170.6199 2058.1873 2038.3532 2347.5983 2290.1806 2671.5877 2598.8134
- [29] 2738.5065 2669.0003 2617.4264 2859.6799 2764.8132 2694.8130 2388.6001
- [36] 2057.0187 2041.2244 2351.3923 2395.7745 2151.1563 2509.3455 2943.5983
- [43] 2599.5925 2706.5242 2710.5283 2587.0943 2696.7068 2573.2741 2267.4747
- [50] 2676.7501 2638.4771 2306.5914 2464.6563 2126.1586 2090.3945 2131.9059
- [57] 2676.4949 2435.6190 2128.2608 2225.5276 2179.7877 2706.6805 2989.4001
- [64] 3277.0129 3609.7139 3843.7520 4117.0917 4546.7481 4706.4806 5077.8774
- [71] 6248.7845 5797.8300 5824.2902 5400.6019 4948.3756 4747.5507 5046.0663
- [78] 5894.9432 6207.3198 6030.4074 6706.5260 6884.1739 6946.1890 7204.8305
- [85] 7895.0993 7563.0521 8655.1318 8783.0285 7210.3333 8300.1920 10254.6761
- [92] 9983.9319 10467.9362 9487.7283 9186.1128 10096.7386 8892.1724 11403.9986
- [99] 11699.4072 12772.6927



Replicating the simulation:

#Replicate the simulation for 1000 times

- out = replicate(n = 1000, expr = pop_sim(100, 1.1, 0.1, 0.08, F))

- out = large matrix (10000 elements, 800.2 kb)

#View this matrix in R Studio:

- View(out)

Summarization of simulation:

- After replicating a calculation many times, we will need to summarize the results.
- We must show the central tendency and variability
- We can also show Frequencies and cross-tabulations

#Central Tendency: mean

- N_mean = apply(out, 1, mean)
- N_mean[1:10]

#Variability:

N_sd = apply(out, 1, sd)

N_sd[1:10]

Summarization of simulation:

#Frequencies 1

out10 = **ifelse**(out[10,] < 1000, "less10", "greater10")

table(out10)

#Frequencies 2

out20 = **ifelse**(out[20,] < 1100, "less20", "greater20")
table(out20)

#Cross-tabulations

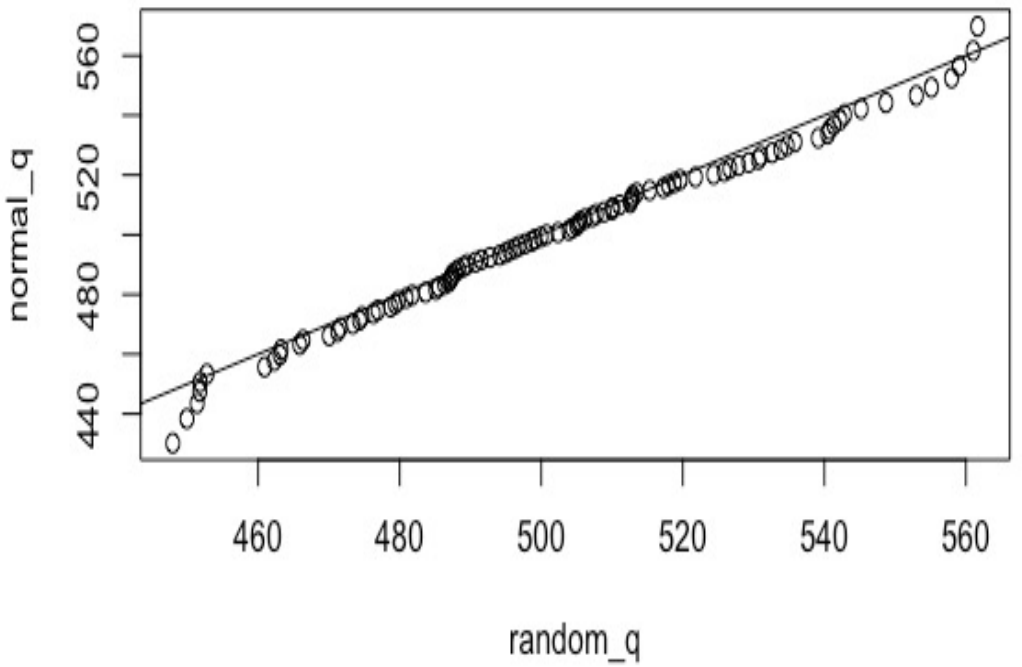
- table(out10, out20)

#Cross-tabulations with probabilities

- **round**(**table**(out10, out20)/1000, 2)

Simulation Based Learning: Example 1

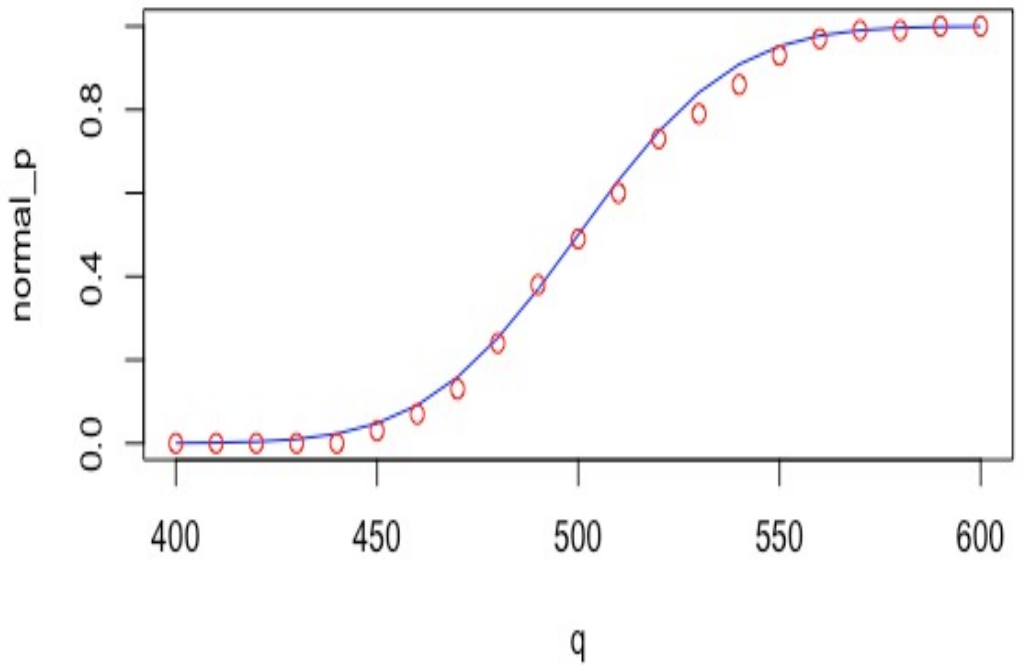
- mu = 500; sig = 30
- random = **rnorm**(100, mu, sig)
- p = **seq**(0.01, 0.99, 0.01)
  random_q = **quantile**(random, p)
  normal_q = **qnorm**(p, mu, sig)
- **plot**(normal_q ~ random_q);
  abline(**c**(0,1))



Simulation Based Learning: Example 2

- q = **seq**(400, 600, 10)
- random_cdf = **ecdf**(random)
- random_p = **random_cdf**(q)
- normal_p = **pnorm**(q, mu, sig)

- **plot**(normal_p ~ q, type = "l", col = "blue") **points**(random_p ~ q, col = "red")



Use in Machine learning:

https://machinelearningmastery.com/monte-carlo-sampling-for-probability/

- In machine learning, Monte Carlo methods provide the basis for resampling techniques like the bootstrap method for estimating a quantity, such as the accuracy of a model on a limited dataset.
- We have seen its use in:
  - Resampling algorithms
  - Random hyperparameter tuning (caret package)
  - Ensemble learning algorithms

- Random sampling of model hyperparameters when tuning a model is a Monte Carlo method

- **Ensemble models** used to overcome challenges such as the limited size and noise in a small data sample and the stochastic variance in a learning algorithm **are** all examples of **Monte Carlo methods**.

Question/queries so far?

Class imbalance problem: Binary dep. var. (y)

- It happens in the classification problems
- When we have a categorical binary dependent variable then distribution of 1 and 0 may not be equal (or very skewed)
- When it is very skewed then it is known as “class imbalance”
- We can deal with it using statistics or data science

- Statistical approach
  - Instead of binary logistic regression
    - Use exact logistic regression
    - Use Poisson regression
    - Use zero-inflated Poisson regression
    - Use negative binomial regression
- Data science approach
  - Generate new data using simulations, make the balanced class and get accuracy measures

Class imbalance problem: Categorical “y”

- It happens in the classification problems
- When we have a categorical dependent variable then distribution of 0, 1 or 2 may not be equal (or very skewed)
- When it is very skewed then it is known as “class imbalance”
- We can deal with it using statistics or data science
  - Statistical approach
    - Instead of multinominal or ordinal logistic regression
      - Use exact multinomial/ordinal logistic regression
      - Use Poisson regression
      - Use zero-inflated Poisson regression
      - Use negative binomial regression
  - Data science approach
    - Generate new data using simulations, make the balanced class and get accuracy measures

In statistics, we are more concerned with “Simpson’s Paradox” than the Class Imbalance problems!
UCLA Admission “paradox”: Overall **few females were admitted** but **more females were admitted** when the same data was analyzed by departments! **Same can happen with all the supervised models**!

Example: binary.csv data

#Admission to UCLA

- Four variables in the data
- admit = Admitted or not
- gre = GRE score
- gpa = GPA score
- rank = Rank of the institute where they got their GPA

#Class imbalance problem data

```
data <- read.csv("binary.csv", header = T)
str(data)
summary(data)
```

#Change the admit as factor variable

```
data$admit <- as.factor(data$admit)
summary(data)
```

Outputs:

admit	gre	gpa	rank
Min. :0.0000	Min. :220.0	Min. :2.260	Min. :1.000
1st Qu.:0.0000	1st Qu.:520.0	1st Qu.:3.130	1st Qu.:2.000
Median :0.0000	Median :580.0	Median :3.395	Median :2.000
Mean :0.3175	Mean :587.7	Mean :3.390	Mean :2.485
3rd Qu.:1.0000	3rd Qu.:660.0	3rd Qu.:3.670	3rd Qu.:3.000
Max. :1.0000	Max. :800.0	Max. :4.000	Max. :4.000

- prop. table(table(data$admit))

0	**1**
0.6825	0.3175

admit	gre	gpa	rank
0:273	Min. :220.0	Min. :2.260	Min. :1.000
1:127	1st Qu.:520.0	1st Qu.:3.130	1st Qu.:2.000
	Median :580.0	Median :3.395	Median :2.000
	Mean :587.7	Mean :3.390	Mean :2.485
	3rd Qu.:660.0	3rd Qu.:3.670	3rd Qu.:3.000
	Max. :800.0	Max. :4.000	Max. :4.000

Class imbalance as dependent variable “admit” has 273 (68.25%) cases in 0 (not admitted category) and 127 (31.75%) in 1 (admitted) category.

In statistics, we deal it using different methods but in data science we deal it with making these classes “balanced”

Let’s predict without correcting imbalance:

#Data partition

#set.seed(1234)

- Ind <- sample(2, nrow(data), replace=T, prob=c(0.7,0.3))

- train <- data[ind==1,]

- test <- data[ind==2,]

- #Check the imbalance in the train data
- table(train$admit)
- 0 1
- 196 83
- prop.table(table(train$admit))
- 0 1
- 0.702509 0.297491 (**Is this really imbalance!**)

Let’s predict without correcting imbalance:

#Prediction model

#Random forest model

```
library(randomForest)
rfm.train <- randomForest(admit~., data=train)
```

#Model evaluation with test data using caret package

```
library(caret)
confusionMatrix(predict(rfm.train, test), test$admit, positive = '1')
```

- #Outputs

	Reference	
Prediction	0	1
0	**73**	32
1	4	**12**

Accuracy : 0.7025 (**misleading!**)

95% CI : (0.6126, 0.7821)

Sensitivity : 0.27273 (not good for 1)

Specificity : 0.94805 (good for 0)

This is due to “class imbalance” problem!

Let’s predict with correction: Oversampling

#Correcting class imbalance by oversampling: Using Randomly Oversampling Examples (ROSE) package

- library(ROSE)

- over.samp <- ovun.sample(admit~., data = train, method = "over", N = **196***2)$data

- table(over.samp$admit)

- Here 196 is used as there was an imbalance in the train data
- table(train$admit)
- 0 1
- **196** 83

#We will get equal values now:

- table(over.samp$admit)
- 0 1
- **196** **196**

#Resampling of observed values of category=1 is used to get more 1s!

Let’s predict with correction: Oversampling

#Check summary for changes in the other variables too!

- summary(over.samp)

#Random Forrest model with over sampled data

- rfm.os <- randomForest(admit~., data=over.samp)
- confusionMatrix(predict(rfm.os, test), test$admit, positive = '1')

	Reference	
Prediction	0	1
0	59	22
1	18	22

Accuracy : 0.6694

95% CI : (0.5781, 0.7522)

Sensitivity : 0.5000

Specificity : 0.7662

Sensitivity improved (**good if we wanted to improve prediction for 1**) but overall accuracy decreased!

What else can be done with ROSE package?

- We can do the undersampling and check the model accuracy, sensitivity and specificity again
- We can do both i.e. oversampling and undersamplign and check the model accuracy, sensitivity and specificity again
- We can create a synthetic data, fit the model, predict it to check the model accuracy, sensitivity, specificity etc.
- While creating synthetic data, we must use random seed too in the function to get replicable results!

More on Synthetic Minority Oversampling (SMOTE) here: https://www.youtube.com/watch?v=dkXB8HH_4-k

Missing values:

https://towardsdatascience.com/7-ways-to-handle-missing-values-in-machine-learning-1a6326adf79e

- The real-world data often has a lot of missing values. The cause of missing values can be data corruption or failure to record data.
- **The handling of missing data is very important during the preprocessing of the dataset as many machine learning algorithms do not support missing values.**
- Visit the link to learn more about handling missing values to learn:

- The 7 ways to handle missing values in the dataset
  - Deleting Rows with missing values
  - Impute missing values for continuous variable (mean, median etc.)
  - Impute missing values for categorical variable (predict the categories)
  - Other Imputation Methods
  - Using Algorithms that support missing values
  - Prediction of missing values
  - Imputation using Deep Learning Library — Datawig

Missing values checking and handling in R:

#Check missing values in R

- *colsum(is.na(data frame))*
- *sum(is.na(data frame$column name)*

#Strategies

- List-wise deletion
- Pair-wise deletion
- Mean/ Mode/ Median Imputation
  - Generalized Imputation
  - Similar case Imputation
- Prediction Model
- KNN Imputation

#List of R Packages

- **MICE**
- Amelia
- missForest
- Hmisc
- mi
- etc.

https://medium.com/coinmonks/dealing-with-missing-data-using-r-3ae428da2d17

https://www.analyticsvidhya.com/blog/2016/03/tutorial-powerful-packages-imputing-missing-values/

MICE package:

- MICE (Multivariate Imputation via Chained Equations) is one of the commonly used package by R users. Creating multiple imputations as compared to a single imputation (such as mean) takes care of uncertainty in missing values.
- MICE assumes that the missing data are **Missing at Random (MAR)**, which means that the probability that a value is missing depends only on observed value and can be predicted using them.

More here: https://medium.com/coinmonks/dealing-with-missing-data-using-r-3ae428da2d17

- It imputes data on a variable by variable basis by specifying an imputation model per variable.
- The methods used by this package are:
  - PMM (Predictive Mean Matching) — For numeric variables
  - logreg(Logistic Regression) — For Binary Variables(with 2 levels)
  - polyreg(Bayesian polytomous regression) — For Factor Variables (>= 2 levels)
  - Proportional odds model (**ordered and censored variables**, >= 2 levels)

Use of MICE with an example data is here: https://www.youtube.com/watch?v=An7nPLJ0fsg

Question/queries?

- Next classes

- Communicating the results of data science projects
- Defining projects in R studio
  - Local file/folder
  - GitHub repository
- R notebook

Thank you!

@shitalbhandary