Creating New Data

Author: Nagdev Amruthnath

Most of us have come across situations where we have not enough data for building reliable models due to various reasons such as its expensive to collect data (human studies), limited resources, lack of historical data availability (earth quakes). Even before we begin talking about how to overcome the challange, let me first talk about why we need minimum samples even before we consider building model. First of all we can build a model with low samples. It is defnitly possible! But, the as the number of samples decreases, the margin of error increases and vice versa. If you want to build a model with the highest accuracy you would need a as many samples as possible. There is a formula that can be used to calculate the sample size and is as follows

$$\mu \text{ is } n \geq \left(\frac{z * \sigma}{\text{MOE}}\right)^2;$$

Where, n = sample size Z = Z-score value σ = populated standard deviation MOE = acceptable margin of error

You can also calulated with an online calculator as in this link https://www.qualtrics.com/blog/calculating-sample-size/ (https://www.qualtrics.com/blog/calculating-sample-size/)

Now we know that why minimum samples are required for achieveing required accuracy, say in same case we do not have an opportunity to collect more samples or available. They we have an option to do the following

1. K-fold cross validation
2. Leave-P-out cross validation
3. Leave-one-out cross validation
4. New data creation through estimation

In K-fold method, the data is split into k partitions and then is trained with each partition and tested with the leftout kth partition. In k-hold method, not all combinations are considered. Only user specified partions are considered. While in leave-one/p-out, all combinations or partitions are cosidered. This is more exhasustive technique in validating the results. The following above two techniques are the most popular techniques that is used both in machine learning and deep learning.

In new data creation through estimation technique, rows of missing data is created in the data set and a seperate data imputation model is used to impute missing data in the rows. Multivariate Imputation by Chained Equations (MICE) is one of the most popular algorthims that are available to insert missing data irrespective of data types such as mixes of continuous, binary, unordered categorical and ordered categorical data.

There are various tutorials available for k-fold and leave one out models. This tutorial will focus on the fourth model where new data will be created to handle less sample size. In the and a simple classification model with be trained to see if there was a significant imporvement. Also, distribution of imputed and non-imputed data will be compared to see any signifcant difference.

In [1]:
```
options(warn=-1)

# load libraies
library(mice)
library(dplyr)
```

```
Loading required package: lattice

Attaching package: 'mice'

The following objects are masked from 'package:base':

    cbind, rbind


Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Load data into a dataframe

The data availabe in the repository is used for the analysis

In [2]:
```
setwd("C:/Users/aanamruthn/Documents/Jupyter/Testing/RNotebooksTesting/OpenSou
rceWork/Experiment")
#read csv files
file1 = read.csv("dry run.csv", sep=",", header =T)
file2 = read.csv("base.csv", sep=",", header =T)
file3 = read.csv("imbalance 1.csv", sep=",", header =T)
file4 = read.csv("imbalance 2.csv", sep=",", header =T)

#Add labels to data
file1$y = 1
file2$y = 2
file3$y = 3
file4$y = 4

#view top rows of data
head(file1)
```

time	ax	ay	az	aT	y
0.002	-0.3246	0.2748	0.1502	0.451	1
0.009	0.6020	-0.1900	-0.3227	0.709	1
0.019	0.9787	0.3258	0.0124	1.032	1
0.027	0.6141	-0.4179	0.0471	0.744	1
0.038	-0.3218	-0.6389	-0.4259	0.833	1
0.047	-0.3607	0.1332	-0.1291	0.406	1

Create some features from data

the data used in this study is vibration data with different states. The data was collected at 100Hz. The data to be used as is is high dimentional also, we do not have any good summary of the data. Hence, some statistical features are extracted. In this case, sample standard deviation, sample mean, sample min, sample max and sample median is calculated. Also, the data is aggrigated by 1 second.

In [3]:
```
file1$group = as.factor(round(file1$time))
file2$group = as.factor(round(file2$time))
file3$group = as.factor(round(file3$time))
file4$group = as.factor(round(file4$time))
#(file1,20)

#list of all files
files = list(file1, file2, file3, file4)

#loop through all files and combine
features = NULL
for (i in 1:4){
res = files[[i]] %>%
    group_by(group) %>%
    summarize(ax_mean = mean(ax),
              ax_sd = sd(ax),
              ax_min = min(ax),
              ax_max = max(ax),
              ax_median = median(ax),
              ay_mean = mean(ay),
              ay_sd = sd(ay),
              ay_min = min(ay),
              ay_may = max(ay),
              ay_median = median(ay),
              az_mean = mean(az),
              az_sd = sd(az),
              az_min = min(az),
              az_maz = max(az),
              az_median = median(az),
              aT_mean = mean(aT),
              aT_sd = sd(aT),
              aT_min = min(aT),
              aT_maT = max(aT),
              aT_median = median(aT),
              y = mean(y)
             )
    features = rbind(features, res)
}

features = subset(features, select = -group)

# store it in a df for future reference
actual.features = features
```

Study data

First, lets look at the size of our populations and summary of our features along with thier data types.

In [4]:
```
nrow(features)
str(features)
```

```
362
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':       362 obs. of  21 variables:
 $ ax_mean  : num  -0.03816 -0.00581 0.06985 0.01155 0.04669 ...
 $ ax_sd    : num  0.659 0.633 0.667 0.551 0.643 ...
 $ ax_min   : num  -1.26 -1.62 -1.46 -1.93 -1.78 ...
 $ ax_max   : num  1.38 1.19 1.47 1.2 1.48 ...
 $ ax_median: num  -0.0955 -0.0015 0.107 0.0675 0.0836 ...
 $ ay_mean  : num  -0.068263 0.003791 0.074433 0.000826 -0.017759 ...
 $ ay_sd    : num  0.751 0.782 0.802 0.789 0.751 ...
 $ ay_min   : num  -1.39 -1.56 -1.48 -2 -1.66 ...
 $ ay_may   : num  1.64 1.54 1.8 1.56 1.44 ...
 $ ay_median: num  -0.19 0.0101 0.1186 -0.0027 -0.0253 ...
 $ az_mean  : num  -0.138 -0.205 -0.0641 -0.0929 -0.1399 ...
 $ az_sd    : num  0.985 0.925 0.929 0.889 0.927 ...
 $ az_min   : num  -2.68 -3.08 -1.82 -2.16 -1.85 ...
 $ az_maz   : num  2.75 2.72 2.49 3.24 3.55 ...
 $ az_median: num  0.0254 -0.2121 -0.1512 -0.1672 -0.1741 ...
 $ aT_mean  : num  1.27 1.26 1.3 1.2 1.23 ...
 $ aT_sd    : num  0.583 0.545 0.513 0.513 0.582 ...
 $ aT_min   : num  0.4 0.41 0.255 0.393 0.313 0.336 0.275 0.196 0.032 0.358
...
 $ aT_maT   : num  3.03 3.2 2.64 3.32 3.6 ...
 $ aT_median: num  1.08 1.14 1.28 1.12 1.17 ...
 $ y        : num  1 1 1 1 1 1 1 1 1 ...
```

Create observations with NA values in the end

In [5]:
```
features1 = features
for(i in 363:400){
  features1[i,] = NA
}
```

Look at bottom 50 rows

In [6]:
```
tail(features1, 50)
```

ax_mean	ax_sd	ax_min	ax_max	ax_median	ay_mean	ay_sd	ay_min	a
-0.016097030	0.8938523	-2.3445	2.3006	-0.07360	-0.009759406	1.311817	-3.4215	2
-0.015565347	0.8956615	-2.2661	2.5089	0.08640	0.027313861	1.294063	-2.9421	2
0.024006250	0.8653758	-2.4099	2.5328	-0.03170	0.008440625	1.376398	-3.0422	2
-0.015563000	0.8720967	-2.3451	2.3269	-0.05325	0.013962000	1.240091	-3.1360	2
0.003894898	0.8806773	-2.3098	3.1902	-0.09260	0.022575510	1.301955	-3.2561	2
-0.039379208	0.8127135	-2.1523	1.8828	-0.11250	0.005454455	1.189519	-2.8057	2
0.021469000	0.8272527	-1.5895	3.7505	-0.08995	0.011312000	1.285206	-2.7423	2
0.005917000	0.9139808	-2.3310	2.8131	-0.07800	-0.040868000	1.320873	-2.9778	2
-0.034448571	0.8640626	-2.4917	2.4113	-0.01960	-0.013410476	1.235196	-3.3305	2
0.046837374	0.9776022	-1.8688	2.6644	-0.03600	0.019817172	1.293644	-2.7836	2
-0.014453061	0.9553743	-2.7118	2.4640	-0.01000	-0.037717347	1.285358	-3.1225	2
0.046810870	0.9259427	-1.5309	1.9420	-0.11455	0.230676087	1.491983	-2.8435	2
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N

ax_mean	ax_sd	ax_min	ax_max	ax_median	ay_mean	ay_sd	ay_min	a
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N
NA	NA	NA	NA	NA	NA	NA	NA	N

Impute NA's with best values using iteration method

In [7]:
```
imputed_Data = mice(features1,
                    m=1,
                    maxit = 50,
                    method = 'pmm',
                    seed = 999,
                    printFlag =FALSE)
```

View imputed results

In [8]:
```
imputedResultData = mice::complete(imputed_Data,1)
tail(imputedResultData, 50)
```

	ax_mean	ax_sd	ax_min	ax_max	ax_median	ay_mean	ay_sd	ay_
351	-0.016097030	0.8938523	-2.3445	2.3006	-0.07360	-0.009759406	1.3118166	-3.4
352	-0.015565347	0.8956615	-2.2661	2.5089	0.08640	0.027313861	1.2940627	-2.9
353	0.024006250	0.8653758	-2.4099	2.5328	-0.03170	0.008440625	1.3763983	-3.0
354	-0.015563000	0.8720967	-2.3451	2.3269	-0.05325	0.013962000	1.2400913	-3.1
355	0.003894898	0.8806773	-2.3098	3.1902	-0.09260	0.022575510	1.3019546	-3.2
356	-0.039379208	0.8127135	-2.1523	1.8828	-0.11250	0.005454455	1.1895194	-2.8
357	0.021469000	0.8272527	-1.5895	3.7505	-0.08995	0.011312000	1.2852056	-2.7
358	0.005917000	0.9139808	-2.3310	2.8131	-0.07800	-0.040868000	1.3208731	-2.9
359	-0.034448571	0.8640626	-2.4917	2.4113	-0.01960	-0.013410476	1.2351957	-3.3
360	0.046837374	0.9776022	-1.8688	2.6644	-0.03600	0.019817172	1.2936436	-2.7
361	-0.014453061	0.9553743	-2.7118	2.4640	-0.01000	-0.037717347	1.2853576	-3.1
362	0.046810870	0.9259427	-1.5309	1.9420	-0.11455	0.230676087	1.4919834	-2.8
363	0.011238614	0.8127502	-1.9602	2.1430	0.00680	-0.013367308	1.3019546	-3.0
364	-0.009812264	0.7680463	-2.3492	1.3919	0.03110	0.013984158	0.6084791	-1.4
365	-0.026760000	0.4780558	-1.1826	0.9934	0.05560	-0.035218269	0.5632648	-1.0
366	0.029083000	0.7515921	-2.2628	2.4640	-0.00820	0.011159596	1.3073606	-3.1
367	0.002401000	0.5641062	-1.1533	1.4479	-0.04215	0.011159596	1.0358946	-1.9
368	0.017670707	0.4158231	-0.9785	1.0647	0.07680	-0.026719608	0.4759174	-0.9
369	-0.078038776	0.4413032	-1.1099	0.9826	-0.03910	-0.010626042	0.4768587	-0.9
370	0.004372632	0.8352791	-1.6966	2.3897	0.00845	-0.010064000	1.2746954	-2.7
371	0.016103000	0.3997476	-0.9537	1.1546	0.03655	-0.031622772	0.4828770	-0.9
372	-0.020355446	0.4178729	-1.0524	0.9076	-0.09340	0.044400000	0.5439558	-0.9
373	0.001363636	0.4868077	-0.9027	1.5155	0.04820	0.031339000	1.0619675	-2.3
374	-0.008122222	0.8831968	-1.9394	3.3244	-0.09610	0.017400971	1.3778757	-3.7
375	-0.065401010	0.8489219	-2.4871	2.1672	-0.11250	-0.043491753	0.5648206	-1.5
376	0.039720000	0.5946125	-1.5250	1.7390	0.05040	0.061424510	0.8133879	-1.2
377	0.022841000	0.8646867	-2.1253	2.6378	0.05720	0.052515306	1.1332836	-2.5
378	-0.001924510	0.5975310	-1.4775	1.4089	-0.11455	-0.040868000	1.0363392	-2.3
379	0.017975000	0.4780750	-1.2011	1.4923	-0.07450	-0.022319802	0.5072372	-1.1
380	-0.070804000	0.4780558	-1.9254	0.9244	-0.05830	-0.074927551	0.5037149	-1.0
381	-0.002204762	0.9310547	-2.7832	2.5242	-0.07875	-0.019305882	1.3019546	-2.4
382	0.021469000	0.8646867	-2.0001	2.4477	-0.03400	0.051977895	1.3628383	-2.6

	ax_mean	ax_sd	ax_min	ax_max	ax_median	ay_mean	ay_sd	ay_
383	-0.015468354	0.8127502	-2.2034	2.3405	-0.02150	0.046179798	1.3628383	-2.8
384	-0.002143000	0.4442709	-0.9949	1.0734	-0.04265	-0.007904000	0.5386439	-1.2
385	0.027587129	0.4551125	-1.2785	1.0285	0.05660	-0.035263725	0.4854652	-1.0
386	0.017670707	0.6981887	-1.5387	2.1808	-0.04500	0.043603191	1.2152972	-2.6
387	0.017401000	0.7680463	-1.4528	2.2822	-0.00350	0.055612871	1.0989870	-2.7
388	0.001363636	0.4354711	-1.0677	0.9579	0.03655	-0.017115842	0.5501718	-1.1
389	0.036087000	0.8741671	-2.2967	3.3393	-0.03330	-0.019919792	1.4065464	-2.9
390	0.007588000	0.8409728	-1.9602	2.2383	-0.07985	0.025797000	1.3525870	-3.1
391	0.065754545	0.4533416	-0.7769	1.1179	0.10470	0.047955446	0.5539467	-0.9
392	-0.030526733	0.4442709	-1.7119	1.0302	0.03000	-0.021866667	0.6103892	-1.0
393	-0.001643000	0.8086920	-1.9033	2.5242	-0.03200	-0.033747959	1.3111909	-3.0
394	-0.023916346	0.4139117	-0.6977	1.1179	-0.04360	0.011312000	0.4828770	-1.2
395	0.037914706	0.4369138	-0.9701	0.9937	0.07080	-0.011703810	0.4883374	-1.0
396	-0.024820792	0.8127135	-1.9299	2.6378	0.01800	-0.044580000	1.1363141	-2.5
397	-0.016237500	0.7620745	-2.4099	1.7855	-0.05150	0.032355102	1.1534694	-2.6
398	-0.039379208	0.5614528	-1.7119	1.4600	-0.11620	-0.032463000	1.1096189	-2.4
399	0.026206186	0.7980083	-1.9033	2.3863	0.00210	0.009870874	1.2557210	-2.8
400	0.072777778	0.4051881	-0.8386	0.8847	0.15575	0.015370408	0.4759174	-0.9

Looking at distribution actual data and imputed data

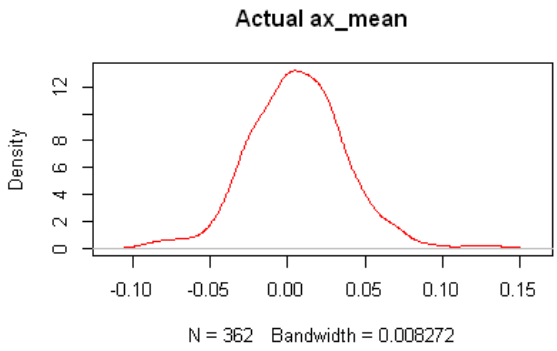
We will first compare basic statistics and then distributions of the couple of features. In the comparison of statistics between actual and imputed we can observe that the mean and SD for both imputed and actual are almost equal.

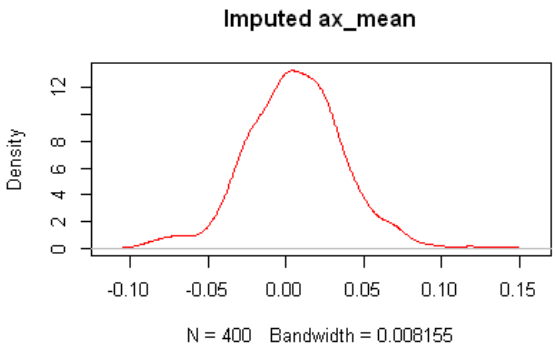
In [9]:
```
data.frame(actual_ax_mean = c(mean(features$ax_mean), sd(features$ax_mean))
           , imputed_ax_mean = c(mean(imputedResultData$ax_mean), sd(imputedRe
sultData$ax_mean))
           , actual_ax_median = c(mean(features$ax_median), sd(features$ax_med
ian))
           , imputed_ax_median = c(mean(imputedResultData$ax_median), sd(imput
edResultData$ax_median))
           , actual_az_sd = c(mean(features$az_sd), sd(features$az_sd))
           , imputed_az_sd = c(mean(imputedResultData$az_sd), sd(imputedResult
Data$az_sd))
           , row.names = c("mean", "sd"))
```

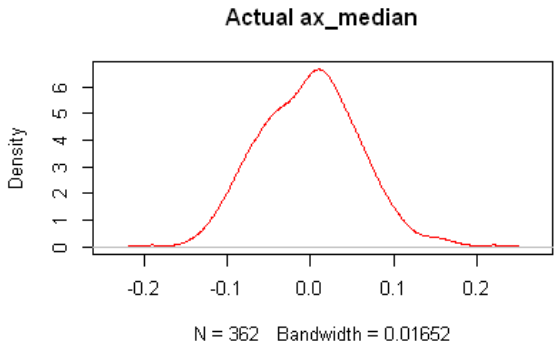
	actual_ax_mean	imputed_ax_mean	actual_ax_median	imputed_ax_median	actu
mean	0.006307909	0.005851233	-0.001328867	-0.00214025	1.058
sd	0.030961085	0.031125848	0.059619834	0.06011342	0.244

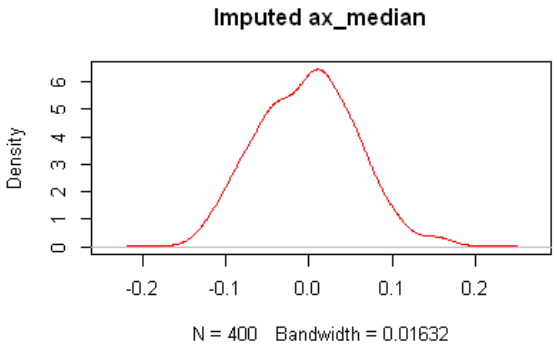
Now, lets look at the distributions in the data. From the distribution below, we can observe that the distributions for actual data and imputed data is almost identical. We can confirm it with the bandwidth in the plots

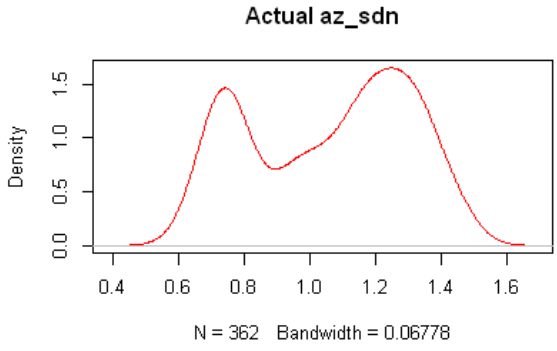
In [10]:
```
par(mfrow=c(3,2))
plot(density(features$ax_mean), main = "Actual ax_mean", type="l", col="red")
plot(density(imputedResultData$ax_mean), main = "Imputed ax_mean", type="l", c
ol="red")
plot(density(features$ax_median), main = "Actual ax_median", type="l", col="re
d")
plot(density(imputedResultData$ax_median), main = "Imputed ax_median", type=
"l", col="red")
plot(density(features$az_sd), main = "Actual az_sdn", type="l", col="red")
plot(density(imputedResultData$az_sd), main = "Imputed az_sd", type="l", col=
"red")
```

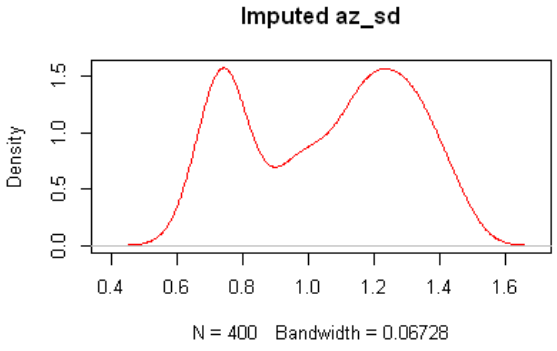












Building a classification model based on actual data and Imputed data

In the following data y will be our classification variable. We will build a classification model using basic support vector machine(SVM) with actual and imputed data. No transformation will be done on the data. In the end we will compare the results

Actual Data

In [11]:
```
#create samples of 80:20 ratio
features$y = as.factor(features$y)
sample = sample(nrow(features) , nrow(features)* 0.8)
train = features[sample,]
test = features[-sample,]
```

Build a SVM model

In [12]:
```
library(e1071)
library(caret)

actual.svm.model = svm(y ~., data = train)
summary(actual.svm.model)
```

```
Loading required package: ggplot2

Call:
svm(formula = y ~ ., data = train)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.05

Number of Support Vectors:  142

 ( 47 18 47 30 )


Number of Classes:  4

Levels:
 1 2 3 4
```

Validate SVM model

In the below confusion matrix, we observe the following

1. accuary>NIR indicating model is very good
2. Higher accuray and kappa value indicates a very accurate model
3. Even the balanced accuracy is close to 1 indicating the model is highly accurate

In [13]:
```
confusionMatrix(predict(actual.svm.model, test), test$y)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  1  2  3  4
         1 10  1  0  0
         2  0 26  0  0
         3  0  0 22  0
         4  0  0  3 11

Overall Statistics

               Accuracy : 0.9452
                 95% CI : (0.8656, 0.9849)
    No Information Rate : 0.3699
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9234
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: 1 Class: 2 Class: 3 Class: 4
Sensitivity            1.0000   0.9630   0.8800   1.0000
Specificity            0.9841   1.0000   1.0000   0.9516
Pos Pred Value         0.9091   1.0000   1.0000   0.7857
Neg Pred Value         1.0000   0.9787   0.9412   1.0000
Prevalence             0.1370   0.3699   0.3425   0.1507
Detection Rate         0.1370   0.3562   0.3014   0.1507
Detection Prevalence   0.1507   0.3562   0.3014   0.1918
Balanced Accuracy      0.9921   0.9815   0.9400   0.9758
```

Imputed Data

Sample data creation

In [14]:
```
#create samples of 80:20 ratio
imputedResultData$y = as.factor(imputedResultData$y)
sample = sample(nrow(imputedResultData) , nrow(imputedResultData)* 0.8)
train = imputedResultData[sample,]
test = imputedResultData[-sample,]
```

Build a SVM model

In [15]:
```
imputed.svm.model = svm(y ~., data = train)
summary(imputed.svm.model)
```

```
Call:
svm(formula = y ~ ., data = train)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.05

Number of Support Vectors:  167

 ( 59 47 36 25 )


Number of Classes:  4

Levels:
 1 2 3 4
```

Validate SVM model

In the below confusion matrix, we observe the following

1. accuary>NIR indicating model is very good
2. Higher accuray and kappa value indicates a very accurate model
3. Even the balanced accuracy is close to 1 indicating the model is highly accurate

In [16]:
```
confusionMatrix(predict(imputed.svm.model, test), test$y)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  1  2  3  4
         1 15  0  0  0
         2  1 21  0  0
         3  0  0 17  0
         4  0  0  0 26

Overall Statistics

               Accuracy : 0.9875
                 95% CI : (0.9323, 0.9997)
    No Information Rate : 0.325
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9831
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: 1 Class: 2 Class: 3 Class: 4
Sensitivity            0.9375   1.0000   1.0000    1.000
Specificity            1.0000   0.9831   1.0000    1.000
Pos Pred Value         1.0000   0.9545   1.0000    1.000
Neg Pred Value         0.9846   1.0000   1.0000    1.000
Prevalence             0.2000   0.2625   0.2125    0.325
Detection Rate         0.1875   0.2625   0.2125    0.325
Detection Prevalence   0.1875   0.2750   0.2125    0.325
Balanced Accuracy      0.9688   0.9915   1.0000    1.000
```

Overall results

What we saw above and their interpretation is completly subjective. One way to truly validate them is to create random train and test samples multiple times (say 30), build a model, validate the model, capture kappa value. Finally use a simple t-test to see if there is a significant difference.

Null hypotheisis:
H0: there is no significant difference between two samples

In [17]:
```
# lets create functions to simplify the process

test.function = function(data){
    # create samples
    sample = sample(nrow(data) , nrow(data)* 0.75)
    train = data[sample,]
    test = data[-sample,]

    # build model
    svm.model = svm(y ~., data = train)

    # get metrics
    metrics = confusionMatrix(predict(svm.model, test), test$y)
    return(metrics$overall['Accuracy'])

}
```

In [18]:
```
# now lets calculate accuracy with actual data to get 30 results
actual.results  = NULL
for(i in 1:100) {
    actual.results[i] = test.function(features)
}
head(actual.results)
```

```
0.978021978021978  0.978021978021978  0.978021978021978
0.945054945054945  0.989010989010989  0.967032967032967
```

In [19]:
```
# now lets calculate accuracy with imputed data to get 30 results
imputed.results  = NULL
for(i in 1:100) {
    imputed.results[i] = test.function(imputedResultData)
}
head(imputed.results)
```

```
0.97  0.95  0.92  0.96  0.92  0.96
```

T-test to test the results

In [20]:
```
# Do a simple t-test to see if there is a difference in accuracy when data is
 imputed
t.test(x= actual.results, y = imputed.results, conf.level = 0.95)
```

```
        Welch Two Sample t-test

data:  actual.results and imputed.results
t = 7.9834, df = 194.03, p-value = 1.222e-13
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01673213 0.02771182
sample estimates:
mean of x mean of y
 0.968022  0.945800
```

In the above t-test we have set the confidence level at 95%. From the results we can observe that the p-value is less than 0.05 indicating that there is a significant difference in accuracy between actual data and imputed data. From the means we can notics that the average accuracy of actual data is about 96.5% while the accuracy of imputed data y is about 92.5%. There is a variation of 4%. So, does that mean imputing more data results in reducing the accuracy across various models?

Why not do a test to compare the results? let's consider 4 other models for that and those will be

1. Random forest
2. Decision tree
3. KNN
4. Naive Bayes

Random Forest

In [21]:
```
library(randomForest)

# lets create functions to simplify the process

test.rf.function = function(data){
    # create samples
    sample = sample(nrow(data) , nrow(data)* 0.75)
    train = data[sample,]
    test = data[-sample,]

    # build model
    rf.model = randomForest(y ~., data = train)

    # get metrics
    metrics = confusionMatrix(predict(rf.model, test), test$y)
    return(metrics$overall['Accuracy'])

}

# now lets calculate accuracy with actual data to get 30 results
actual.rf.results  = NULL
for(i in 1:100) {
    actual.rf.results[i] = test.rf.function(features)
}
#head(actual.rf.results)

# now lets calculate accuracy with imputed data to get 30 results
imputed.rf.results  = NULL
for(i in 1:100) {
    imputed.rf.results[i] = test.rf.function(imputedResultData)
}
head(data.frame(Actual = actual.rf.results, Imputed = imputed.rf.results))

# Do a simple t-test to see if there is a difference in accuracy when data is
 imputed
t.test(x= actual.rf.results, y = imputed.rf.results, conf.level = 0.95)
```

```
randomForest 4.6-14
Type rfNews() to see new features/changes/bug fixes.

Attaching package: 'randomForest'

The following object is masked from 'package:ggplot2':

    margin

The following object is masked from 'package:dplyr':

    combine
```

Actual	Imputed
0.956044	0.95
1.000000	0.93
0.967033	0.96
0.967033	0.96
1.000000	0.97
0.967033	0.93

```
        Welch Two Sample t-test

data:  actual.rf.results and imputed.rf.results
t = 11.734, df = 183.2, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.02183138 0.03065654
sample estimates:
mean of x mean of y
 0.976044  0.949800
```

Decision Tree

In [22]:
```
library(rpart)

# lets create functions to simplify the process

test.dt.function = function(data){
    # create samples
    sample = sample(nrow(data) , nrow(data)* 0.75)
    train = data[sample,]
    test = data[-sample,]

    # build model
    dt.model = rpart(y ~., data = train, method="class")

    # get metrics
    metrics = confusionMatrix(predict(dt.model, test, type="class"), test$y)
    return(metrics$overall['Accuracy'])

}

# now lets calculate accuracy with actual data to get 30 results
actual.dt.results  = NULL
for(i in 1:100) {
    actual.dt.results[i] = test.dt.function(features)
}
#head(actual.rf.results)

# now lets calculate accuracy with imputed data to get 30 results
imputed.dt.results  = NULL
for(i in 1:100) {
    imputed.dt.results[i] = test.dt.function(imputedResultData)
}
head(data.frame(Actual = actual.dt.results, Imputed = imputed.dt.results))

# Do a simple t-test to see if there is a difference in accuracy when data is
 imputed
t.test(x= actual.dt.results, y = imputed.dt.results, conf.level = 0.95)
```

Actual	Imputed
0.978022	0.92
0.967033	0.94
0.967033	0.95
0.956044	0.94
0.956044	0.94
0.978022	0.95

```
        Welch Two Sample t-test

data:  actual.dt.results and imputed.dt.results
t = 16.24, df = 167.94, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.03331888 0.04254046
sample estimates:
mean of x mean of y
0.9703297 0.9324000
```

K-Nearest Neighbor (KNN)

In [23]:
```
library(class)

# lets create functions to simplify the process

test.knn.function = function(data){
    # create samples
    sample = sample(nrow(data) , nrow(data)* 0.75)
    train = data[sample,]
    test = data[-sample,]

    # build model
    knn.model = knn(train,test, cl=train$y, k=5)

    # get metrics
    metrics = confusionMatrix(knn.model, test$y)
    return(metrics$overall['Accuracy'])

}

# now lets calculate accuracy with actual data to get 30 results
actual.dt.results  = NULL
for(i in 1:100) {
    actual.dt.results[i] = test.knn.function(features)
}
#head(actual.rf.results)

# now lets calculate accuracy with imputed data to get 30 results
imputed.dt.results  = NULL
for(i in 1:100) {
    imputed.dt.results[i] = test.knn.function(imputedResultData)
}
head(data.frame(Actual = actual.dt.results, Imputed = imputed.dt.results))

# Do a simple t-test to see if there is a difference in accuracy when data is
 imputed
t.test(x= actual.dt.results, y = imputed.dt.results, conf.level = 0.95)
```

Actual	Imputed
0.967033	0.97
1.000000	0.98
0.978022	0.99
0.978022	1.00
0.967033	1.00
0.978022	1.00

```
        Welch Two Sample t-test

data:  actual.dt.results and imputed.dt.results
t = 3.2151, df = 166.45, p-value = 0.001566
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.002126868 0.008895110
sample estimates:
mean of x mean of y
 0.989011  0.983500
```

Naive Bayes

In [24]:
```
# lets create functions to simplify the process

test.nb.function = function(data){
    # create samples
    sample = sample(nrow(data) , nrow(data)* 0.75)
    train = data[sample,]
    test = data[-sample,]

    # build model
    nb.model = naiveBayes(y ~., data = train)

    # get metrics
    metrics = confusionMatrix(predict(nb.model, test), test$y)
    return(metrics$overall['Accuracy'])

}

# now lets calculate accuracy with actual data to get 30 results
actual.nb.results  = NULL
for(i in 1:100) {
    actual.nb.results[i] = test.nb.function(features)
}
#head(actual.rf.results)

# now lets calculate accuracy with imputed data to get 30 results
imputed.nb.results  = NULL
for(i in 1:100) {
    imputed.nb.results[i] = test.nb.function(imputedResultData)
}
head(data.frame(Actual = actual.nb.results, Imputed = imputed.nb.results))

# Do a simple t-test to see if there is a difference in accuracy when data is
 imputed
t.test(x= actual.nb.results, y = imputed.nb.results, conf.level = 0.95)
```

Actual	Imputed
0.989011	0.95
0.967033	0.92
0.978022	0.94
1.000000	0.95
0.989011	0.90
0.967033	0.93

```
        Welch Two Sample t-test

data:  actual.nb.results and imputed.nb.results
t = 18.529, df = 174.88, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.04214191 0.05218996
sample estimates:
mean of x mean of y
0.9740659 0.9269000
```

Conclusion

From the above resuls we observe that irrespective of the type of model built, we observed a standard variation in accuracy in the range of 3% - 5% between using actual data and imputed data. In all the cases, actual data helped in building a better model compared to using imputed data for building the model.