CW2 COMP4030 R Code (ID: 20046542)

1. Importing Libraries & Data

Load the libraries that we will be using:

```
library(tidyverse) # For data manipulation with dplyr and graphing with ggplot2
library(randomForest) # For random forest models and corresponding attributes
library(Boruta) # For the Boruta feature selection algorithm
library(ROSE) # For oversampling techniques
library(pROC) # For ROC curves
library(ROCR) # Additional methods for ROC curves
library(corrplot) # For pairwise correlation plots
library(caret) # For cross validation and splitting datasets
```

First, we specify the working directory and read the Breast Cancer Wisconsin (Original) dataset into R (BCW).

```
setwd("C:/Users/jacob/Documents/Nottingham/Data Modelling and Analysis/CW2/COMP4030_20046542_
Supplementary")

bcw=read.table("bcwo.txt",header = FALSE,sep = ",",dec = ".")
```

The column names are contained in a separate document so we need to import these manually into R.

```
names(bcw)=c("ID", "Clump_Thickness", "CellSize_Uniformity",
             "CellShape_Uniformity", "Marg_Adhesion","CellSize_S_Epi",
             "Bare_Nuclei", "Bland_Chromatin", "Normal_Nucleoli", "Mitoses", "Class")
```

Examine the variables to look for changes to make:

```
str(bcw)
```

```
## 'data.frame':    699 obs. of  11 variables:
##  $ ID                  : int  1000025 1002945 1015425 1016277 1017023 1017122 1018099 1018
561 1033078 1033078 ...
##  $ Clump_Thickness     : int  5 5 3 6 4 8 1 2 2 4 ...
##  $ CellSize_Uniformity : int  1 4 1 8 1 10 1 1 1 2 ...
##  $ CellShape_Uniformity: int  1 4 1 8 1 10 1 2 1 1 ...
##  $ Marg_Adhesion       : int  1 5 1 1 3 8 1 1 1 1 ...
##  $ CellSize_S_Epi      : int  2 7 2 3 2 7 2 2 2 2 ...
##  $ Bare_Nuclei         : chr  "1" "10" "2" "4" ...
##  $ Bland_Chromatin     : int  3 3 3 3 3 9 3 3 1 2 ...
##  $ Normal_Nucleoli     : int  1 2 1 7 1 7 1 1 1 1 ...
##  $ Mitoses             : int  1 1 1 1 1 1 1 1 5 1 ...
##  $ Class               : int  2 2 2 2 2 4 2 2 2 2 ...
```

In the ‘Class’ column, ‘2’ represents ‘Benign’ and ‘4’ represents ‘Malignant’. Let’s make this clearer by using worded names, and also convert the column to a factor to make modelling easier.

```
bcw$Class=ifelse(bcw$Class==2,"Benign","Malignant") # Use words instead of numbers to represe
nt classes
bcw$Class=as.factor(bcw$Class) # Convert class column to a factor
```

2. Dealing With Duplicates

Now let’s analyse different types of potential duplicates:

```
sum(duplicated(bcw$ID)) # 54 'duplicated' ID values
```

```
## [1] 54
```

```
sum(duplicated(bcw)) # 8 'duplicated' entire rows
```

```
## [1] 8
```

```
sum(duplicated(bcw[,-1])) # 236 'duplicated' entire rows (excluding ID)
```

```
## [1] 236
```

At first glance, one would assume ID numbers should be unique, however we see that there are 54 duplicated IDs.

Also, there are only 8 fully duplicated rows. This suggests that 46 of the duplicated IDs have other attributes that do not match the first instance of that ID. This is likely due to the dataset being an aggregate of 8 smaller datasets collected at different times between 1989 and 1991 [1]. The fact that tumours with the same ID may have different attributes suggests that some tumours were observed by Dr Wolberg more that once over the course of these two years, but occasionally the tumour will not have changed much, explaining why there are some complete duplicates.

The raw values of the features have been coded by Dr Wolberg to values between 1 and 10, with 1 being closest to benign and 10 being closest to malignant [1]. This means that slightly different tumours can end up with the same set of coded values despite the raw values not being the same (we see this is the case for 236 samples).

All in all, it seems that tumours with the same ID are observations taken at different points in time. It also seems that tumours with different IDs but the same coded values are similar but not identical and this leads to duplicated coded values.

For this reason, we will not remove any ‘duplicated’ rows (whether duplicated on ID or other attributes), because they represent data from different tumours or data from the same tumour at different times.

We will not be needing the ID column for classification purposes so this can now be removed.

```
bcwid=bcw$ID # Store IDs in case we need them later
bcw=bcw %>% dplyr::select(-ID) # Remove ID column
```

3. Dealing With Missing Values

We need to test for missing values. It would seem there are no missing values, however these are denoted by question marks instead of NAs:

```
sapply(bcw,function(x) sum(is.na(x))) # No NAs (missing values only denoted by '?')
```

```
##      Clump_Thickness  CellSize_Uniformity CellShape_Uniformity 
##                    0                    0                    0 
##        Marg_Adhesion       CellSize_S_Epi          Bare_Nuclei 
##                    0                    0                    0 
##      Bland_Chromatin      Normal_Nucleoli              Mitoses 
##                    0                    0                    0 
##                Class 
##                    0
```

```
sapply(bcw,function(x) sum(x=="?")) # 16 missing values for Bare Nuclei
```

```
##      Clump_Thickness  CellSize_Uniformity CellShape_Uniformity 
##                    0                    0                    0 
##        Marg_Adhesion       CellSize_S_Epi          Bare_Nuclei 
##                    0                    0                   16 
##      Bland_Chromatin      Normal_Nucleoli              Mitoses 
##                    0                    0                    0 
##                Class 
##                    0
```

Convert these question marks to NAs to make them easier to work with:

```
bcw$Bare_Nuclei[bcw$Bare_Nuclei=="?"]="" # Blank values are automatically registered as NAs
```

Since the missing values are in only one of ten feature columns (the 'Bare_Nuclei' column), and only 16 out of the 699 values are missing, it would be wasteful to remove the samples with missing values completely.

Instead, we will try imputing these missing values using two methods:

a. conditional mean imputation
b. random forest imputation

Method A: Conditional Mean Imputation

A simple method of imputing the missing values is to use conditional mean imputation.

Convert the 'Bare_Nuclei' column from character to integer values it can be easily used in mathematical calculations such as calculating means:

```
bcw$Bare_Nuclei=as.integer(bcw$Bare_Nuclei)
```

First find the mean 'Bare_Nuclei' values for the non-null Benign and Malignant samples respectively:

```
benign.mean=mean(as.integer(bcw$Bare_Nuclei[!is.na(bcw$Bare_Nuclei)&bcw$Class=="Benign"])) #
 Mean of non-NA benign Bare Nuclei values
malignant.mean=mean(as.integer(bcw$Bare_Nuclei[!is.na(bcw$Bare_Nuclei)&bcw$Class=="Malignant"
])) # Mean of non-NA malignant Bare Nuclei values
```

Then round the means to the nearest integer so that they are comparable to the other values, and insert these into the dataset in place of the NAs. We round these to integers so that we don't give a false illusion of having a greater deal of accuracy that we actually do (all the other data is given as integers from 1 to 10).

```
bcw.impute.V1=bcw # Make a copy of the original dataset to store imputed values
bcw.impute.V1$Bare_Nuclei[is.na(bcw$Bare_Nuclei)&bcw$Class=="Benign"]=round(benign.mean) # Re
place NAs with the mean values
bcw.impute.V1$Bare_Nuclei[is.na(bcw$Bare_Nuclei)&bcw$Class=="Malignant"]=round(malignant.mea
n)
bcw.impute.V1$Bare_Nuclei=as.integer(bcw.impute.V1$Bare_Nuclei) # Make sure the values are st
ill represented as integers
```

Method B: Random Forest Imputation

More sophisticated methods can also be used for imputing missing values. One such method uses random forests to do so.

First, the missing values are replaced by an average value for the column and a random forest is built. A proximity matrix is used to keep track of which samples end up in the same leaf node of each tree in the forest. Then the missing values are replaced with weighted averages of each sample where the weights are determined by the relative proportion of times samples ended up in the same leaf node as the sample with the missing value. This process is then repeated several times with new random forests, until the imputed values stabilise. [18]

In this case, 6 random forests are built to impute the missing values (4-6 iterations are recommended by Breiman). [2]

```
set.seed(1) # set seed so results are reproducible (note that I will reset the seed increment
ed by 1 at the start of each code block that involves randomness in any way)

bcw.impute.V2=data.frame(sapply(bcw[,-10],as.numeric),bcw$Class) # Random forest imputation r
equires features to be numerical variables

names(bcw.impute.V2)=c("Clump_Thickness", "CellSize_Uniformity",
             "CellShape_Uniformity", "Marg_Adhesion","CellSize_S_Epi",
             "Bare_Nuclei", "Bland_Chromatin", "Normal_Nucleoli", "Mitoses", "Class")

bcw.impute.V2=rfImpute(Class~.,data=bcw.impute.V2,iter=6) # Apply random forest missing value
imputation
```

Let's see if random forest imputation gives meaningfully different values to mean imputation:

```
round(bcw.impute.V2$Bare_Nuclei[bcw.impute.V2$Bare_Nuclei %% 1 !=0],0) # Gives set of imputat
ed values from RF imputation (imputed values are decimal numbers so we can detect them by loo
king for a remainder when dividing by 1)
```

```
##  [1] 8 8 1 1 1 1 1 1 1 8 1 1 7 1 1 1
```

```
c(bcw.impute.V1$Bare_Nuclei[is.na(bcw$Bare_Nuclei)&bcw$Class=="Benign"],bcw.impute.V1$Bare_Nu
clei[is.na(bcw$Bare_Nuclei)&bcw$Class=="Malignant"]) # Imputed values from mean imputation
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 8 8
```

When rounding the imputed values we mainly get values of 1 and 8 (except one value of 7), which correspond exactly to the mean values for benign and malignant tumours respectively. However, we get a greater number of high predictions for random forest imputation than mean imputation (as can be seen above). Let's choose random forest imputation, because the predictions are more individualised than with mean imputation.

```
bcw.impute.V2$Bare_Nuclei=round(bcw.impute.V2$Bare_Nuclei) # Use rounded versions of the impu
ted values

bcwc=data.frame(sapply(bcw.impute.V2[,-1],as.integer),bcw.impute.V2$Class) # Convert all the
 features back to integers

names(bcwc)=c("Clump_Thickness", "CellSize_Uniformity",
             "CellShape_Uniformity", "Marg_Adhesion","CellSize_S_Epi",
             "Bare_Nuclei", "Bland_Chromatin", "Normal_Nucleoli", "Mitoses", "Class")
```

We now have our cleaned dataset (bcwc = Breast Cancer Wisconsin Clean).

It would be interesting to validate the decision to choose random forest imputation over mean imputation. To do this, let's create default random forest models for each method and compare the misclassification rates.

```
set.seed(2)
m.imputation=randomForest(Class ~ ., data=bcw.impute.V1) # RF model using the dataset with me
an imputation
rf.imputation=randomForest(Class ~ ., data=bcwc) # RF model using the dataset with RF imputat
ion
m.imputation
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = bcw.impute.V1)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 3.29%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       446        12  0.02620087
## Malignant     11       230  0.04564315
```

```
rf.imputation
```

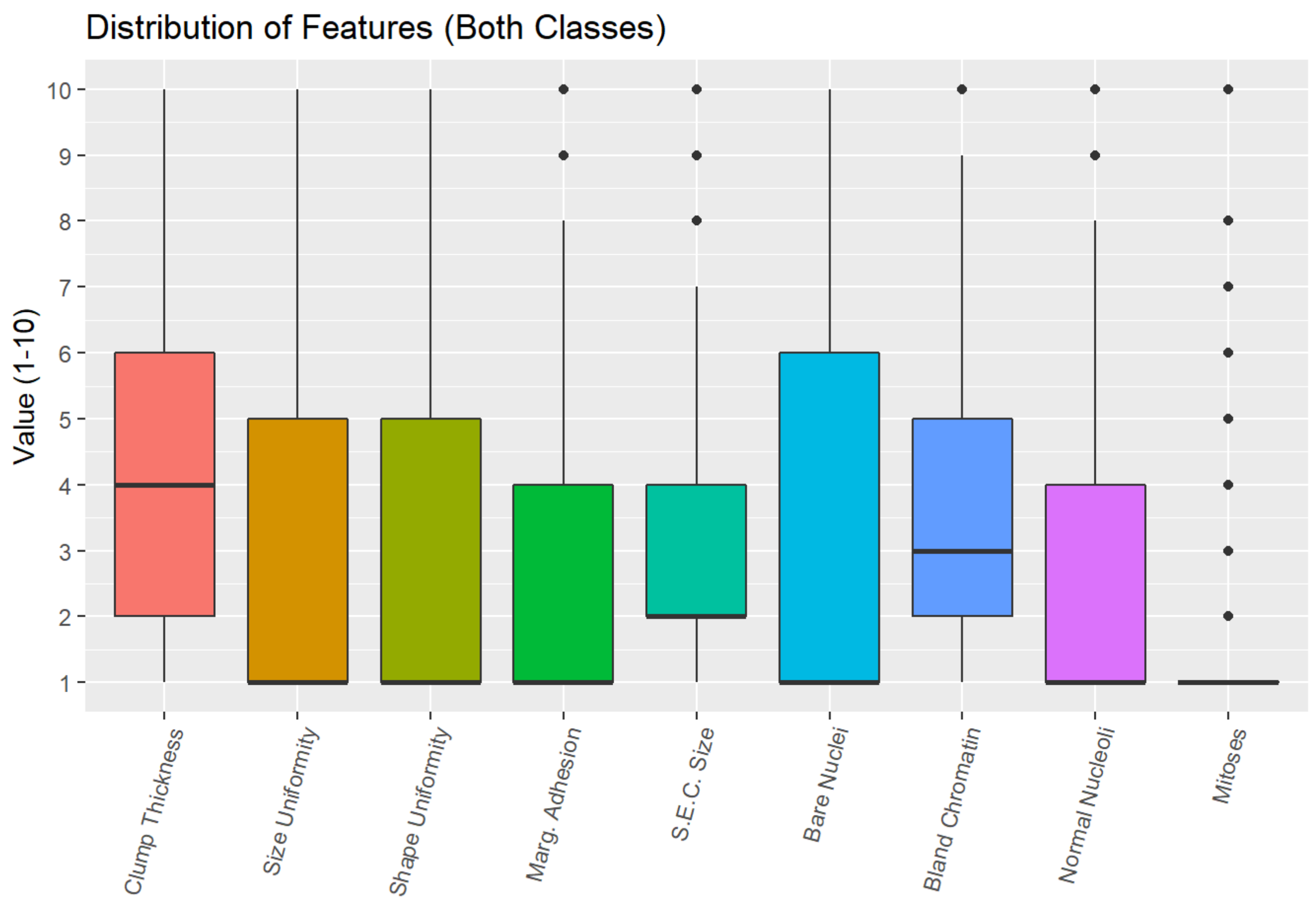
```
##
## Call:
##  randomForest(formula = Class ~ ., data = bcwc)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 2.86%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       445        13  0.02838428
## Malignant      7       234  0.02904564
```

Indeed, the random forest imputation performs slightly better than the mean imputation with a misclassification rate of 2.86% compared with 3.29%, although it is quite possible that this difference is not statistically significant.

4. Feature Distribution

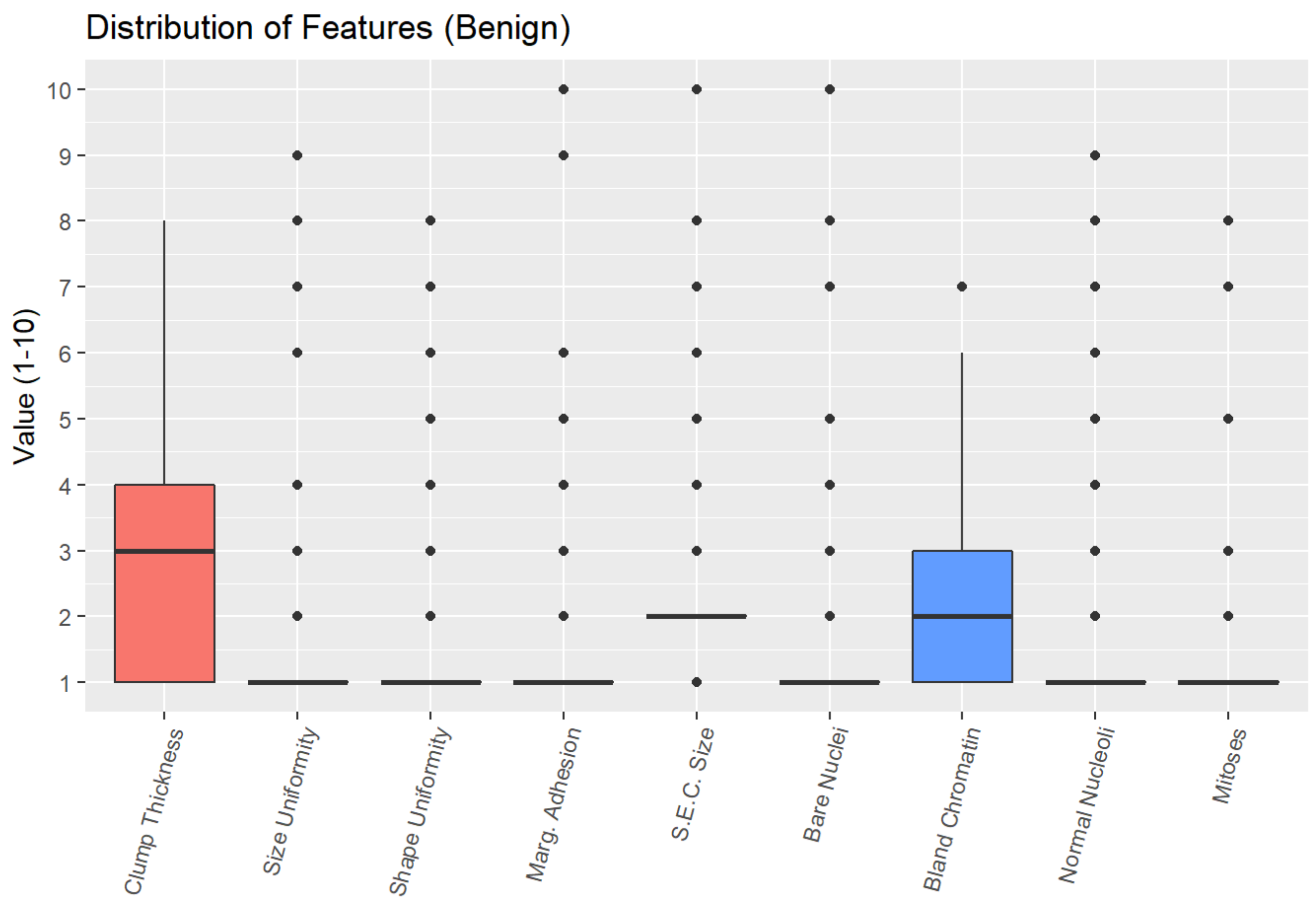
An interesting observation is that for a lot of variables there is heavy positive skew, with the vast majority of values being 1 (the lowest possible value). This is demonstrated by the following boxplot. Only ‘Clump_Thickness’ and ‘Bland_Chromatin’ appear to have somewhat symmetric distributions.

```
bcwcbox=bcwc # Create a copy of the dataset so that we can easily use cleaner names when plot
ting boxplots
names(bcwcbox)=c("Clump Thickness", "Size Uniformity",
             "Shape Uniformity", "Marg. Adhesion","S.E.C. Size","Bare Nuclei", "Bland Chromat
in", "Normal Nucleoli", "Mitoses","Class")

# Boxplot 1
ggplot(stack(bcwcbox[1:9]),aes(x=ind,y=values,fill=ind))+
  geom_boxplot()+
  labs(title="Distribution of Features (Both Classes)",x="",y="Value (1-10)") +
  theme(axis.text.x  =  element_text(angle  =  75,  hjust  =  1),legend.position="none")+
  scale_y_continuous(breaks = scales::pretty_breaks(n = 10))
```



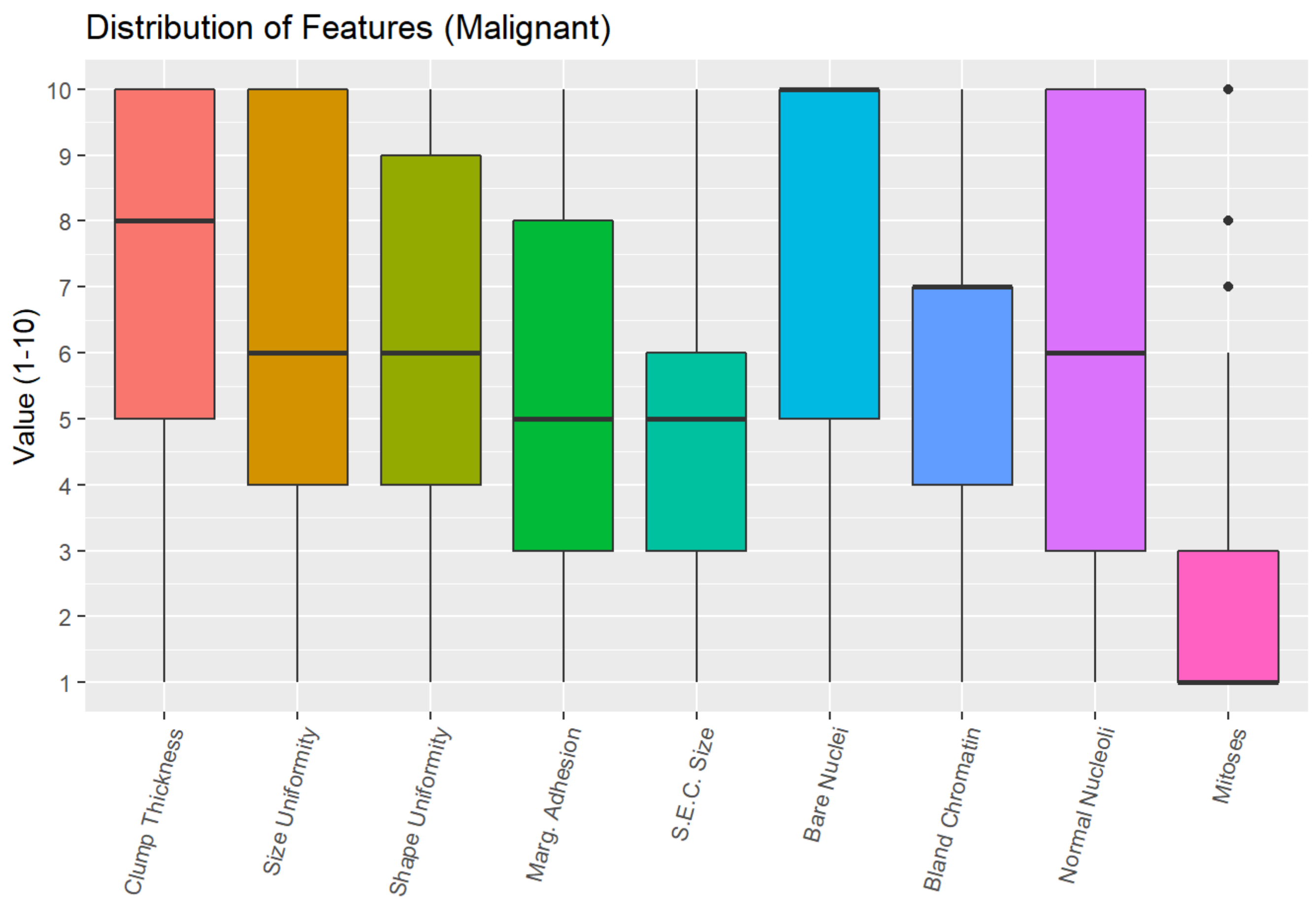
The reason the overall dataset is positively skewed is that the vast majority of attribute values for benign tumours are 1 as can be seen below:

```
# Boxplot 2
ggplot(stack(bcwcbox[bcwcbox$Class=="Benign",1:9]),aes(x=ind,y=values,fill=ind))+
  geom_boxplot()+
  labs(title="Distribution of Features (Benign)",x="",y="Value (1-10)") +
  theme(axis.text.x  =  element_text(angle  =  75,  hjust  =  1),legend.position="none")+
  scale_y_continuous(breaks = scales::pretty_breaks(n = 10))
```



However, malignant tumours tend to have attribute values greater than one (and these values are spread out across the whole range from 1 to 10) as can be seen below:

```
# Boxplot 3
ggplot(stack(bcwcbox[bcwcbox$Class=="Malignant",1:9]),aes(x=ind,y=values,fill=ind))+
  geom_boxplot()+
  labs(title="Distribution of Features (Malignant)",x="",y="Value (1-10)") +
  theme(axis.text.x  =  element_text(angle  =  75,  hjust  =  1),legend.position="none")+
  scale_y_continuous(breaks = scales::pretty_breaks(n = 10))
```



The fact that benign tumours have attribute values mainly equal to one whereas malignant tumours tend to have attribute values greater than one provides a clear source of differentiation, suggesting we can expect high performance from our classification models.

5. Feature Selection

Now that we have our cleaned dataset, it is time to move onto feature selection. Overall, there are three categories of feature selection methods: filtering methods, embedded methods, and wrapper methods.

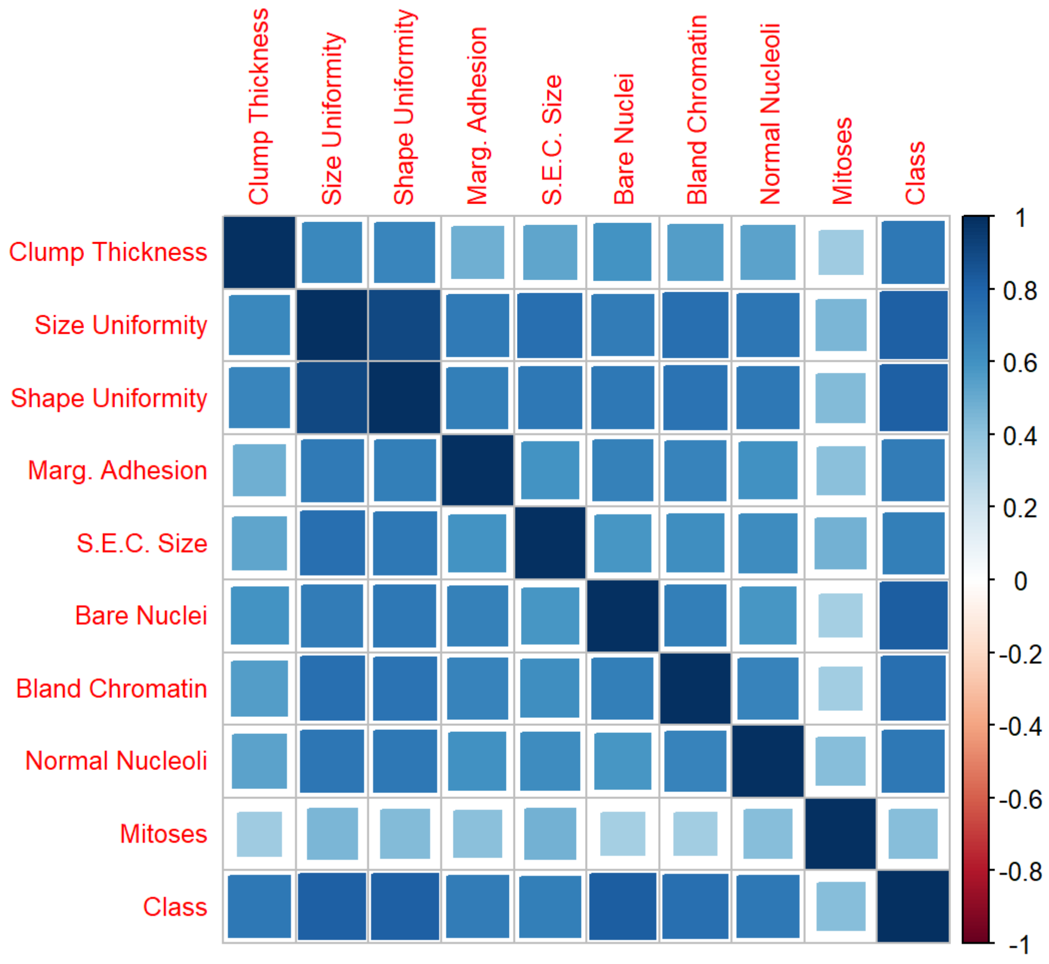
We will try one of each method:

a. correlation analysis (filtering method)
b. variable importance plot (embedded method)
c. Boruta algorithm (wrapper method)

Method A: Correlation analysis

We can construct a pairwise correlation plot to look at the strength of correlation between each feature and the outcome variable, and also the correlation between the features themselves (multicollinearity). The coefficients reported are the Pearson correlation coefficients.

```
# Pairwise Correlation Plot
c=cor(mutate(bcwcbox[1:9],Class=as.integer(bcw$Class)))
corrplot(c,method='square',number.cex=0.7,tl.cex=0.8)
```



We see that cell shape and cell size uniformity are highly correlated with each other (0.91) so we might consider removing one of these features. However, the main problem with multicollinearity is that it makes regression coefficients volatile [11]. In the context of decision trees (and hence random forests), multicollinearity would make some of the decision criteria volatile with respect to small changes in the data or parameters. However, explainability and tree robustness are not priorities when constructing a random forest model (which is often considered a black box). Instead, we care mostly about predictive power, in which case multicollinearity is not a problem. We do care about the time taken to construct a random forest model, but considering that our dataset is relatively small (699 rows by 10 columns), we can afford to keep both cell shape and cell size uniformity as features.

Note that since Class is a binary variable, R automatically uses the Point-Biserial correlation coefficient when calculating correlation between the outcome and features (the Point-Biserial coefficient is a special case of the Pearson coefficient) [11].

Mitoses is the feature that is the most weakly correlated with ‘Class’ given its correlation coefficient of 0.42. The following (two-tailed) statistical test shows this coefficient is statistically different from zero (at almost any significance level) because zero does not lie anywhere near the 95% confidence interval (and the p-value is tiny).

```
cor.test(as.integer(bcwc$Class),bcwc$Mitoses,alternative="two.sided") # two-tailed test of th
e correlation coefficient between two columns
```

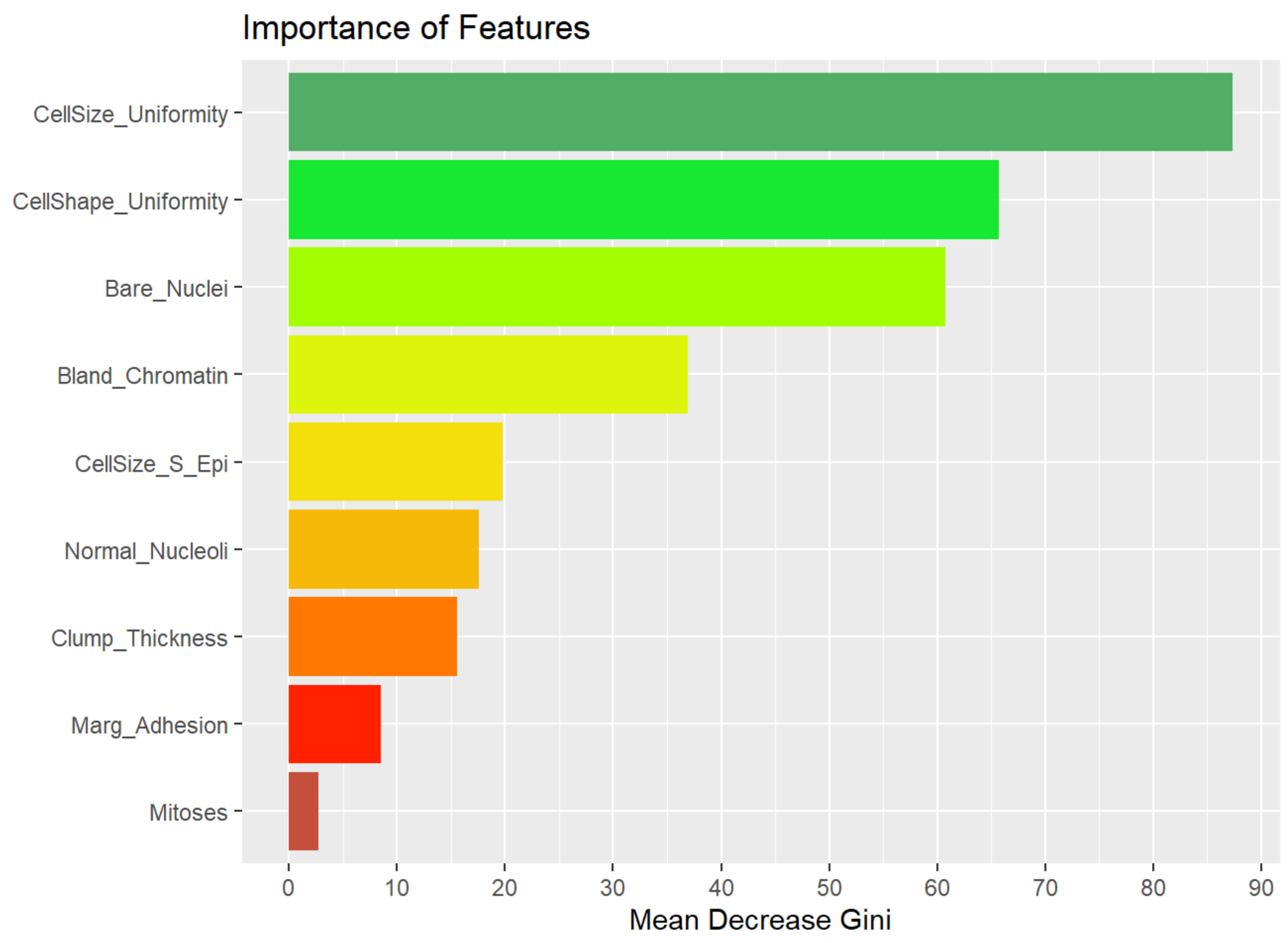
```
##
##  Pearson's product-moment correlation
##
## data:  as.integer(bcwc$Class) and bcwc$Mitoses
## t = 12.33, df = 697, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.3603215 0.4821946
## sample estimates:
##       cor
## 0.4231703
```

This statistical significance of the correlation coefficient suggests that ‘Mitoses’ has predictive power and there is therefore no reason to remove this feature.

Method B: Variable Importance Plot

We can construct a variable importance plot to see which variables would cause the greatest accuracy drop if they were to be removed. This plot can be used as an embedded feature selection method because the selection process is embedded within the random forest model itself. Therefore, we must create a default random forest model before using this method.

```
set.seed(3)
default=randomForest(Class ~ ., data=bcwc) # RF model with default parameter values

imp=data.frame(importance(default)) %>% arrange(MeanDecreaseGini) # the importance() function
gives the data we wish to plot, and we sort this in descending order
imp=data.frame(variable=rownames(imp),importance=imp[1:9,1]) # Convert the row names into an
 actual column of data
imp$variable=factor(imp$variable,levels=imp$variable) # Converting to a factor allows us to m
aintain order when plotting a bar chart

# Variable Importance Plot
ggplot(imp,aes(x=variable,y=importance,fill=variable))+
geom_bar(stat = "identity")+
theme(legend.position="none")+coord_flip()+
labs(title="Importance of Features",y="Mean Decrease Gini",x="") +
scale_fill_manual(values=c("#c44f3b","#ff2100", "#ff7800", "#f7b908", "#f4de0b", "#dcf50a",
"#a3ff00", "#17e830","#52ad67"))+
scale_y_continuous(breaks = scales::pretty_breaks(n = 10))
```



From the plot, we see that cell size and shape are the most important features whilst mitoses and marginal adhesion are the least important features.

Let's now try removing these features and see if this has an effect on the model performance:

```
set.seed(4)
bcwc2=bcwc[,-c(4,9)] # Remove 'Mitoses' and 'Marginal Adhesion'
reduced=randomForest(Class ~ ., data=bcwc2) # Create new model without the above two features
reduced
```

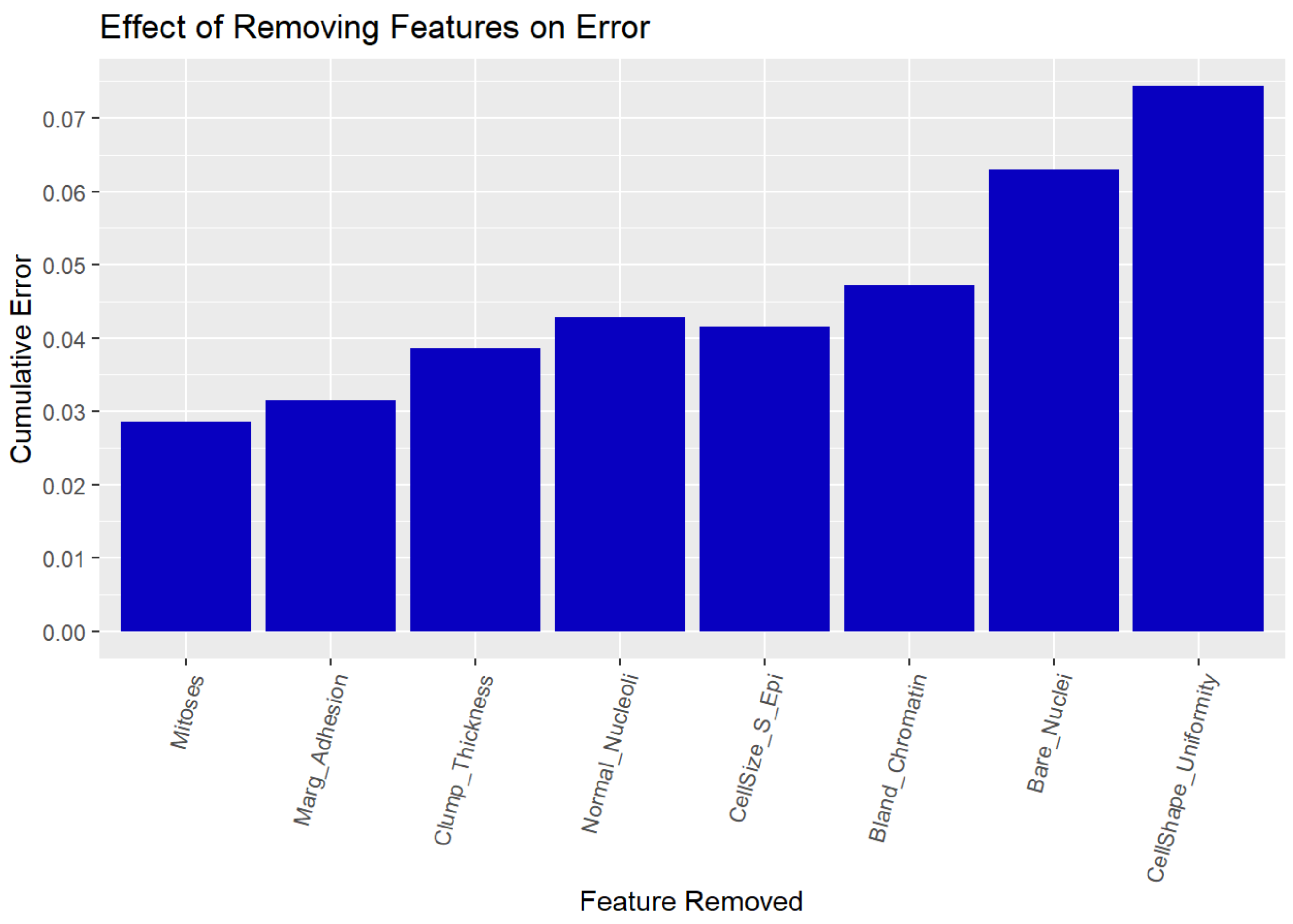
```
##
## Call:
##  randomForest(formula = Class ~ ., data = bcwc2)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 2.86%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       445        13  0.02838428
## Malignant      7       234  0.02904564
```

Interestingly, there is no difference in OOB error rate when we remove these two features (in fact, even the sensitivity and specificity haven't changed).

This raises the question of whether other features could be removed too.

Note that number of features tried at each node (mtry) decreased from 3 to 2 when we removed these two features. This is because R takes the default value for mtry to be the square root of the number of features.

```r
# The order in which features should be removed is given by the variable importance plot (see
above)
varorder=c(9,4,1,8,5,7,6,3,2) # leftward column numbers should be removed earlier

set.seed(5)
chosen=vector(length=0)
storage=vector(length=length(varorder)-1) # vector to store our results in

for(i in 1:(length(varorder)-1))
{
  chosen=varorder[1:i] # the number of variables we wish to exclude increases with each itera
tion
  bcwc.temp=bcwc[,-chosen] # exclude the vector of columns given by 'chosen'
  temp.model=randomForest(Class ~ ., data=bcwc.temp) # build a model with the excluded column
s
  storage[i]=temp.model$err.rate[nrow(temp.model$err.rate),1] # this gives the OOB error rate
for the model with the excluded columns

}

errors=data.frame(imp[1:8,1],cumulative.error=storage) # convert error results to a data fram
e so we can easily use ggplot2

# Bar chart
ggplot(errors,aes(x=imp[1:8,1],y=cumulative.error))+
geom_bar(stat = "identity",fill="#0800c0")+
labs(title="Effect of Removing Features on Error",y="Cumulative Error",x="Feature Removed") +
scale_y_continuous(breaks = scales::pretty_breaks(n = 10))+
theme(axis.text.x  =  element_text(angle  =  75,  hjust  =  1))
```



```
errors
```

imp.1.8..1.	**cumulative.error**
<fct>	<dbl>
Mitoses	0.02861230
Marg_Adhesion	0.03147353
Clump_Thickness	0.03862661
Normal_Nucleoli	0.04291845
CellSize_S_Epi	0.04148784
Bland_Chromatin	0.04721030
Bare_Nuclei	0.06294707
CellShape_Uniformity	0.07439199

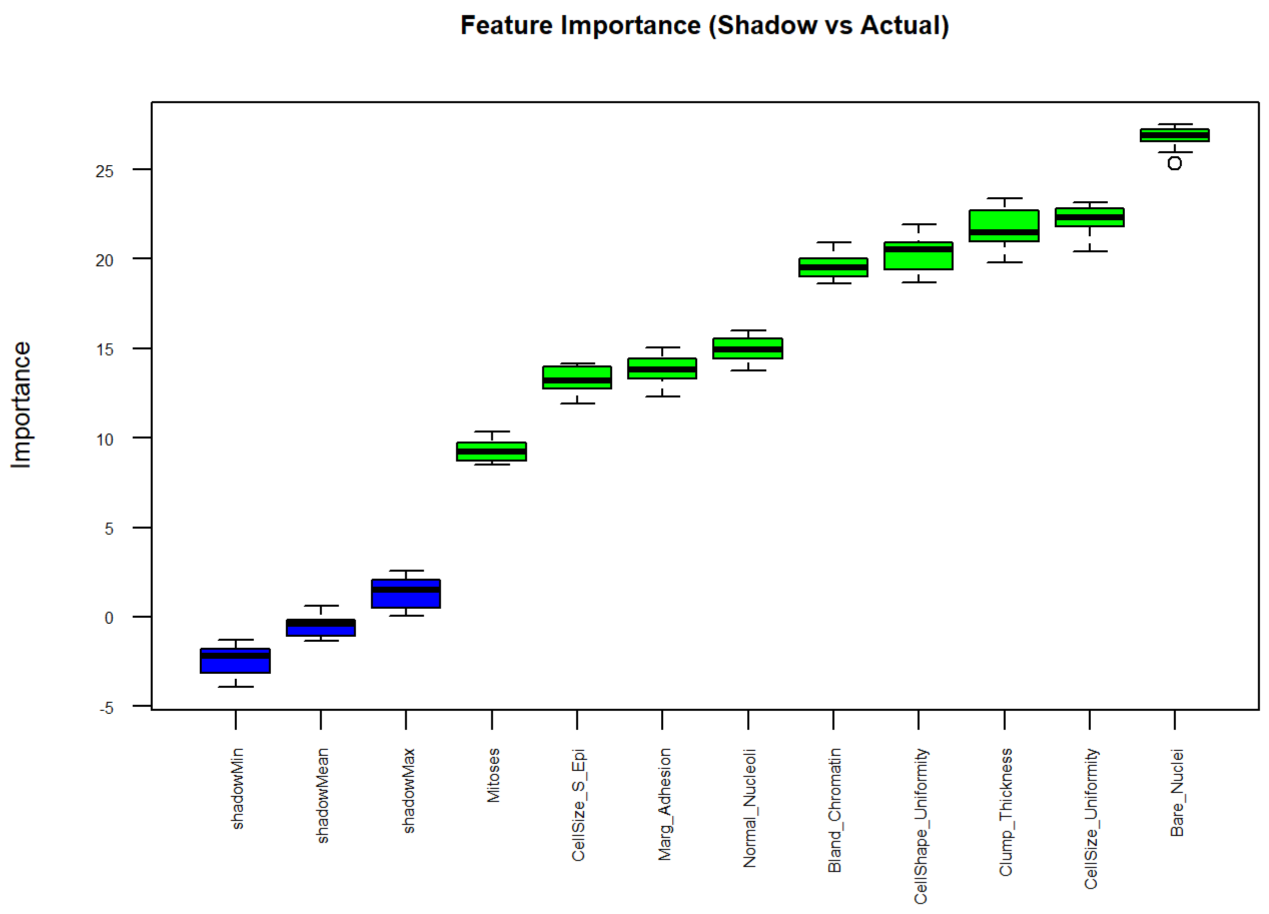
8 rows

Even if we only keep the most important feature (and remove all others), we still get an accuracy of 92.6%, and if we keep only the three most important features we get an accuracy of 95.3%. This suggests that outside of the top three features, the remaining six features collectively only contribute to just under 2% more accuracy, which calls into question whether it is worth including all of these features.

Method C: Boruta algorithm

The Boruta algorithm is a wrapper feature selection method that uses Random Forests for eliminating features. It works by making copies of the dataset, randomly shuffling the columns of data (to create ‘shadow features’) and comparing the performance of random forest models with the actual features versus with the shadow features. If the models with an actual feature don’t perform better than the models with a randomised shadow feature then that feature should be removed [4].

```
set.seed(6)
boruta=Boruta(Class~.,data=bcwc) # execute Boruta algorithm
plot(boruta,las=2,cex.axis=0.5,cex.lab=0.8,xlab="",main="Feature Importance (Shadow vs Actua
l)",cex.main=0.8) # plot feature importance
```



```
attStats(boruta)[5] # show the percentage of times each feature beat the shadow features
```

	normHits <dbl>
Clump_Thickness	1
CellSize_Uniformity	1
CellShape_Uniformity	1
Marg_Adhesion	1
CellSize_S_Epi	1
Bare_Nuclei	1
Bland_Chromatin	1

	normHits <dbl>
Normal_Nucleoli	1
Mitoses	1

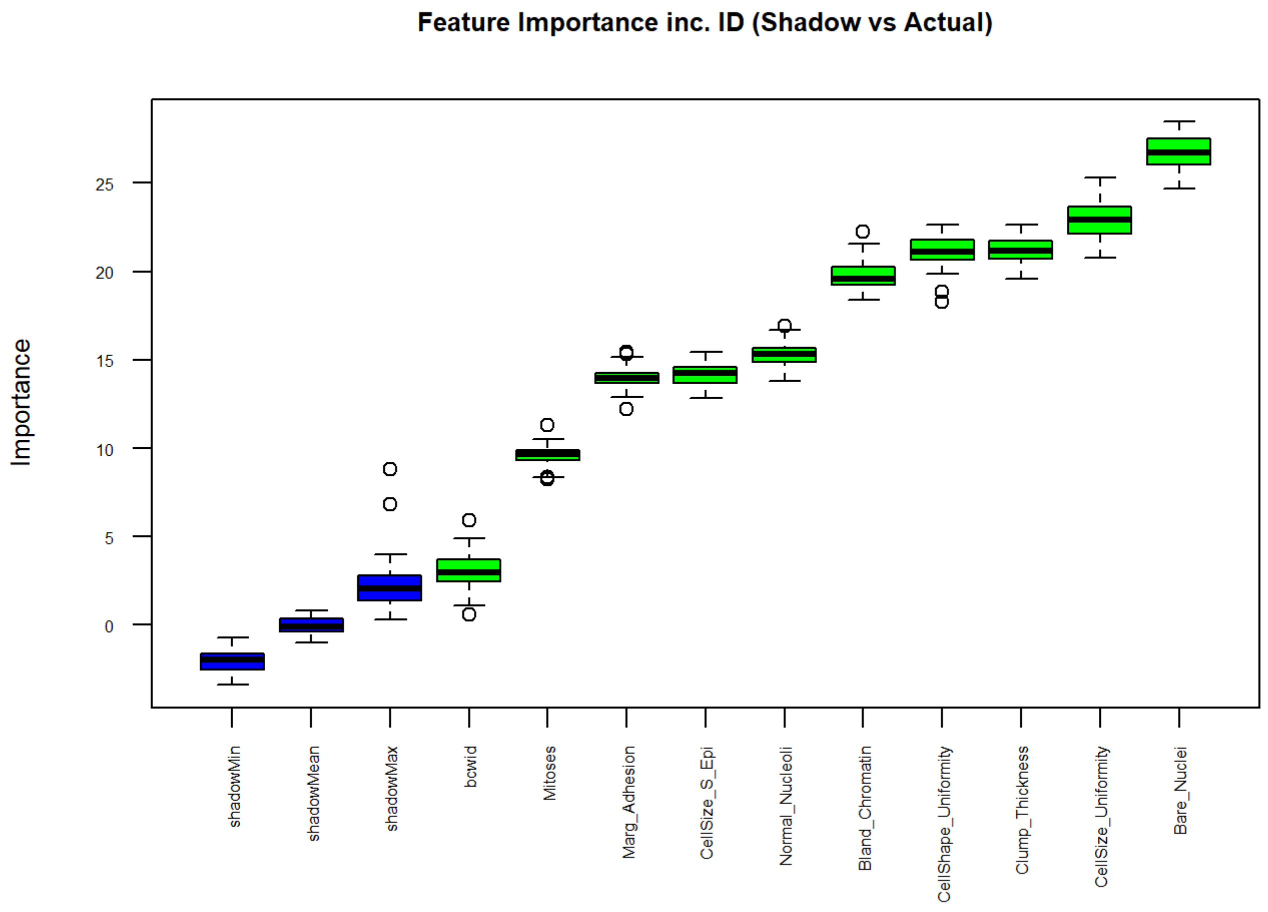
9 rows

Running the Boruta algorithm on the cleaned dataset shows that all 9 features are classed as important enough to not be removed. In fact, the attStats() function shows that all the features beat the shadow features 100% of the time. This contrasts with our prior observation that removing the two least important features made no difference to misclassification rate.

Another interesting observation is that Boruta classed Bare Nuclei as the most important feature, whereas cell size and shape uniformity were most important features specified in the variable importance plot from earlier.

To make sure that the Boruta algorithm is working, let's add ID back and see if ID is classed as insignificant (this should be the case assuming that Dr Wolberg observed the same ratio of benign to malignant cases over time)

```
set.seed(7)
bcwcid=dplyr::mutate(bcwc,bcwid) # add back the ID column
boruta=Boruta(Class~.,data=bcwcid)
plot(boruta,las=2,cex.axis=0.5,cex.lab=0.8,xlab="",main="Feature Importance inc. ID (Shadow v
s Actual)",cex.main=0.8)
```



```
round(attStats(boruta)[5],3)
```

	normHits <dbl>
Clump_Thickness	1.000
CellSize_Uniformity	1.000
CellShape_Uniformity	1.000
Marg_Adhesion	1.000
CellSize_S_Epi	1.000
Bare_Nuclei	1.000
Bland_Chromatin	1.000
Normal_Nucleoli	1.000
Mitoses	1.000
bcwid	0.725

1-10 of 10 rows

ID is actually classed as a significant feature by Boruta, so clearly the algorithm is reluctant to remove features. However, the algorithm appears to be working, because we see that ID only performed better than a useless shadow feature 72.5% of the time (whereas all the other features always beat the shadow features). The 72.5% figure could be explained by a subtle change in the ratio of benign to malignant cases seen by Dr Wolberg over time.

Bringing together our three feature selection methods:

Following the variable importance plot, we saw that removing the weakest features (mitoses, and marginal adhesion) did not reduce accuracy (although doing so also did not increase accuracy). Despite this, the Boruta algorithm advocated keeping all features, and this is supported by the correlation analysis (where even the weakest correlation with the outcome was significantly different from zero). Given these findings, and that computational load is not a big consideration due to the small dataset, no features will be deleted in the feature selection process.

6. Partitioning Data

Now that the dataset has been cleaned and features have been selected, we will partition it. Traditionally, it is recommended to split the dataset into a training set (for creating the model), a validation set (for tuning parameters, alternatively this set can be combined with the training set in a cross-validation procedure), and a testing set (completely unseen data for evaluating the final optimised model) [13].

Random forest uses bootstrapping to create each tree in the forest, which leaves unused observations for each tree. This acts as a form of cross-validation (with each bootstrapped dataset leaving a different set of unused observations), and hence the training set used for a random forest model acts as both a training and a validation set.

However, it is still worth setting aside a testing dataset. Since we will be using the OOB error rate to optimise parameters, we are still using the out-of-bag (OOB) data for tuning parameters, so it is good to test our final model on a completely unseen dataset [5]. This is because parameters are tuned based on how well they perform on the training/validation datasets and therefore even OOB error rates will be biased downwards because the parameters are designed to perform particularly well on the OOB data. Applying the chosen model specification to a fresh testing set is therefore the only way to avoid bias. Also, setting aside a traditional testing dataset will allow for easier comparison to non-bootstrap based methods [5].

We will use an 85%-15% training-testing split because there are a fairly low number of observations (699) so a high proportion of the data should be reserved for training, whilst the 15% testing set still leaves over 100 observations to test the final model on.

We use stratified sampling to form the training and testing datasets to ensure that the proportion of benign and malignant cases in each set reflects the proportions in the whole dataset (this is helpful because there are a relatively low number of malignant cases and these need to be shared correctly between the training and testing datasets):

```
set.seed(8)

indices=createDataPartition(y=bcwc$Class,p=0.85,list=FALSE) # Take 85% of the data for the tr
aining set using stratified sampling to maintain the dataset's class distribution

bcwctrain=bcwc[indices,]
bcwctest=bcwc[-indices,] # The remaining data should form the testing set

# In the whole dataset, 65.5% of observations are benign, let's test whether the training and
testing sets have been stratified properly
round(sum(bcwctrain$Class=="Benign")/length(bcwctrain$Class),3) # should be around 65.5% if s
tratified properly
```

```
## [1] 0.655
```

```
round(sum(bcwctest$Class=="Benign")/length(bcwctest$Class),3) # should be around 65.5% if str
atified properly
```

```
## [1] 0.654
```

```
default2=randomForest(Class ~ ., data=bcwctrain) # Let's create another version of the defaul
t model, this time using only the training dataset
```

It would also be interesting to test my hypothesis that a validation set is not required when constructing random forest models.

Therefore we will also try 10-fold cross validation on the training set as a comparison point. This will essentially create nested cross validation, first with the 10-fold CV, second with the bootstrapping process (i.e. we are splitting the data into 10-folds, then taking a bootstrapped dataset from 90% of the data, leaving the remaining 10% fresh, then repeating this process, choosing a different allocation of folds each time).

We now set up for the 10-fold cross validation that will be applied later on in this analysis:

```
cvspec=trainControl(method="cv",number=10,savePredictions = "all",classProbs = TRUE) # this f
unction call will be fed into the train() function when we perform CV
```

7. Classification Models

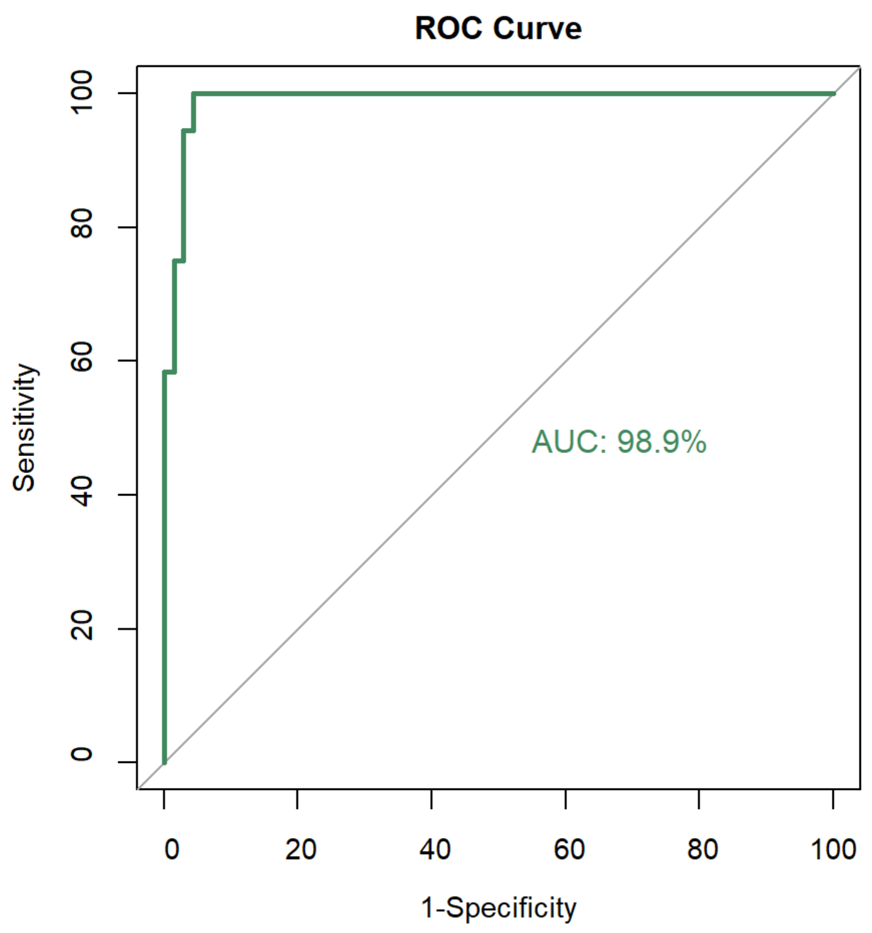
Before tuning any parameters, it would be interesting to see how the default random forest model (with default values for all parameters) performs:

```
predictions=predict(default2,bcwctest) # use the 'default2' random forest model to predict th
e classes of the tumours in the testing dataset
confusion=table(actual=bcwctest$Class,predictions=predictions) # create a confusion matrix to
show the performance of the 'default2' model on the testing dataset
sum(diag(confusion))/sum(confusion) # accuracy calculation
```

```
## [1] 0.9519231
```

The default model achieves an accuracy of 95.2% on the test dataset. Let's now build an ROC curve for this model (this code block is based on [8]):

```
probpred=predict(default2,bcwctest,type="prob") # we store the percentages of trees in the ra
ndom forest that classed each observation as benign and malignant respectively
par(pty="s") # set up the graphics engine to nicely plot ROC curves

# ROC curve
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC Curve",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC Curve", cex.main = 1, cex.
axis = 0.9,     cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 98.94%
```

```
par(pty="m") # revert the changes made to the graphics engine
```

The area under this ROC curve is 98.9% of the total area of the graphing square. A model that can perfectly distinguish between benign and malignant cases would have an AUC of 100% so clearly even the default model shows high classification performance. Note that the 45 degree line represents a worthless model that randomly predicts the class of an observation according to a given cutoff value ranging from 0 to 1, so the further outward the ROC curve is from this line the better.

There are two main parameters to tune in a random forest model:

a. the number of trees
b. the number of variables considered at each node

And there are three additional parameters we will consider tuning:

c. the minimum size of terminal nodes of each tree
d. the balance of benign and malignant observations in each bootstrapped dataset
e. the proportion of votes beyond which an observation is classified as malignant

Parameter A: ntree

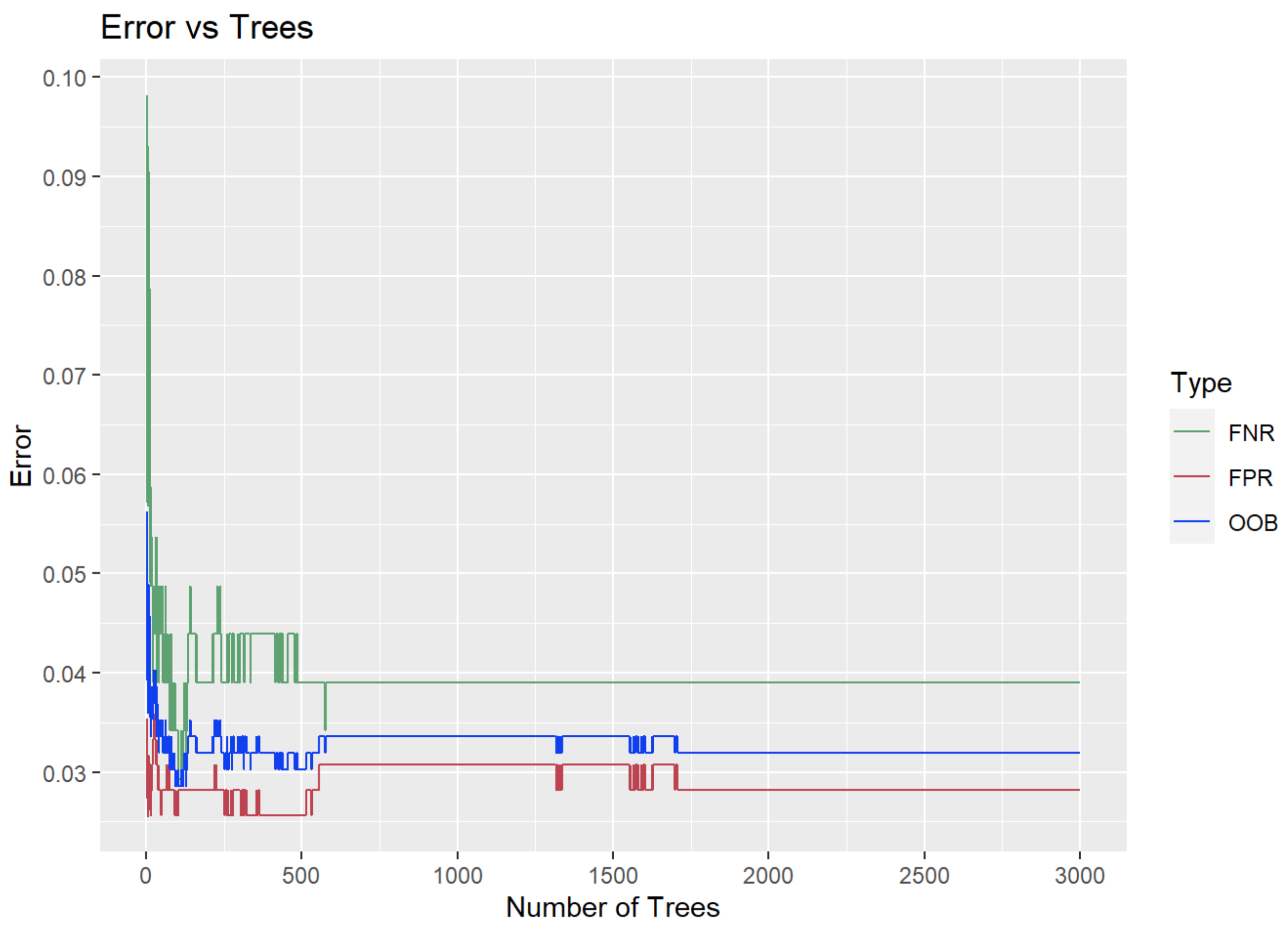
The first main parameter to tune in a random forest model is the number of trees. Let’s plot a graph that shows how the error rates change as we continue to add additional trees to our model:

FNR=false negative rate

FPR=false positive rate

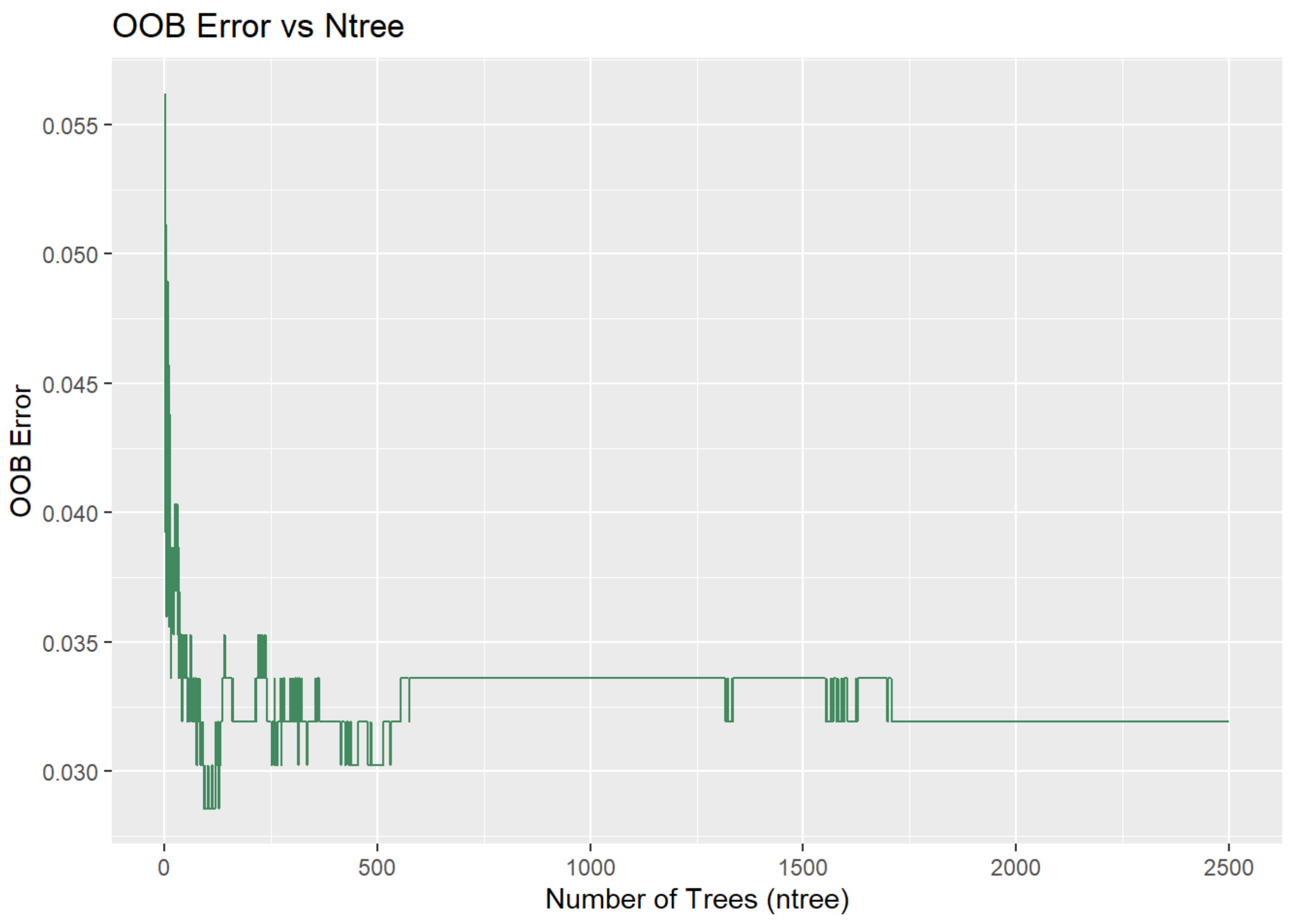
This code segment is based on [3]

```
set.seed(9)
model = randomForest(Class ~ ., data=bcwctrain, ntree=3000) # create a model with 3000 trees

oob.error.data =data.frame( # Store data in a data frame to be used in the call to ggplot
  Trees=rep(1:nrow(model$err.rate), times=3),
  Type=rep(c("OOB", "FPR", "FNR"), each=nrow(model$err.rate)), # first record OOB error for e
ach cumulative tree, then record FPR, and finally record FNR
  Error=c(model$err.rate[,"OOB"],  # Overall OOB error rate
          model$err.rate[,"Benign"], # How frequently Benign tumours are misclassified
          model$err.rate[,"Malignant"])) # How frequently Malignant tumours are misclassified

# Plot line graph
ggplot(data=oob.error.data, aes(x=Trees, y=Error)) +
  geom_line(aes(color=Type))+
  labs(title="Error vs Trees",y="Error",x="Number of Trees") +
scale_y_continuous(breaks = scales::pretty_breaks(n = 10))+
  scale_x_continuous(breaks = scales::pretty_breaks(n = 10))+
  scale_color_manual(values=c("#5da270","#bc434f", "#103fef"))
```



Simpler version of this graph to use in the report:

```
Trees=1:2500
Error=model$err.rate[1:2500,"OOB"]
treegraph=data.frame(Trees,Error)

# Plot line graph
ggplot(data=treegraph, aes(x=Trees, y=Error)) +
  geom_line(color="#41895e")+
  labs(title="OOB Error vs Ntree",y="OOB Error",x="Number of Trees (ntree)") +
scale_y_continuous(breaks = scales::pretty_breaks(n = 10))+
  scale_x_continuous(breaks = scales::pretty_breaks(n = 9))
```



We see that the model stabilises after about 600 trees and there is a final drop in error rate after about 1750 trees. It is safer to choose a point a few hundred trees after the model stabilises, which would suggest 2000 trees to be optimal. However, the default number of trees is 500 so choosing 4 times this number would limit the ability to efficiently train and test a large number of models. Therefore, the final drop in error rate will be ignored and 1000 trees will be chosen as the optimal number instead, since this is a few hundred trees after the initial stabilisation at 600 trees.

Parameter B: mtry

The other main parameter to tune is the number of variables randomly selected to be considered at each node in each tree of the random forest (mtry).

We will test values ranging between 1 and 9 corresponding to the minimum and maximum number of features in this dataset. We will choose ntree=1000 following on from the previous section. Given this value for ntree, we measure OOB error rates for models with different values for mtry and choose the value for mtry giving us the lowest OOB error:

This code segment is based on [3].

```
set.seed(10)
oob.values = vector(length=9)
for(i in 1:9) {
  temp.model = randomForest(Class ~ ., data=bcwctrain, mtry=i, ntree=1000) # iteratively crea
te new models with incrementing values for mtry
  oob.values[i] = temp.model$err.rate[nrow(temp.model$err.rate),1] # Access the entry in the
 last row and first column in the error rate matrix (since this is OOB error)
}
oob.values # OOB error values increasing in mtry from 1 to 9
```

```
## [1] 0.02689076 0.02521008 0.03193277 0.03529412 0.03697479 0.04033613 0.04033613
## [8] 0.04033613 0.04201681
```

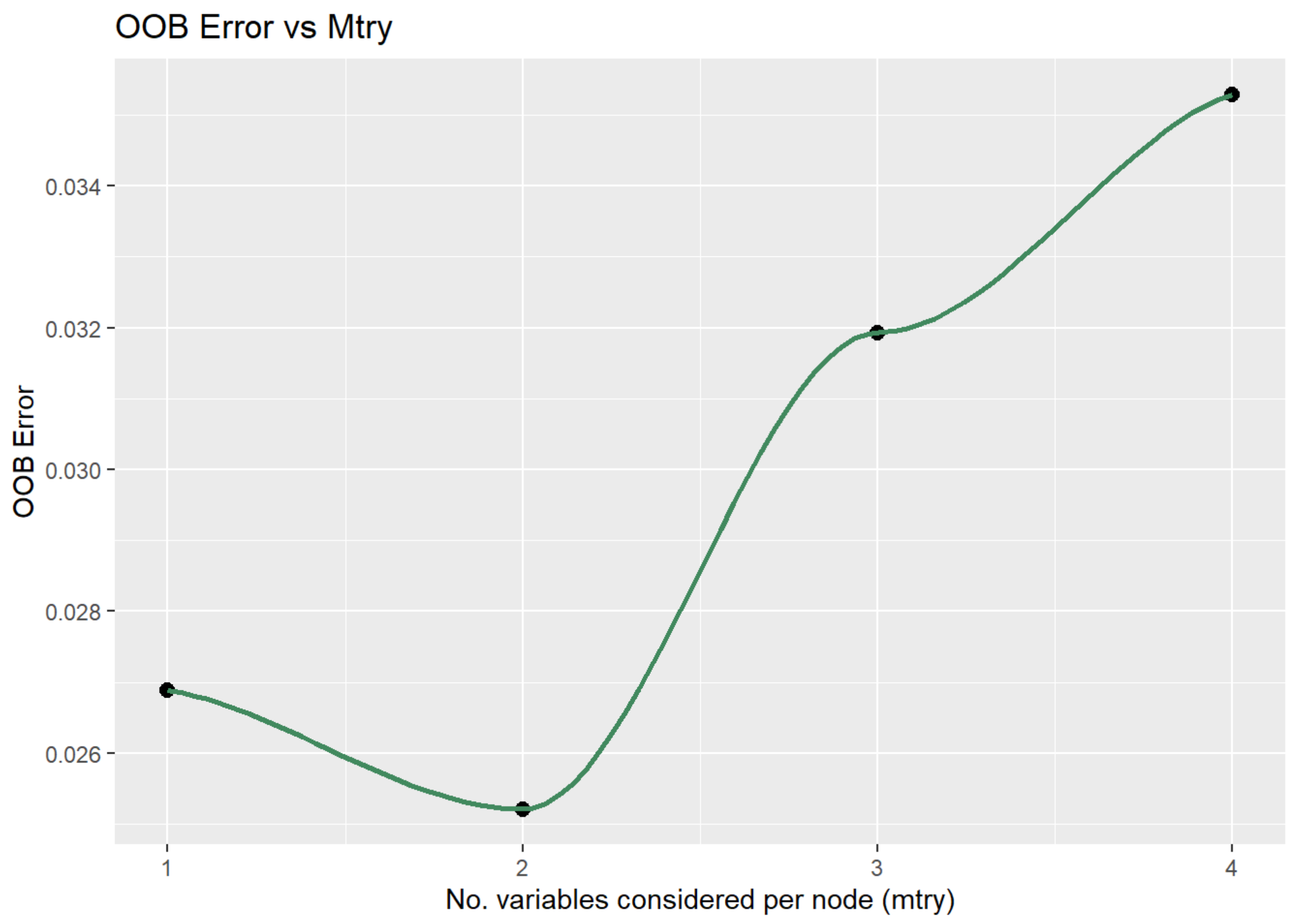
```
which(oob.values == min(oob.values)) # find the optimal value for mtry
```

```
## [1] 2
```

We see that the optimal value for mtry is 2.

Now let’s more closely examine the relationship between mtry and OOB error:

```
# Plot line graph
val=data.frame(mtry=1:4,error=oob.values[1:4])
ggplot(val,aes(x=mtry,y=error))+
  geom_point(size=2.5)+geom_smooth(se=FALSE,color="#41895e")+
    labs(title="OOB Error vs Mtry",y="OOB Error",x="No. variables considered per node (mtry)"
) +
scale_y_continuous(breaks = scales::pretty_breaks(n = 5))
```



We see an interesting u-shape curve by plotting our results.

Lower values of mtry mean that trees will be less correlated because the lack of variables to choose from at each node forces the trees to use all the features (even the weaker ones) rather than just relying on the same few strong features. This lack of correlation between trees prevents overfitting which we would expect to push down the OOB error rate. However, at the same time each tree is forced to use suboptimal features and so individual trees are weaker at predicting. This would push up the OOB error rate. This balance between fitting and overfitting can explain why we observe a u-shape curve (assuming diminishing rate of improvement in individual trees as mtry increases and/or increasing rate of overfitting as mtry increases).

So we have seen that our initial optimised random forest model has ntree=1000 and mtry=2.

Let’s see how it performs on the testing dataset:

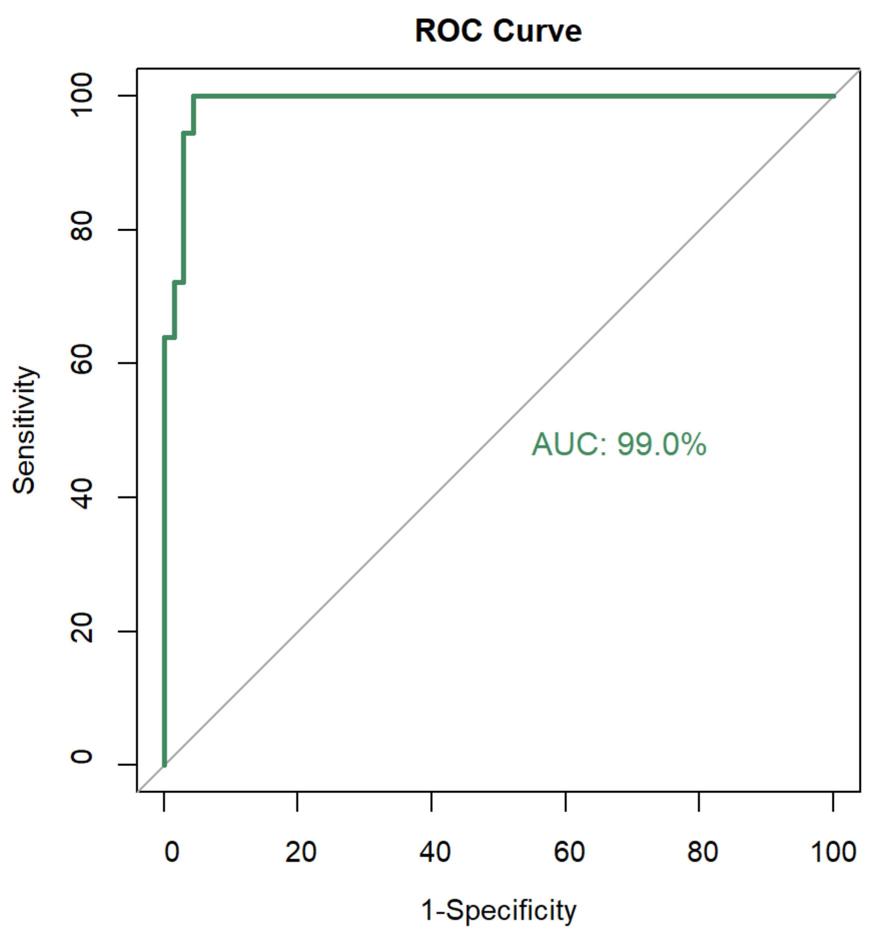
```
set.seed(11)
optimodel=randomForest(Class ~ ., data=bcwctrain, mtry=2, ntree=1000) # build model
predictions=predict(optimodel,bcwctest) # generate predictions
confusion=table(actual=bcwctest$Class,predictions=predictions) # create confusion matrix
sum(diag(confusion))/sum(confusion) # accuracy value
```

```
## [1] 0.9615385
```

An accuracy of 96.2% is achieved on the testing dataset, slightly higher than the 95.2% achieved by the default model.

And let’s also plot an ROC curve for this model:

```
probpred=predict(optimodel,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC Curve",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC Curve", cex.main = 1, cex.
axis = 0.9,     cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 98.98%
```

```
par(pty="m")
```

An AUC of 99.0% is measured on the testing dataset, ever so slightly than the 98.9% from the default model. But the difference is so small that it is not fully clear that this model is better at distinguishing the type of tumour than the default model.

Cross-Validated Model:

As previously mentioned, it would be interesting to compare our initial models to a model built using cross validation (given that there is no clear reason why cross validation would be needed in addition to bootstrapping). Here is this model built with 10-fold CV:

```
set.seed(12)
cvmodel=train(Class~.,data=bcwctrain,method="rf",family=binomial,trControl=cvspec) # build a
 random forest model on the training dataset using 10-fold cross validation
cvmodel=cvmodel$finalModel # access the model that was built
cvmodel
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = min(param$mtry, ncol(x)), family = ..1)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 2.69%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       381         9  0.02307692
## Malignant      7       198  0.03414634
```

We see that the ‘train’ function from caret automatically optimises the model when implementing cross validation. 500 trees and 2 variables at each node are selected as the optimal parameter values. However, mtry is the only parameter actually tuned by the ‘train’ function (ntree is kept at the default value of 500) [16].

Now let’s test the performance of this model on the testing dataset:

```
predictions=predict(cvmodel,bcwctest)
confusion=table(actual=bcwctest$Class,predictions=predictions)
confusion
```

```
##            predictions
## actual      Benign Malignant
##   Benign        66         2
##   Malignant      2        34
```

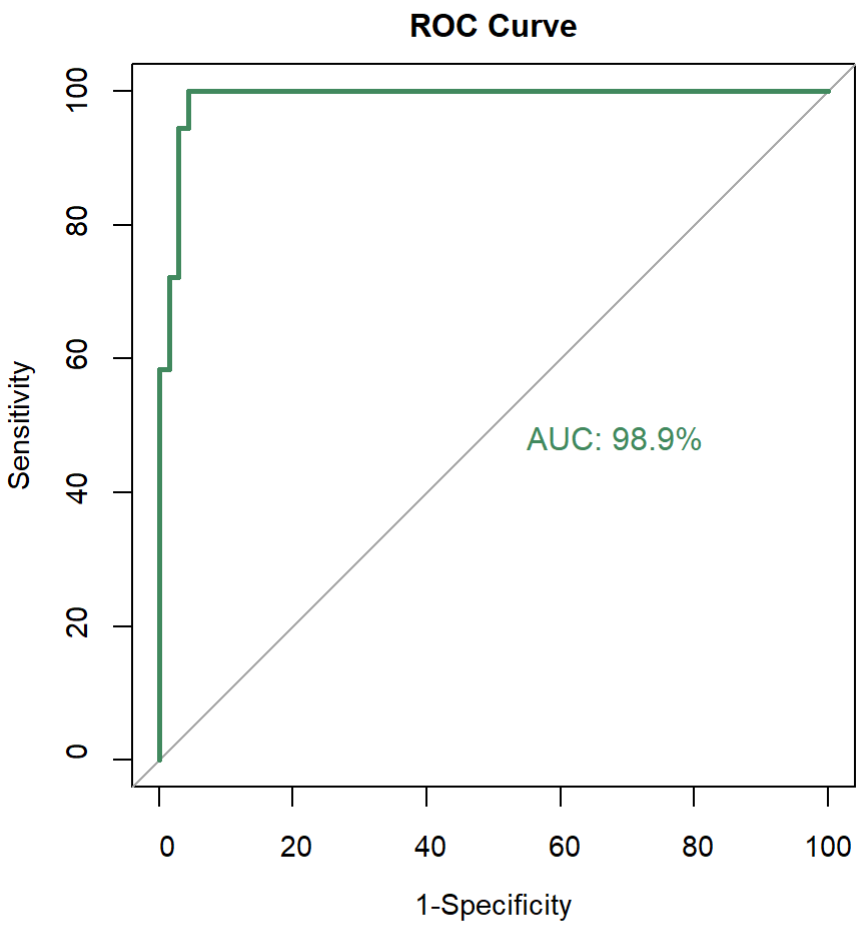
```
sum(diag(confusion))/sum(confusion)
```

```
## [1] 0.9615385
```

We get the same accuracy score of 96.2% when applying the cross validated model to the testing dataset, which suggests that cross validation is not necessary since the same real-world performance can be achieved using only bootstrapping. This also suggests that the extra 500 trees in the non-cross validation model may not have been necessary (which definitely supports the decision not to choose 2000 trees). Adding more trees is generally not rewarded with higher performance after a certain point (which might also explain why the caret package just sticks with the default 500 trees).

Now let’s draw an ROC curve for this model

```
probpred=predict(cvmodel,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC Curve",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC Curve", cex.main = 1, cex.
axis = 0.9,     cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 98.9%
```

```
par(pty="m")
```

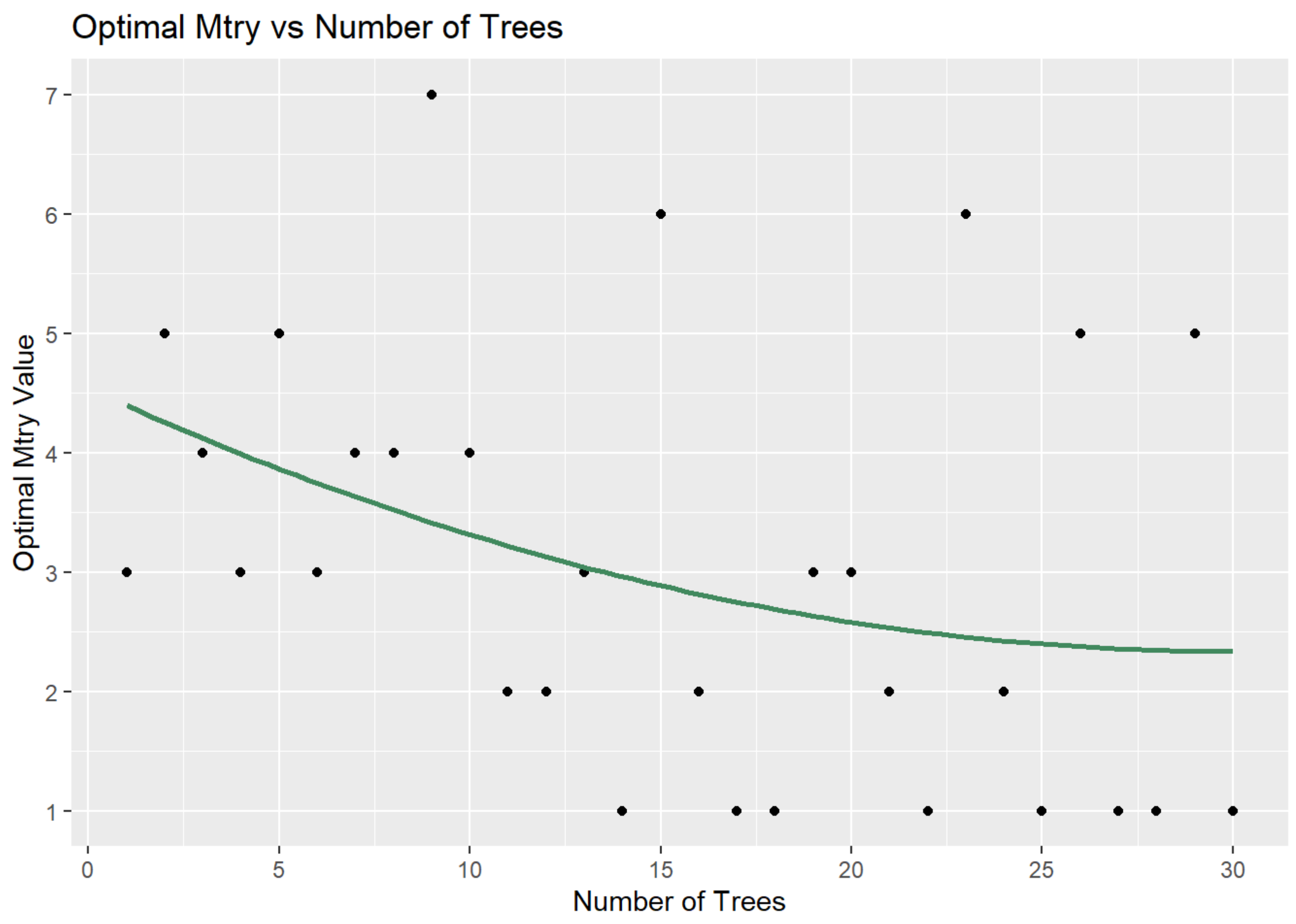
The ROC curve is very similar to the previous two ROC curves, yielding an AUC value of 98.9%. Clearly all models so far have performed quite similarly.

Relationship between ntree and optimal mtry:

I would assume that lower numbers of trees would lead to higher optimal values for mtry (i.e. mtry value that minimises OOB error), because there are not enough trees to benefit much from the benefits of low inter-tree correlation and so it is best to just make strong individual trees using high values for mtry.

Let’s quickly test this hypothesis:

```r
set.seed(13)
hyp=vector(length=30)
for (i in 1:30)
{
  oob.values = vector(length=9) # set up a vector to record OOB for each of the 9 values of m
try
  for (j in 1:9)
  {
      temp.model = randomForest(Class ~ ., data=bcwctrain, mtry=j, ntree=i) # for each value
 of ntree from 1 to 30 build models with each value of mtry
  oob.values[j] = temp.model$err.rate[nrow(temp.model$err.rate),1] # and record the OOB error
rate of each of these models
  }
  hyp[i]=which(oob.values == min(oob.values))[1] # at each value for ntree, choose the value
 of mtry that minimises OOB error
  # If there is a tie in mtry, just take the lower value
}

val2=data.frame(ntree=1:30,mtry_optimal=hyp) # store data in a data frame to be used in the c
all to ggplot2

# Plot scatter graph with a quadratic regression line
ggplot(val2,aes(x=ntree,y=mtry_optimal))+
  geom_point()+geom_smooth(method = "lm",formula = y ~ poly(x, 2),se=FALSE,color="#41895e")+s
cale_y_continuous(breaks = scales::pretty_breaks(n = 7))+scale_x_continuous(breaks = scales::
pretty_breaks(n = 10))+labs(title="Optimal Mtry vs Number of Trees",y="Optimal Mtry Value",x=
"Number of Trees")
```



The hypothesis indeed seems to be supported, more trees enhance the benefits from less-correlated trees, so optimal mtry reduces.

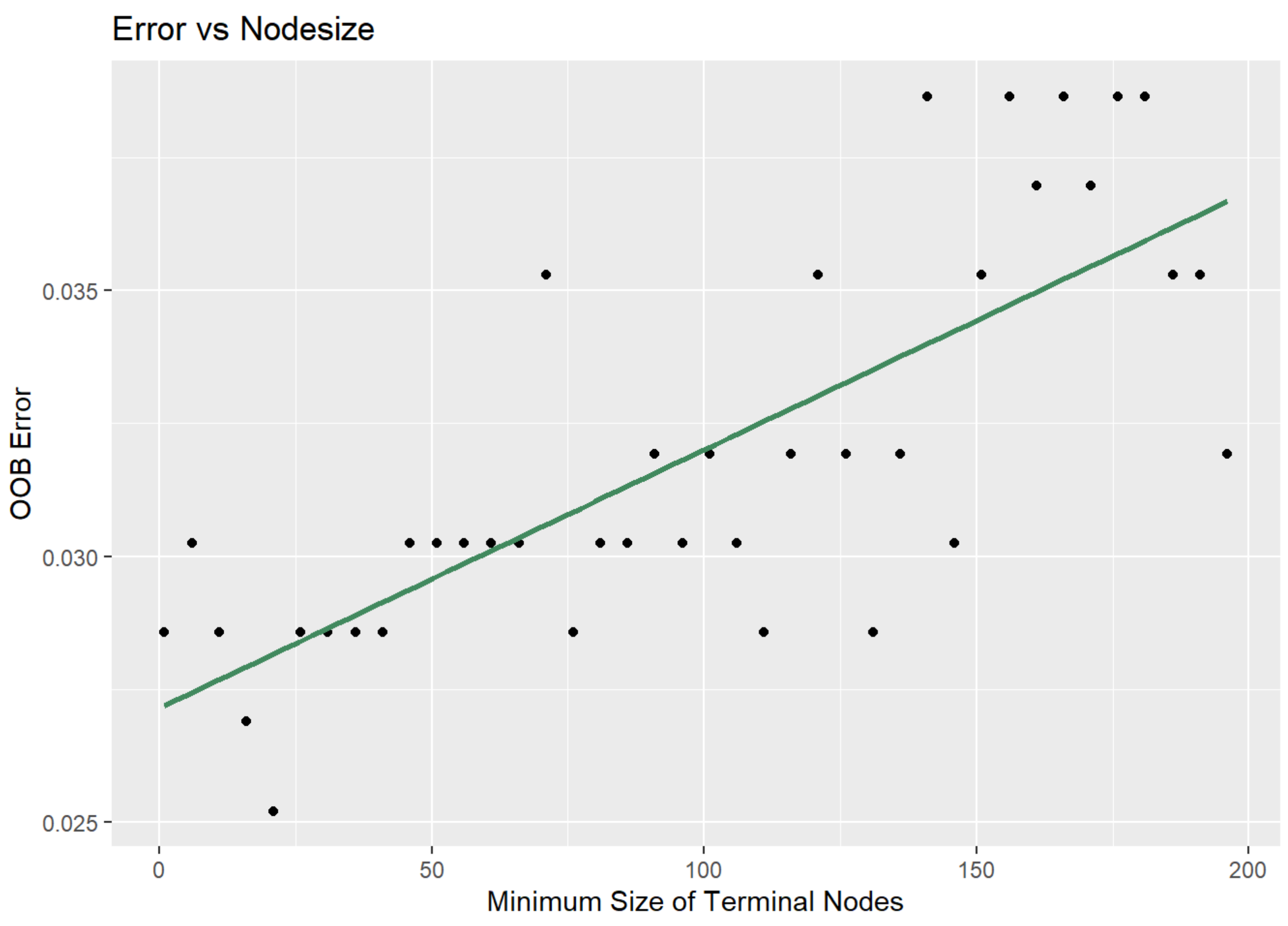
Another parameter that can be tweaked in the randomForest function is 'nodesize'. This represents the minimum number of observations that must exist at each terminal node in each tree. Therefore, higher values of nodesize lead to shallower trees (so nodesize is a generalised way of pruning trees in a random forest).

Parameter C: nodesize

Lets see how OOB error changes as we increase nodesize:

```
set.seed(14)
x=seq(1,200,by=5)
node = vector(length=length(x))
for(i in 1:length(x)) {
  temp.model = randomForest(Class ~ ., data=bcwctrain, ntree=1000,mtry=2,nodesize=x[i]) # cre
ate a model for each increment of nodesize by 5
  node[i] = temp.model$err.rate[nrow(temp.model$err.rate),1] # OOB error rate
}

val2=data.frame(nodesize=x,error=node) # data frame for call to ggplot2

# Scatter plot with linear line of best fit
ggplot(val2,aes(x=nodesize,y=error))+
  geom_point()+geom_smooth(method = "lm",se=FALSE,color="#41895e")+scale_y_continuous(breaks
 = scales::pretty_breaks(n = 5))+scale_x_continuous(breaks = scales::pretty_breaks(n = 5))+la
bs(title="Error vs Nodesize",y="OOB Error",x="Minimum Size of Terminal Nodes")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



We see that there is a positive correlation between nodesize and OOB error, which implies that choosing a lower value for nodesize is best.

For this reason, we will stick with the default nodesize value of 1.

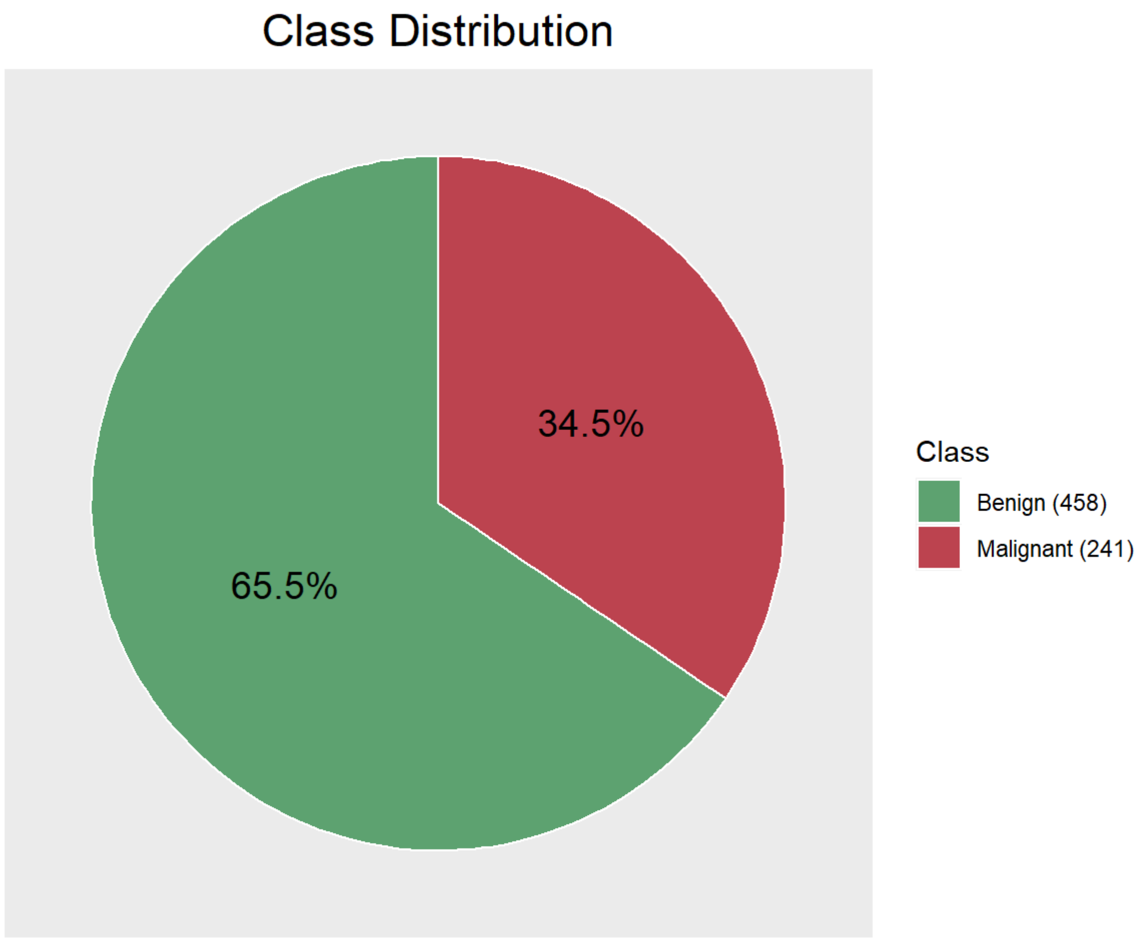
One explanation of why increasing nodesize is not beneficial is that pruning is necessary for single decision trees to prevent overfitting, but with a random forest overfitting is already prevented by building a large number of decision trees. Therefore no gains can be made from pruning individual trees with the nodesize parameter, whilst limiting the growth of the trees may negatively affect the performance of these trees [17].

Class Distribution:

It is worth noting that the Breast Cancer Wisconsin dataset is imbalanced. 458 (66%) of observations are benign whilst just 241 (34%) of observations are malignant.

Here is a pie chart to visualise this disparity between classes (this code block is based on [14]):

```
bcwcalt=bcwc
bcwcalt$Class=ifelse(bcwc$Class=="Benign","Benign (451)","Malignant (241)") # create a copy o
f the dataset with labels to indicate the raw number of observations in each class


x=c(round(sum(bcwc$Class=="Benign")/699*100,1),round(sum(bcwc$Class=="Malignant")/699*100,1))
# store the percentage of data that is benign and malignant in a vector
labels=c("Benign (458)","Malignant (241)") # labels to indicate the raw number of observation
s in each class

piedata=data.frame(label=labels,count=x) # data frame to be called using ggplot

ypos=cumsum(x)-0.5*x
ypos=100-ypos # calculate position of labels to go in pie chart

# Plot pie chart (note that ggplot2 doesn't have a pie geom so polar coordinates are used ins
tead)
g=ggplot(piedata)+geom_bar(aes(x="",y=count,fill=label),stat="identity",color="white")+coord_
polar("y",start=0)+theme(axis.title=element_blank(),axis.text=element_blank(),axis.ticks=elem
ent_blank(),panel.grid=element_blank(),panel.border=element_blank())+  scale_fill_manual(valu
es=c("#5da270","#bc434f"),name="Class")
g1=g+ggtitle("Class Distribution")+theme(plot.title=element_text(hjust=0.5,size=17))

g2=g1+geom_text(aes(x="",y=ypos,label=paste0(count,"%")),size=5)
g2
```



The frequency imbalance between classes may be a problem because it means that the model will be trained to be more effective at diagnosing benign tumours than malignant tumours. And it is arguably worse to wrongly diagnose malignant tumours as benign than vice versa because of the severe consequences of late diagnosis of breast cancer. Therefore, if anything we would prefer our model to be more effective at diagnosing malignant tumours than benign tumours.

Parameter D: sampsize (& oversampling)

Oversampling:

Oversampling is one method that can be used to correct this frequency imbalance. It randomly repeats observations of the rarer class (malignant) until it has an equal number of observations as the other class (benign) [7].

```
set.seed(15)
over=ovun.sample(Class~.,data=bcwctrain, # generate our oversampled dataset
                 method="over",N=780)$data
# 390*2=780 -> there are 390 benign observations in the training set so we want 390 malignant
obervations too using oversampling, giving 780 in total
table(over$Class) # check that there are an equal number of benign and malignant observations
in the oversampled dataset
```

```
## 
##    Benign Malignant 
##       390       390
```

We see that oversampling has successfully equalised the number of benign and malignant observations in the training dataset.

Now let’s try to build a model using the oversampled dataset and the optimal parameters identified earlier:

```
set.seed(16)
overmodel=randomForest(Class ~ ., data=over, mtry=2, ntree=1000) # random forest model using
 oversampled dataset
overmodel
```

```
## 
## Call:
##  randomForest(formula = Class ~ ., data = over, mtry = 2, ntree = 1000) 
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 2
## 
##         OOB estimate of  error rate: 1.67%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       378        12 0.030769231
## Malignant      1       389 0.002564103
```

```
optimodel
```

```
## 
## Call:
##  randomForest(formula = Class ~ ., data = bcwctrain, mtry = 2,      ntree = 1000) 
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 2
## 
##         OOB estimate of  error rate: 2.69%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       381         9  0.02307692
## Malignant      7       198  0.03414634
```

If we look at our best model using the original training dataset, we see that sensitivity is 96.6% and specificity is 97.7%, so it appears that the data imbalance is only making a slight difference of 1.1% which may not be statistically significant. So the relative shortage of malignant tumours may not be negatively affecting our model's ability to detect breast cancer.

However, using oversampling allows us to improve the sensitivity at the cost of slightly reduced specificity (a fair trade considering we care about not missing cases of breast cancer). Even better, we manage to increase sensitivity (now 99.7%) by considerably more than specificity decreases (now 96.9%). Therefore, oversampling has yielded desirable results.

Let's see if these results carry through to the testing dataset:

```
# Oversampled model
predictions=predict(overmodel,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##            predictions
## actual      Benign Malignant
##   Benign        66         2
##   Malignant      2        34
```

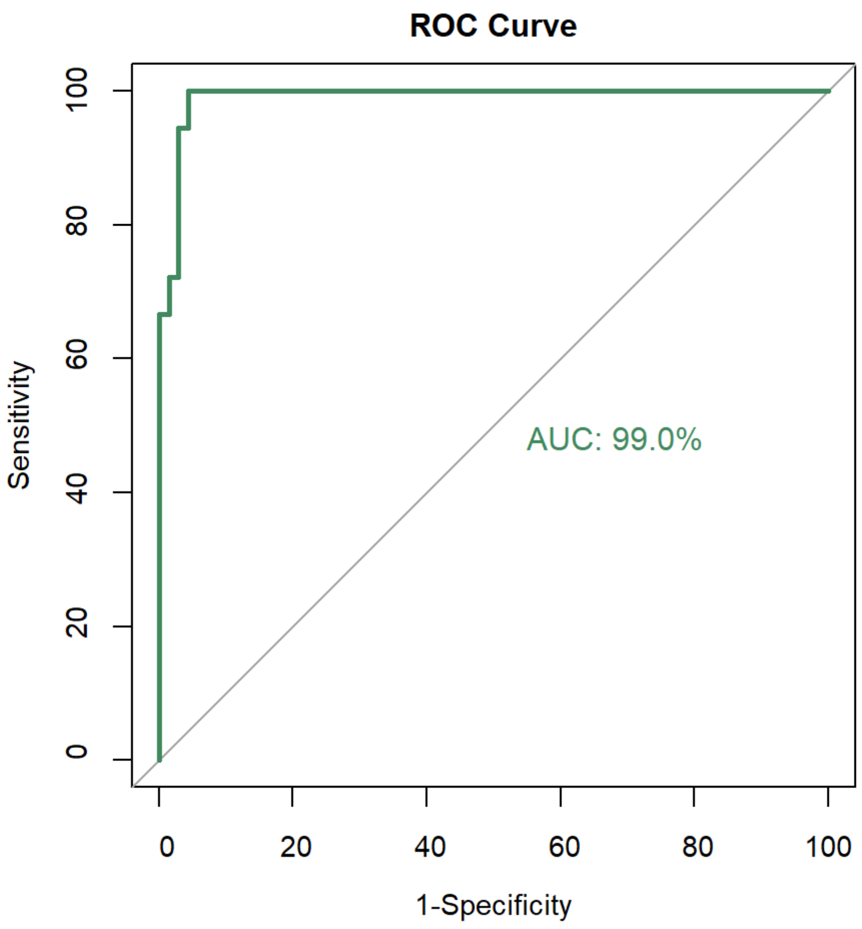
```
# Original model
predictions=predict(optimodel,bcwctest)
confusion2=table(actual=bcwctest$Class,predictions=predictions)
confusion2
```

```
##            predictions
## actual      Benign Malignant
##   Benign        66         2
##   Malignant      2        34
```

Unfortunately, the benefits of oversampling have not carried through to the testing dataset since the confusion matrix is identical to before. However, this may simply be due to the relatively small size of the testing dataset, and we would observe some gains to sensitivity if we had more data.

Here is an ROC curve for the oversampled model:

```
probpred=predict(overmodel,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC Curve",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC Curve", cex.main = 1, cex.
axis = 0.9,     cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 99.02%
```

```
par(pty="m")
```

Unsurprisingly, the ROC curve is almost identical to before as well.

Stratified sampling (sampsize):

Stratified sampling is an alternative to oversampling that also aims to achieve a greater representation of malignant samples in the model given their relative shortage. This is achieved by stratifying each bootstrapped dataset to contain the same amount of benign and malignant observations.

There are 205 malignant observations in total so each bootstrapped dataset will be set to contain 205 benign and 205 malignant observations using the sampsize parameter in the randomForest function:

```
set.seed(17)
stratified=randomForest(Class~.,data=bcwctrain,ntree=1000,mtry=2,sampsize=c(205,205)) # build
ing model with stratified bootstraps each containing 205 benign and 205 malignant observation
s
stratified
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = bcwctrain, ntree = 1000,      mtry = 2, sampsize
= c(205, 205))
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 2.69%
## Confusion matrix:
##           Benign Malignant class.error
## Benign       377        13  0.03333333
## Malignant      3       202  0.01463415
```

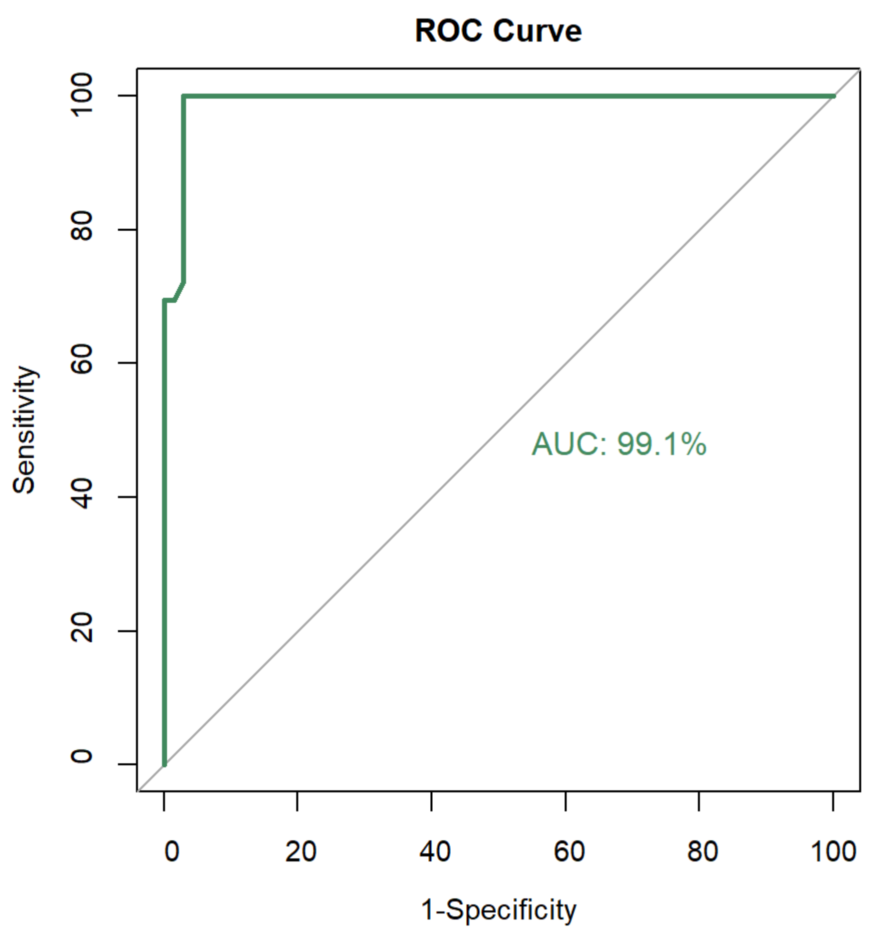
We see that using stratified bootstrapped datasets also increase sensitivity, however this increase is less pronounced than with oversampling (sensitivity increases from 96.6% to 98.5% rather than to 99.7%). Unsurprisingly this means that there is still no change to the performance on the testing dataset as we see below:

```
# Stratified model
predictions=predict(stratified,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##            predictions
## actual      Benign Malignant
##   Benign        66         2
##   Malignant      2        34
```

The ROC curve for the stratified model shows the highest AUC we have seen so far at 99.1% (although this is still almost the same as all the other models):

```
probpred=predict(stratified,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC Curve",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC Curve", cex.main = 1, cex.
axis = 0.9,     cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 99.12%
```

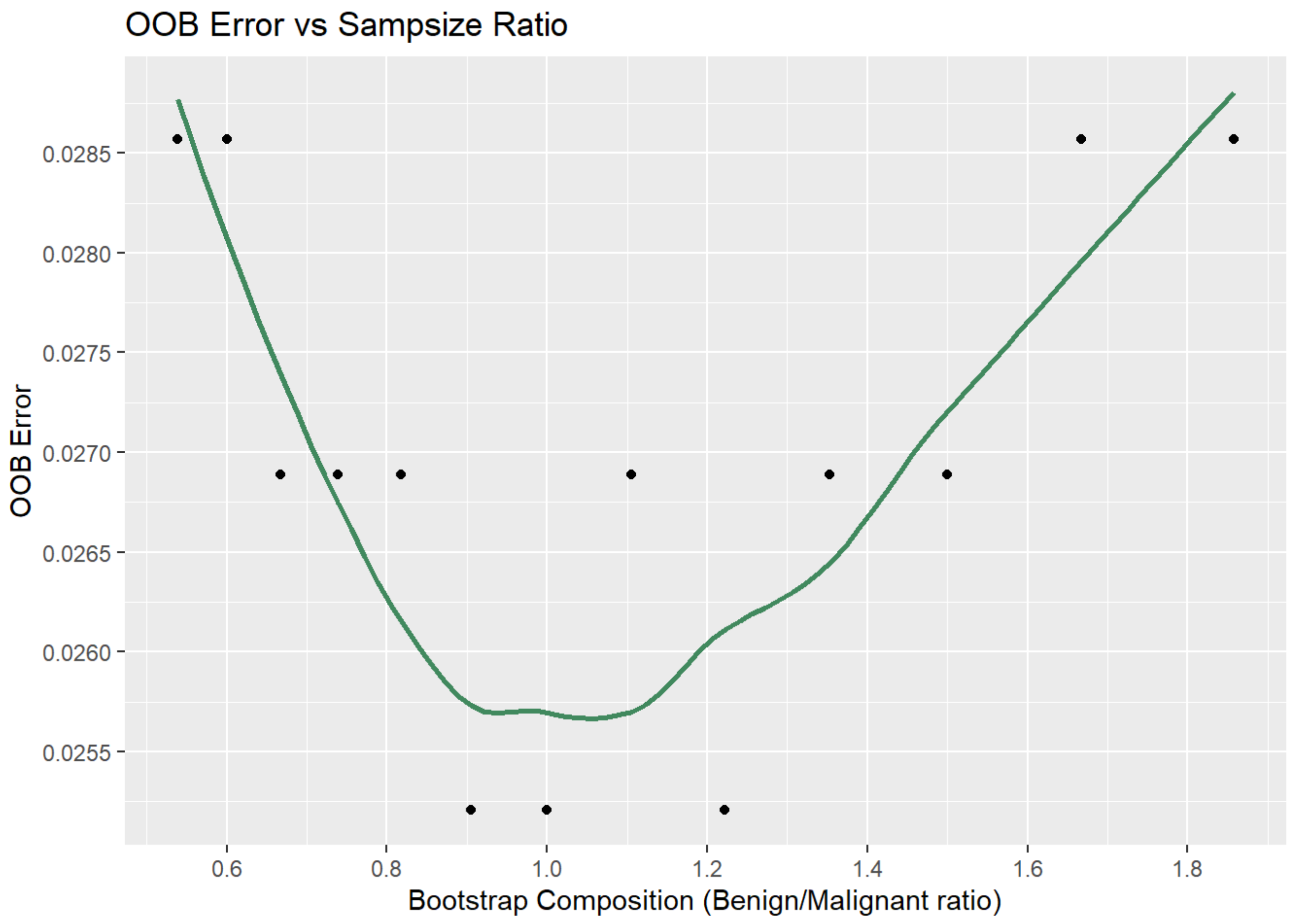
```
par(pty="m")
```

Perhaps it is not best to take equal amounts of benign and malignant observations in stratified bootstraps since this under-represents the more plentiful benign observations. Instead, we could potentially strike a balance between emphasising malignant observations whilst also making good use of benign observations.

Let's try this out by varying the ratio of benign to malignant observations in each bootstrapped dataset from around 0.5 up to around 1.9 (the natural ratio in the data is just under two times as many benign as malignant observations):

```
set.seed(18)
s=seq(-0.3,0.3,by=0.05)
node = vector(length=length(s))

for(i in 1:length(s)) {
  temp.model=randomForest(Class ~ ., data=bcwctrain, mtry=2, ntree=1000,sampsize=c(157*(1+s
[i]),157*(1-s[i]))) # as s[i] increases with i, the benign sample size gets multiplied whilst
the malignant sample size gets diminished
  node[i] = temp.model$err.rate[nrow(temp.model$err.rate),1] # store OOB error rate
}

s.ratio=(1+s)/(1-s) # convert each value of s to its corresponding ratio of benign to maligna
nt observations
val2=data.frame(strata=s.ratio,error=node) # store data in a data frame for use in the ggplot

# Scatter graph with linear line of best fit
ggplot(val2,aes(x=strata,y=error))+
  geom_point()+geom_smooth(se=FALSE,color="#41895e")+
scale_y_continuous(breaks = scales::pretty_breaks(n = 6))+
  scale_x_continuous(breaks = scales::pretty_breaks(n = 6))+
  labs(title="OOB Error vs Sampsize Ratio",y="OOB Error",x="Bootstrap Composition (Benign/Mal
ignant ratio)")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



We see that more evenly distributed bootstrapped datasets tend to be associated with lower OOB error rates. Therefore, it seems best to stick bootstrapped datasets that are stratified to contain the same amount of benign and malignant observations (i.e. a ratio of 1).

Parameter E: cutoff

A final parameter that can be experimented with is the cutoff rate. This is a value between 0 and 1 that is used for determining whether to classify an observation as benign or malignant. For example, with the default cutoff of 0.5, if over 50% of the decision trees in the random forest classify an observation as malignant its final classification will be malignant (otherwise its final classification will be benign).

However, the cutoff can be moved away from the default value. Cutoffs lower than 0.5 will make it easier to classify tumours as malignant (since less than 50% of the votes are needed) but make it more difficult to classify tumours as benign (since over 50% of the votes are needed). This would boost the sensitivity of the model at the cost of reducing the specificity of the model. Cutoffs higher than 0.5 will do the opposite (make it harder to classify tumours as malignant).

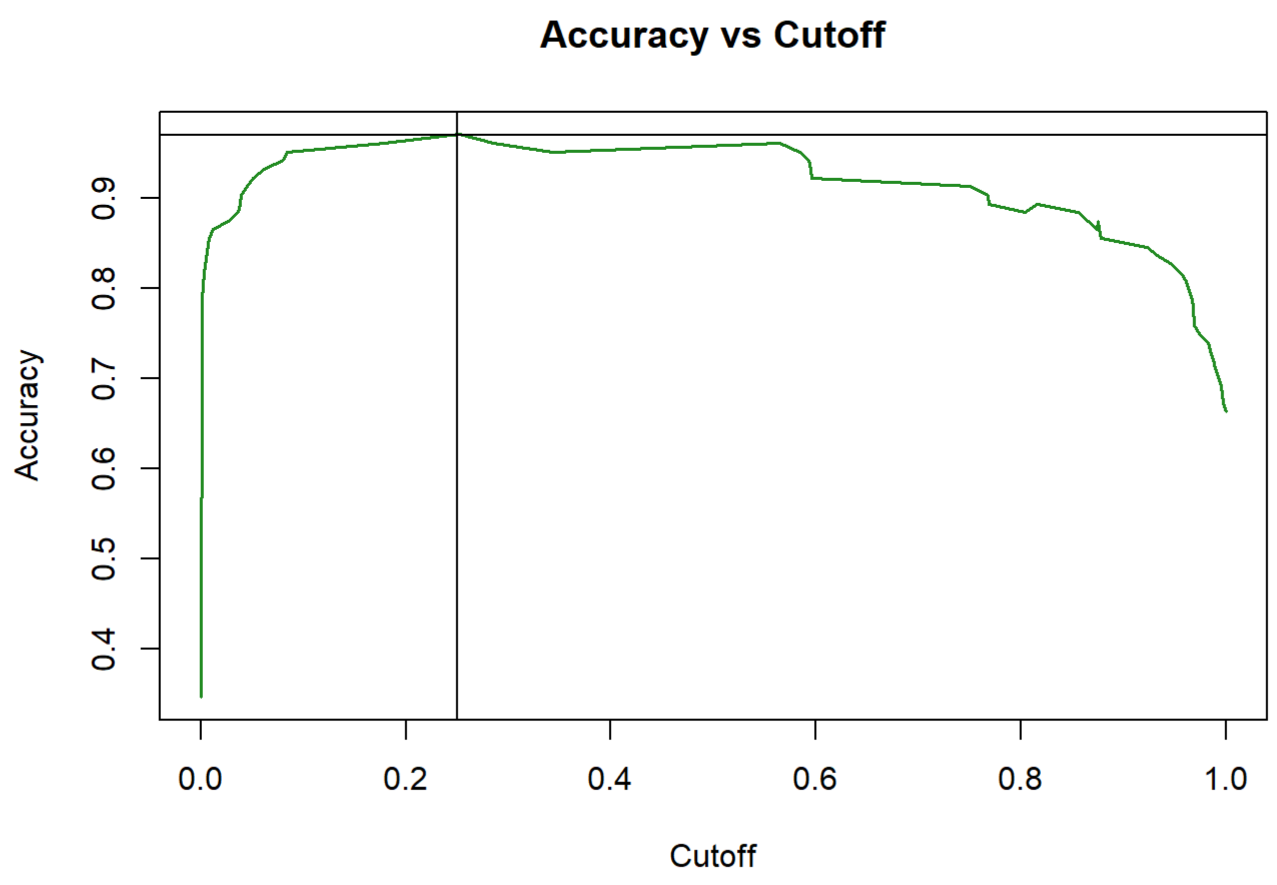
Incidentally, different cutoff values are used to construct an ROC curve, so tuning this parameter amounts to choosing the optimal point on the ROC curve.

We care greatly about sensitivity since we do not want to miss cases of breast cancer, so it could potentially be beneficial to select a cutoff lower than the default of 0.5.

The following graph shows the relationship between cutoff point and accuracy on the testing dataset (using the optimised model with no oversampling or stratified bootstraps):

This code segment is based on [9]

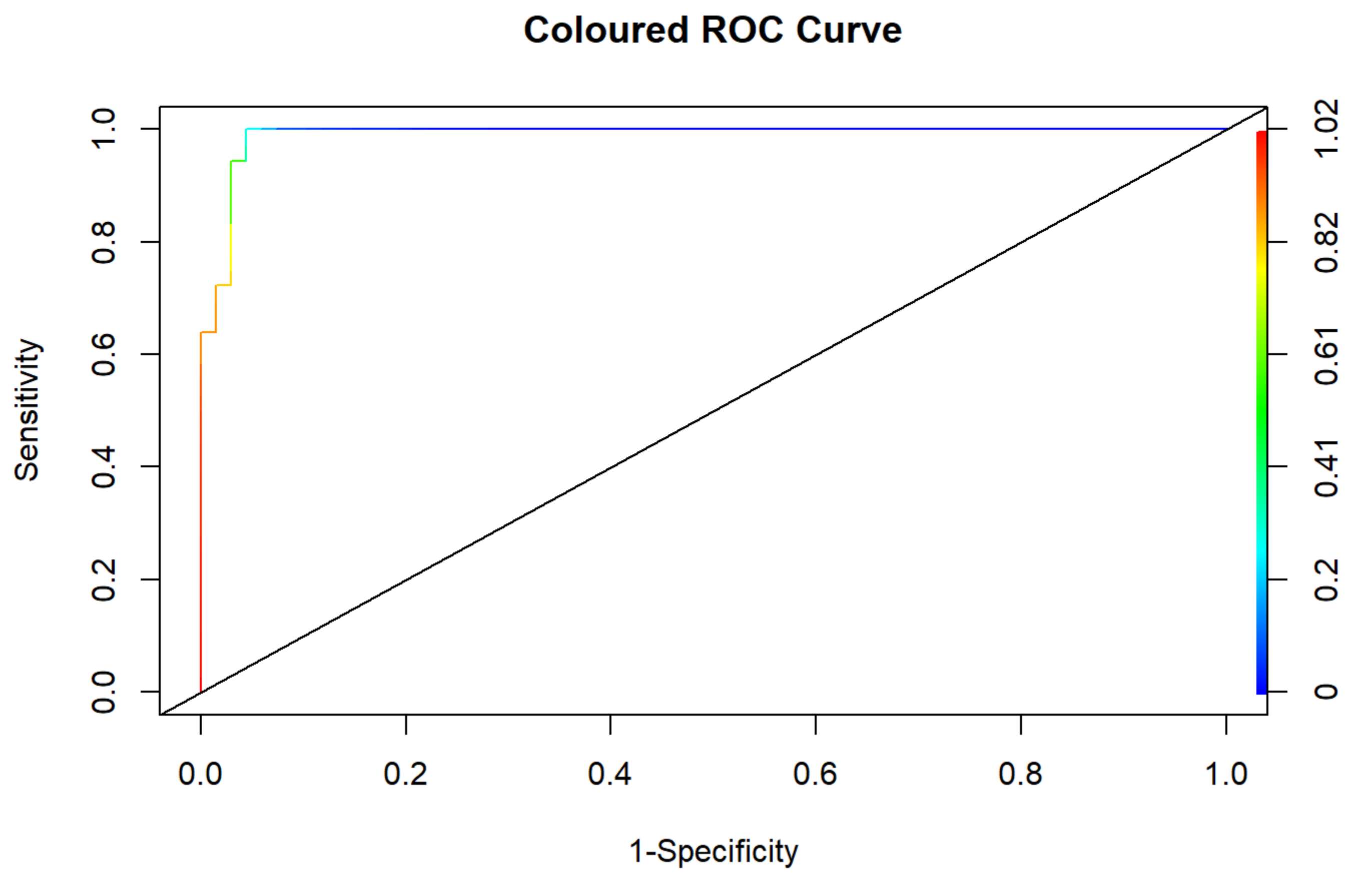
```
pred=predict(optimodel,bcwctest,type="prob")[,2] # generate the predictions and store the per
centage of trees that voted each observation as malignant
pred=prediction(pred,bcwctest$Class) # ROCR function that transforms the data format
eval=performance(pred,"acc") # ROCR function that stores the cutoff and accuracy data
plot(eval,col="forestgreen",main="Accuracy vs Cutoff",lwd=1.5) # Base R line graph
abline(v=0.25,h=0.971) # vertical and horizontal line to emphasise the cutoff point giving th
e highest accuracy
```



We see that extreme cutoff values at both tails are associated with lower accuracy levels, but a wide range of moderate cutoff levels are able to achieve similar accuracy levels. This supports the possibility of choosing a cutoff lower than 0.5 to boost sensitivity. In fact, the highest accuracy here is achieved at a cutoff of about 0.25, considerably lower than 0.5.

Here is a colour-coded ROC curve that shows how different cutoff values correspond to different points on the curve. As expected, lower cutoff values (coded by green and blue) correspond to higher sensitivities and lower specificities (so higher values for 1-specificity).

```
roc=performance(pred, "tpr", "fpr") # ROCR function to store the data needed to plot the ROC
 curve

# Plot coloured ROC curve (triggered by colorize parameter)
plot(roc,colorize=T,main="Coloured ROC Curve",
     ylab="Sensitivity",xlab="1-Specificity")
abline(a=0,b=1) # random guesses
```



All in all, we care about both sensitivity and accuracy. Although we do not wish to miss cases of breast cancer, we also do not wish to cause people unnecessary worry by incorrectly diagnosing them with cancer.

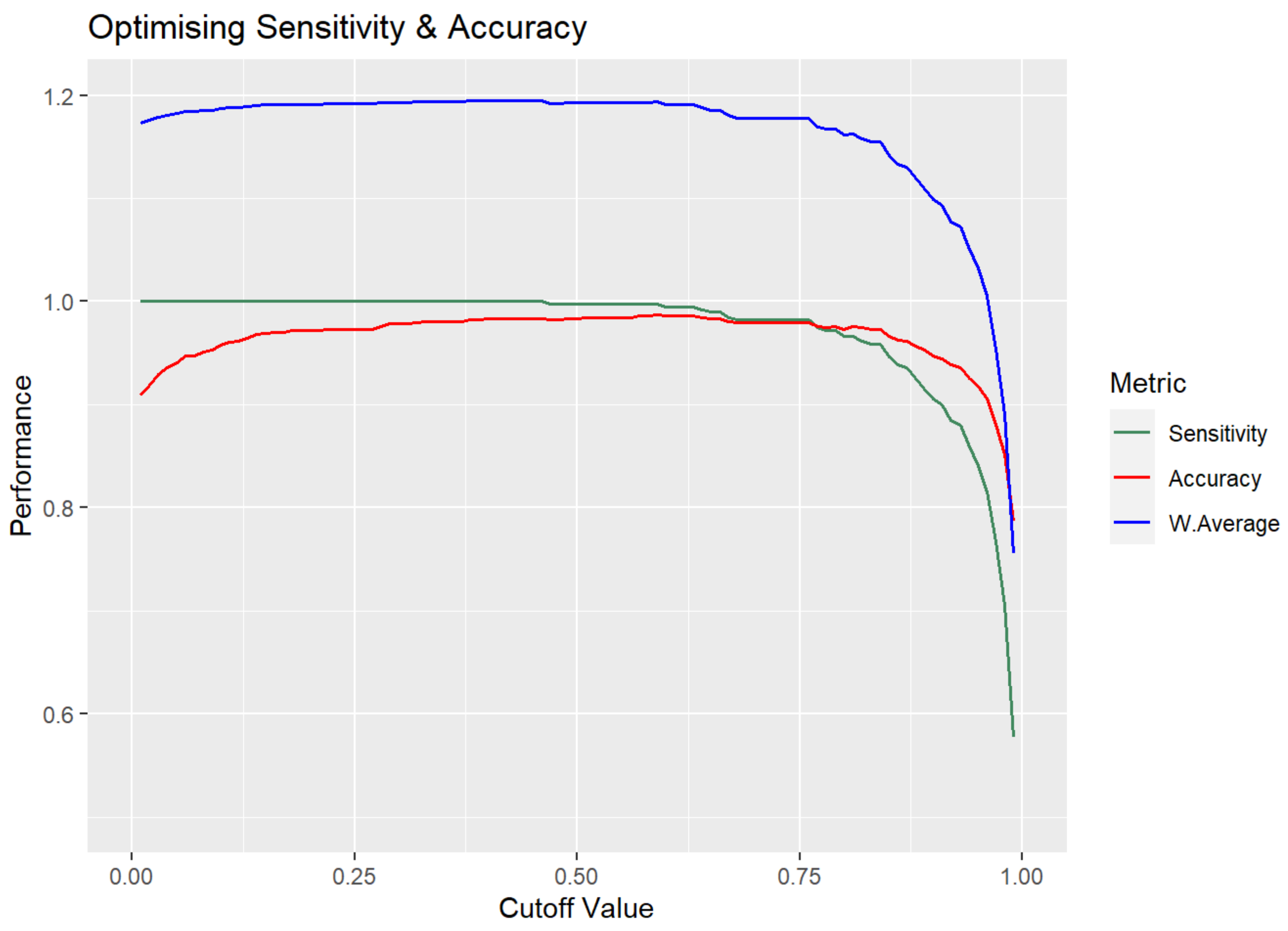
Therefore it makes sense to choose a model that maximises a weighted average of accuracy and sensitivity.

Since we arguably care most about sensitivity, we will select a weighting of 75% for sensitivity and 25% for accuracy.

Note that accuracy is already a function of sensitivity so an alternative approach would have been to take a weighted average of sensitivity and specificity.

Here we graph the relationship between cutoff rate, accuracy, sensitivity, and our composite performance metric (weighted average) for the oversampled model:

```r
x=overmodel$votes
cutoffs=seq(0.01,0.99,by=0.01) # vary cutoff by values of 0.01
storage=matrix(ncol=4,nrow=length(cutoffs))

for(i in 1:length(cutoffs))
{
  classify=ifelse(x[,2]>cutoffs[i],"Malignant","Benign") # generate classifications from the
 raw vote data based on increasing cutoff values
  cm=table(actual=over$Class,predictions=classify) # create confusion matrix so that performa
nce measures can be calculated from this
  storage[i,1]=cutoffs[i] # cutoff value
  storage[i,2]=cm[2,2]/sum(cm[2,]) # sensitivity
  storage[i,3]=sum(diag(cm))/sum(cm) # accuracy
  storage[i,4]=1.2*(0.75*storage[i,2]+0.25*storage[i,3]) # weighted average of sensitivity an
d accuracy (75:25 weighting) (scaled by 1.2 to make it more visible on the graph)
}

# Create data frame for use in the call to ggplot
storage=data.frame(storage)
names(storage)=c("cutoff","sensitivity","accuracy","composite")

# Multi-line plot
ggplot(storage,aes(x=cutoff))+
  geom_line(aes(y=sensitivity,colour="Sensitivity"),size=0.7)+
  geom_line(aes(y=accuracy,colour="Accuracy"),size=0.7)+
  geom_line(aes(y=composite,colour="W.Average"),size=0.7)+
  scale_colour_manual("Metric", breaks = c("Sensitivity", "Accuracy", "W.Average"),
  values = c("#41895e", "red", "blue")) +xlim(0,1)+ ylim(0.5,1.2)+
  labs(title="Optimising Sensitivity & Accuracy",y="Performance",x="Cutoff Value")
```



Sensitivity is a decreasing function of cutoff rate because higher cutoff means more votes are needed to classify an observation as malignant.

Therefore, the composite performance metric (which is a function of sensitivity) is maximised at a lower cutoff than accuracy. This encourages us to use lower cutoff values in our random forest models.

```
storage[which.max(storage$composite),] # finds the (first) row in the storage table that give
s the maximum value of the composite performance metric
```

	cutoff <dbl>	sensitivity <dbl>	accuracy <dbl>	composite <dbl>
40	0.4	1	0.9833333	1.195

1 row

By selecting the point on the graph where the composite metric is maximised, we choose a cutoff of 0.4 (lower than default of 0.5).

```
x=overmodel$votes
classify=ifelse(x[,2]>0.4,"Malignant","Benign")
cm=table(actual=over$Class,predictions=classify)
cm
```

```
##            predictions
## actual       Benign Malignant
##   Benign        377        13
##   Malignant       0       390
```

By choosing 0.4 as the new cutoff, no malignant tumours are missed when looking at the out of bag data (we obtain perfect sensitivity)

We can permanently embed this new cutoff within the model using the cutoff parameter in the randomForest function:

```
set.seed(19)
finalmodel=randomForest(Class ~ ., data=over, mtry=2, ntree=1000,cutoff=c(0.6,0.4)) # Create
 a new model with a 0.4 cutoff value
```

Now let’s see how this new model performs on the testing dataset:

```
predictions=predict(finalmodel,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##            predictions
## actual      Benign Malignant
##   Benign        65         3
##   Malignant      2        34
```

Unfortunately no progress is made with sensitivity and specificity is slightly worse than the 0.5 cutoff, so we acutally see lower accuracy overall.

This may be because the lower cutoff was chosen using the oversampled dataset which naturally yields models with higher sensitivity. Therefore, we did not have to lower the cutoff very far below 0.5 before perfect OOB sensitivity was achieved, so the advantage of reducing the cutoff further below 0.4 was not registered.

This shouldn’t be a problem when finding the optimal cutoff for the model trained on the regular training dataset so let’s try this process again:

```
x=optimodel$votes
cutoffs=seq(0.01,0.99,by=0.01)
storage=matrix(ncol=4,nrow=length(cutoffs))

for(i in 1:length(cutoffs))
{
  classify=ifelse(x[,2]>cutoffs[i],"Malignant","Benign")
  cm=table(actual=bcwctrain$Class,predictions=classify)
  storage[i,1]=cutoffs[i] # cutoff
  storage[i,2]=cm[2,2]/sum(cm[2,]) # sensitivity
  storage[i,3]=sum(diag(cm))/sum(cm) # accuracy
  storage[i,4]=1.2*(0.75*storage[i,2]+0.25*storage[i,3]) # weighted average of sensitivity an
d accuracy (75:25 weighting) (scaled by 1.2 to make it more visible on the graph)
}

storage=data.frame(storage)
names(storage)=c("cutoff","sensitivity","accuracy","composite")

storage[which.max(storage$composite),]
```

	cutoff <dbl>	sensitivity <dbl>	accuracy <dbl>	composite <dbl>
22	0.22	0.995122	0.9630252	1.184517

1 row

As expected, the optimal cutoff this time is 0.22 (much lower than 0.4 from before) because perfect sensitivity was not achieved so easily.

Let’s see how a model with a cutoff of 0.22 performs on the testing dataset:

```
set.seed(20)
finalmodel2=randomForest(Class ~ ., data=bcwctrain, mtry=2, ntree=1000,cutoff=c(0.78,0.22))
predictions=predict(finalmodel2,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##             predictions
## actual       Benign Malignant
##   Benign         65         3
##   Malignant       0        36
```

This time we successfully achieve perfect sensitivity on the testing dataset.

Let’s take an average of the two cutoffs, 0.4 and 0.22, to give a final cutoff of 0.31. The reason for this is that perfect sensitivity was already achieved on the training data at a cutoff of 0.4 for the oversampled dataset, so it is likely that a slightly lower cutoff would have been chosen if more data was available, however, the optimal cutoff would likely be higher than 0.22 (the cutoff for the regular training dataset) because the oversampled dataset already favours high sensitivity by nature of the oversampling.

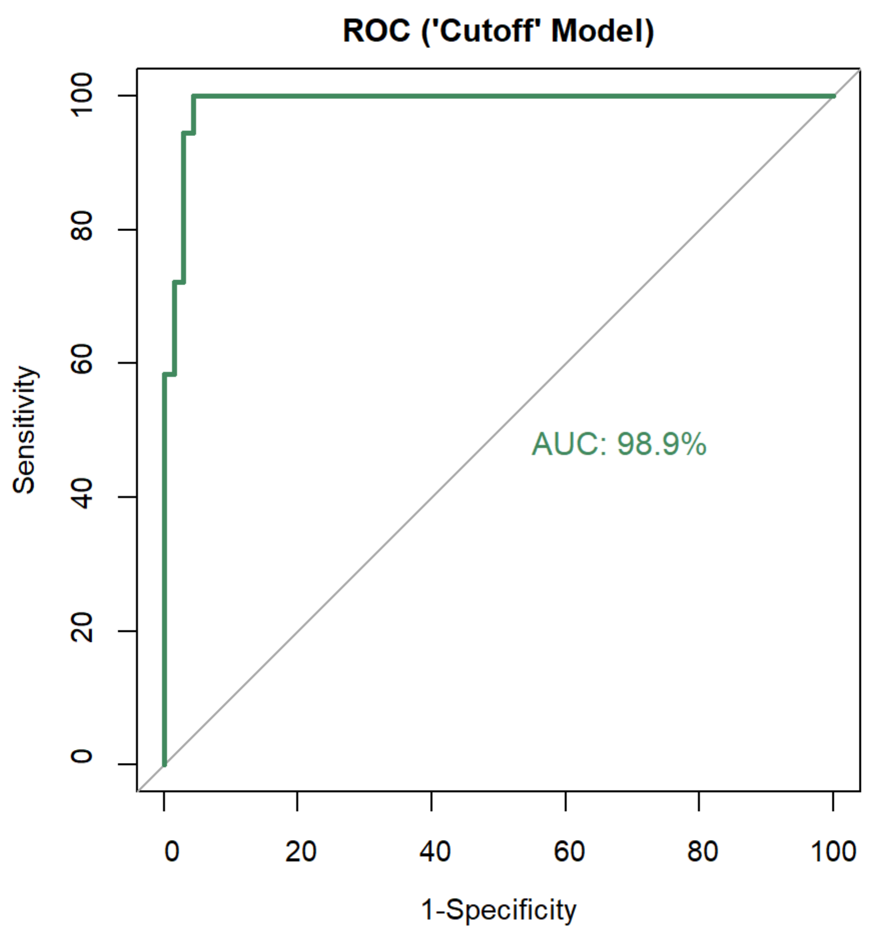
```
set.seed(21)
finalmodel3=randomForest(Class ~ ., data=over, mtry=2, ntree=1000,cutoff=c(0.69,0.31))
predictions=predict(finalmodel3,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##             predictions
## actual       Benign Malignant
##   Benign         65         3
##   Malignant       1        35
```

Using the oversampled dataset and a cutoff of 0.31 we achieve one fewer false negative (1 instead of 2) at the cost of one additional false positive (3 instead of 2). So sensitivity is improved on the testing dataset but overall accuracy remains the same.

Here is an ROC curve for this model:

```
probpred=predict(finalmodel3,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC ('Cutoff' Model)",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC ('Cutoff' Model)", cex.mai
n = 1,     cex.axis = 0.9, cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 98.9%
```

```
par(pty="m")
```

AUC is slightly lower than for the oversampled model with a cutoff of 0.5 (98.9% instead of 99.0%) but this difference is negligible.

8. Simultaneous Optimisation

So far, the parameters in the random forest model have been optimised sequentially rather than simultaneously. But arguably we can only truly optimise our model by tuning all parameters simultaneously rather than sequentially since better performance with respect to one parameter doesn't necessarily mean better performance when changing another parameter on top of the previous parameter.

For this simultaneous optimisation process, the number of trees will be fixed at 500 because we have shown that changing ntree does not affect model performance provided that ntree is high enough (and we are comparing a very large number of models so computation speed is important here).

The three parameters that will therefore be optimised at the same time are: - mtry (the number of variables available to choose from at each node in each tree) - sampsize (the stratified sampling parameter - this is used instead of oversampling so the same dataset can be used over all parameters) - cutoff (the proportion above which observations are classified as malignant)

```
set.seed(22)

mtryval=seq(1,5,by=1) # increment mtry by values of 1
sampsizefactor=seq(0,0.5,by=0.05) # increment stratified sample multiplier by values of 0.05
cutoffval=seq(0.1,0.6,by=0.01) # increment cutoff values by 0.01

x=length(mtryval)
y=length(sampsizefactor)
z=length(cutoffval)

storage=matrix(ncol=6,nrow=x*y*z) # create a matrix to store the data in
colnames(storage)=c("sampsizefactor","cutoff","mtryval","sensitivity","accuracy","composite")

for(i in 1:y)
{
  for(j in 1:z)
  {
    for(k in 1:x)
    {
      temp.model=randomForest(Class ~ ., data=bcwctrain, mtry=k, ntree=500,cutoff=c(1-cutoffv
al[j],cutoffval[j]),sampsize=c(205*(1+sampsizefactor[i]),205*(1-sampsizefactor[i]))) # build
 model whilst changing one of the three free parameters with each iteration
      cm=table(actual=bcwctrain$Class,predictions=temp.model$predicted) # create a confusion
 matrix to calculate the performance metrics from
      storage[x*z*(i-1)+x*(j-1)+k,1]=sampsizefactor[i] # store sampsizefactor value correspon
ding to the model created
      storage[x*z*(i-1)+x*(j-1)+k,2]=cutoffval[j] # store cutoff value corresponding to the m
odel created
      storage[x*z*(i-1)+x*(j-1)+k,3]=mtryval[k] # store mtry value corresponding to the model
created
      storage[x*z*(i-1)+x*(j-1)+k,4]=cm[2,2]/sum(cm[2,]) # sensitivity
      storage[x*z*(i-1)+x*(j-1)+k,5]=sum(diag(cm))/sum(cm) # accuracy
      storage[x*z*(i-1)+x*(j-1)+k,6]=1.2*(0.75*storage[x*z*(i-1)+x*(j-1)+k,4]+0.25*storage[x*
z*(i-1)+x*(j-1)+k,5]) # # weighted average of sensitivity and accuracy (75:25 weighting)
      # print(x*z*(i-1)+x*(j-1)+k,1) # prints the row of the matrix just recorded (a way of e
nsuring that this code block is running smoothly considering that it can take up to ten minut
es to finish running)
    }
  }
}
storage=as.data.frame(storage) # convert matrix to data frame so it can be used in ggplot
storage[storage$composite==storage[which.max(storage$composite),6],] # record (all instances
 of) rows with the highest composite metric scores
```

	sampsizefactor <dbl>	cutoff <dbl>	mtryval <dbl>	sensitivity <dbl>	accuracy <dbl>	composite <dbl>
167	0	0.43	2	0.995122	0.9764706	1.188551

1 row

After simultaneously tuning all three parameters we find our optimal model has mtry=2, cutoff=0.43 and 50-50 stratified bootstrap samples

Let’s create this model with 1000 trees this time and apply it to the testing dataset:

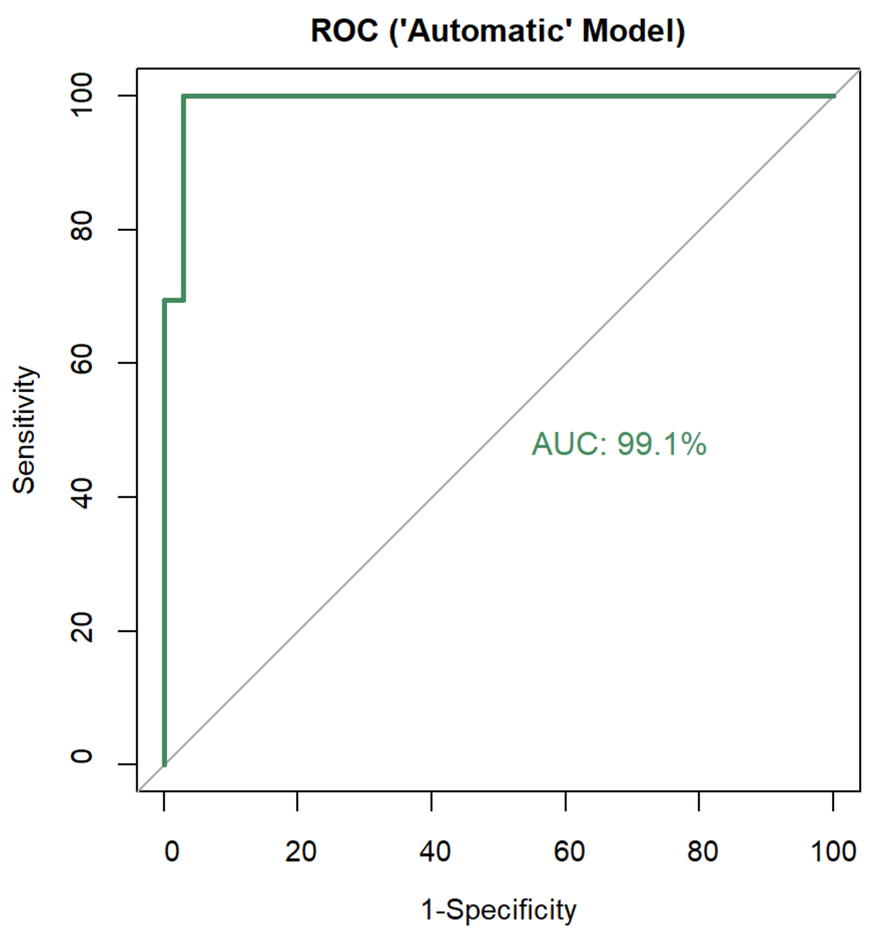
```
set.seed(23)
automodel=randomForest(Class ~ ., data=bcwctrain, mtry=2, ntree=1000,cutoff=c(0.57,0.43),samp
size=c(205,205)) # model with the specification listed above
predictions=predict(automodel,bcwctest)
confusion1=table(actual=bcwctest$Class,predictions=predictions)
confusion1
```

```
##            predictions
## actual      Benign Malignant
##   Benign        66         2
##   Malignant      0        36
```

Out of all the models we have tried, this one is the best performing on the testing dataset with 100% sensitivity and 98.1% accuracy.

Now let’s draw an ROC curve for the model:

```
probpred=predict(automodel,bcwctest,type="prob")
par(pty="s")
roc(response=bcwctest$Class,predictor=probpred[,2],
    plot=TRUE,legacy.axes=TRUE,percent=TRUE,
    xlab="1-Specificity",ylab="Sensitivity",
    col="#41895e",lwd=2.5,print.auc=TRUE,
    print.auc.x=45,main="ROC ('Automatic' Model)",cex.main=1,cex.axis=0.9,cex.lab=0.9)
```



```
##
## Call:
## roc.default(response = bcwctest$Class, predictor = probpred[,     2], percent = TRUE, plot
= TRUE, legacy.axes = TRUE, xlab = "1-Specificity",     ylab = "Sensitivity", col = "#41895
e", lwd = 2.5, print.auc = TRUE,     print.auc.x = 45, main = "ROC ('Automatic' Model)", cex.
main = 1,     cex.axis = 0.9, cex.lab = 0.9)
##
## Data: probpred[, 2] in 68 controls (bcwctest$Class Benign) < 36 cases (bcwctest$Class Mali
gnant).
## Area under the curve: 99.1%
```

```
par(pty="m")
```

This model is tied with the stratified model for the highest AUC value at 99.1% (although the lowest value we have seen is 98.9%) so clearly not too much separates the general performance of all the models in this analysis.

9. Model Comparison

To conclude let's compare the performance of the key random forest models that have been created. In order to create comparative graphs, we first need to gather all the results we care about into a single data frame:

```
data=vector(length=0)
model=vector(length=0)
accuracy=vector(length=0)
sensitivity=vector(length=0)

comparison=data.frame(data,model,accuracy,sensitivity) # data frame to store our model perfor
mance data in

# Raw models and confusion matrices
dvalues=c("OOB","Test")
mvalues=c("Default","Original","CV","Oversample","Cutoff","Automatic")
mstore=list(default2,optimodel,cvmodel,overmodel,finalmodel3,automodel)
mconfusion=list(default2$confusion,optimodel$confusion,cvmodel$confusion,overmodel$confusion,
finalmodel3$confusion,automodel$confusion)

# optimodel: original model (only ntree and mtry optimised sequentially)
# automodel: automated full optimisation (ntree, mtry, cutoff, sampsize all optimised simulta
neously)
# finalmodel3: manual full optimisation (ntree, mtry, cutoff optimised # sequentially and ove
rsampling is used)
# cvmodel: model automatically built with CV using the 'train' function
# overmodel: model built with oversampling but with the default cutoff value

for(i in 1:length(mstore))
{
  # OOB Values
  comparison[2*i-1,1]=dvalues[1] # record the data type (OOB data)
  comparison[2*i-1,2]=mvalues[i] # record the model name
  c1=mconfusion[[i]] # confusion matrix for calculating accuracy and sensitivity
  comparison[2*i-1,3]=round(sum(diag(c1))/sum(c1),3) # accuracy
  comparison[2*i-1,4]=round(c1[2,2]/sum(c1[2,]),3) # sensitivity

  # Test Set Values
  comparison[2*i,1]=dvalues[2] # record the data type (testing dataset)
  comparison[2*i,2]=mvalues[i] # record the model name
  predictions=predict(mstore[[i]],bcwctest) # generate predictions
  c2=table(actual=bcwctest$Class,predictions=predictions) # confusion matrix for calculating
 accuracy and sensitivity
  comparison[2*i,3]=round(sum(diag(c2))/sum(c2),3) # accuracy
  comparison[2*i,4]=round(c2[2,2]/sum(c2[2,]),3) # sensitivity
}

comparison$model = factor(comparison$model, levels = unique(comparison$model)) # convert mode
l names to factor levels so that ggplot bars can maintain the order given in the data frame
```

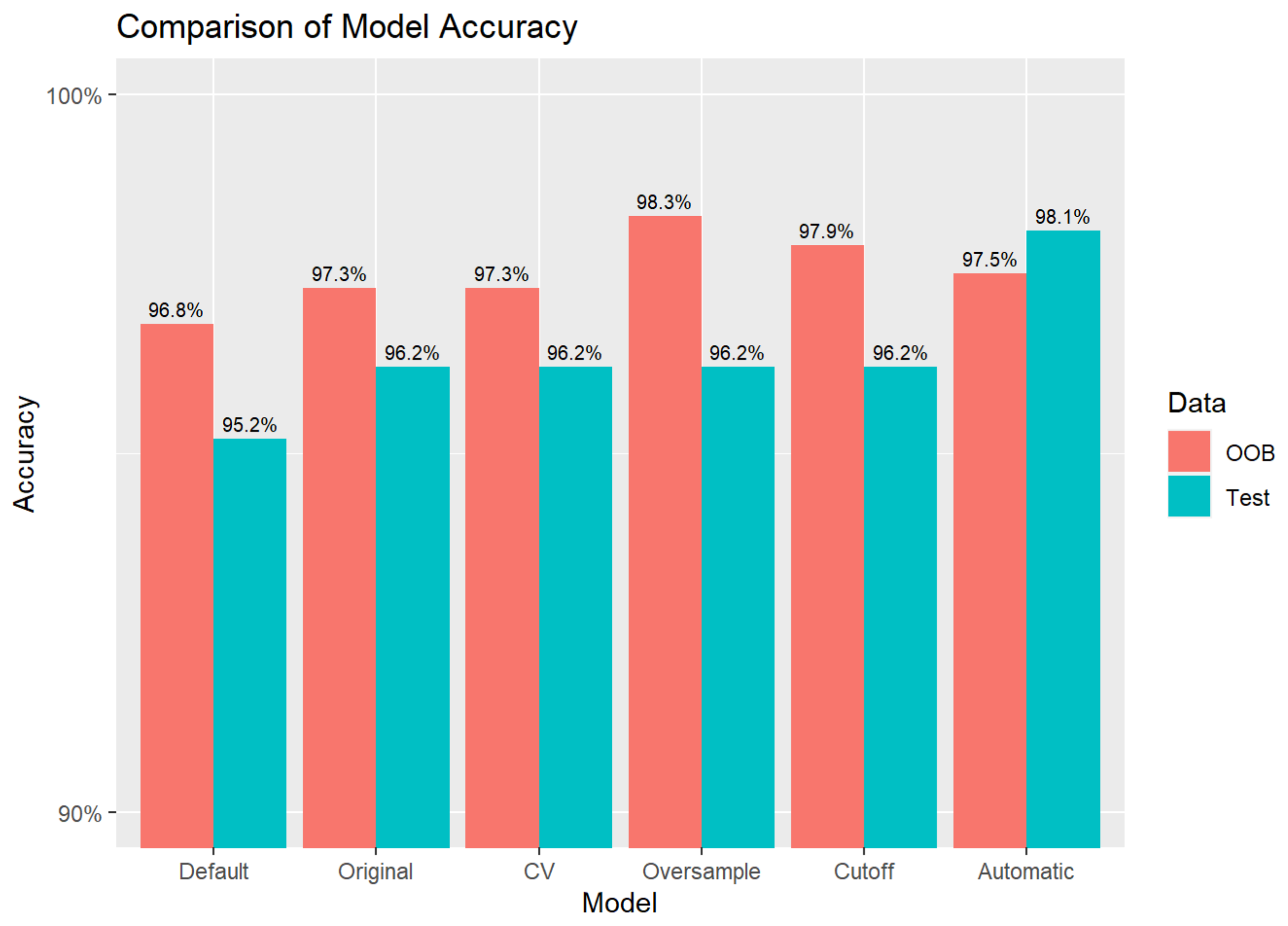
Now let’s create our graph comparing the accuracy of each key model for both the OOB data and the testing dataset.

Here is a description of each of the model specifications mentioned in the graph:

- Default: Model with no parameter values specified (so uses only default values)
- Original: Only ntree and mtry are tuned (sequentially)
- CV: Model built using cross validation (only mtry is optimised automatically by the ‘train’ function)
- Oversample: Model built using the oversampled dataset with the same parameter values as ‘Original’

- Cutoff: Same as ‘Oversample’ but cutoff value has been tuned
- Automatic: Model using the training dataset and 1000 trees where mtry, cutoff, and sampsize have all been optimised simultaneously

```
# Accuracy bar chart
ggplot(comparison,aes(fill=data,y=accuracy,x=model))+geom_bar(position="dodge",stat="identit
y")+geom_text(aes(label = scales::percent(accuracy,accuracy=0.1)), vjust = -0.5,position = po
sition_dodge(width = 0.9),size=2.7)+coord_cartesian(ylim = c(0.9, 1))+scale_y_continuous(brea
ks = scales::pretty_breaks(n = 1),labels = scales::percent)+  labs(title="Comparison of Model
Accuracy",x="Model",y="Accuracy",fill="Data")
```

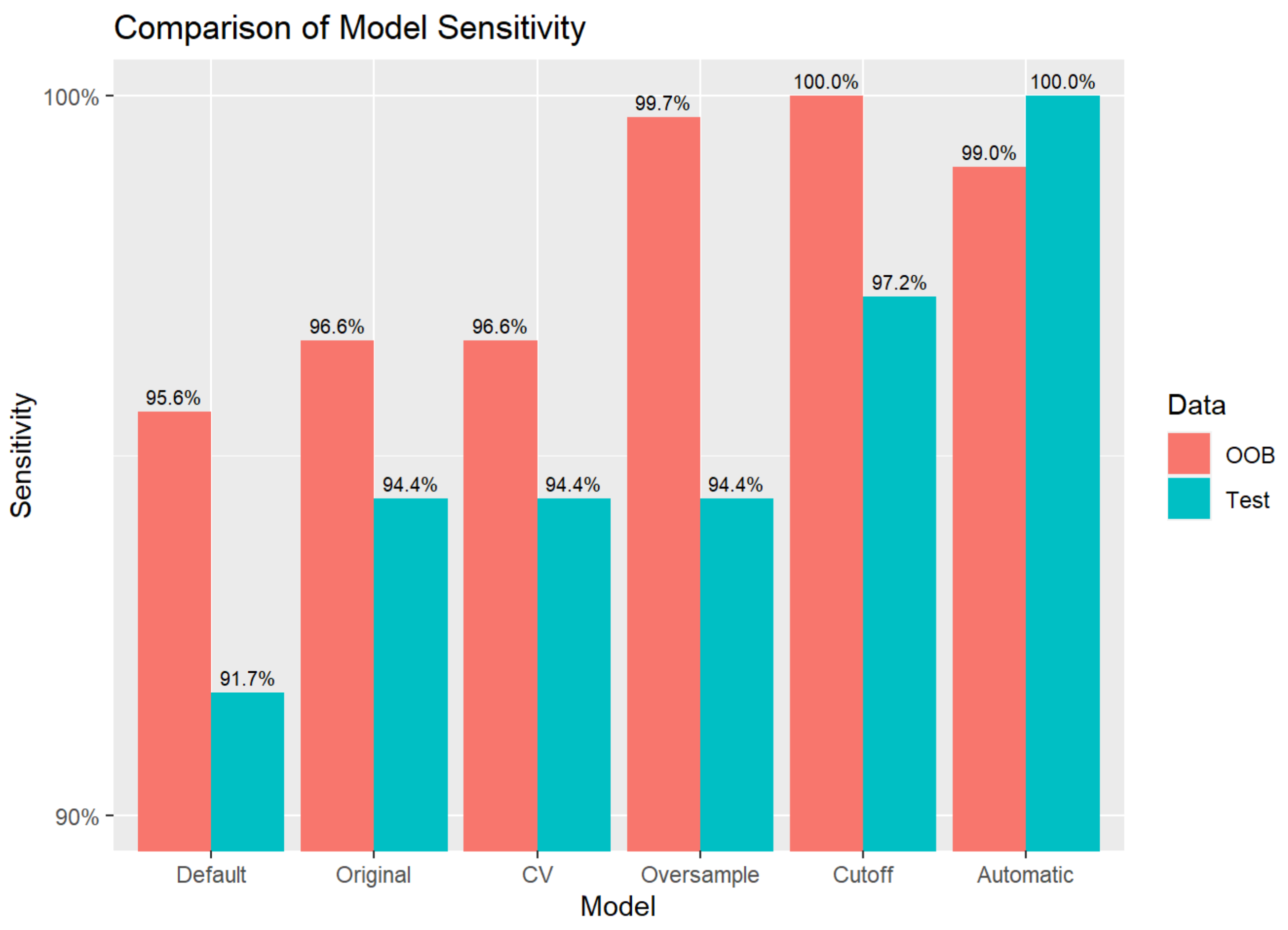


Note that this graph does not start at zero so that the differences between models can be more clearly seen. Source [15] has inspired how this graph has been designed to ensure it does not mislead the reader despite not starting at zero.

We see that the default model performs worst on both datasets, so we can conclude that tuning parameters has had a positive effect on the accuracy of the model. However, since even the default model had a high 95.2% accuracy on the testing dataset, there was limited scope for improvement. Even so, the best performing model on the testing set managed to achieve a 98.1% accuracy. This was the automatic model that optimised the parameters simultaneously. But since this model performed better on the testing dataset than the OOB data (which is unexpected as the OOB data was used to tune the parameters), we cannot have complete confidence that this is the best model overall considering that two other models had higher OOB accuracy rates than it.

Now let’s create our graph comparing the sensitivity of each key model for both the OOB data and the testing dataset:

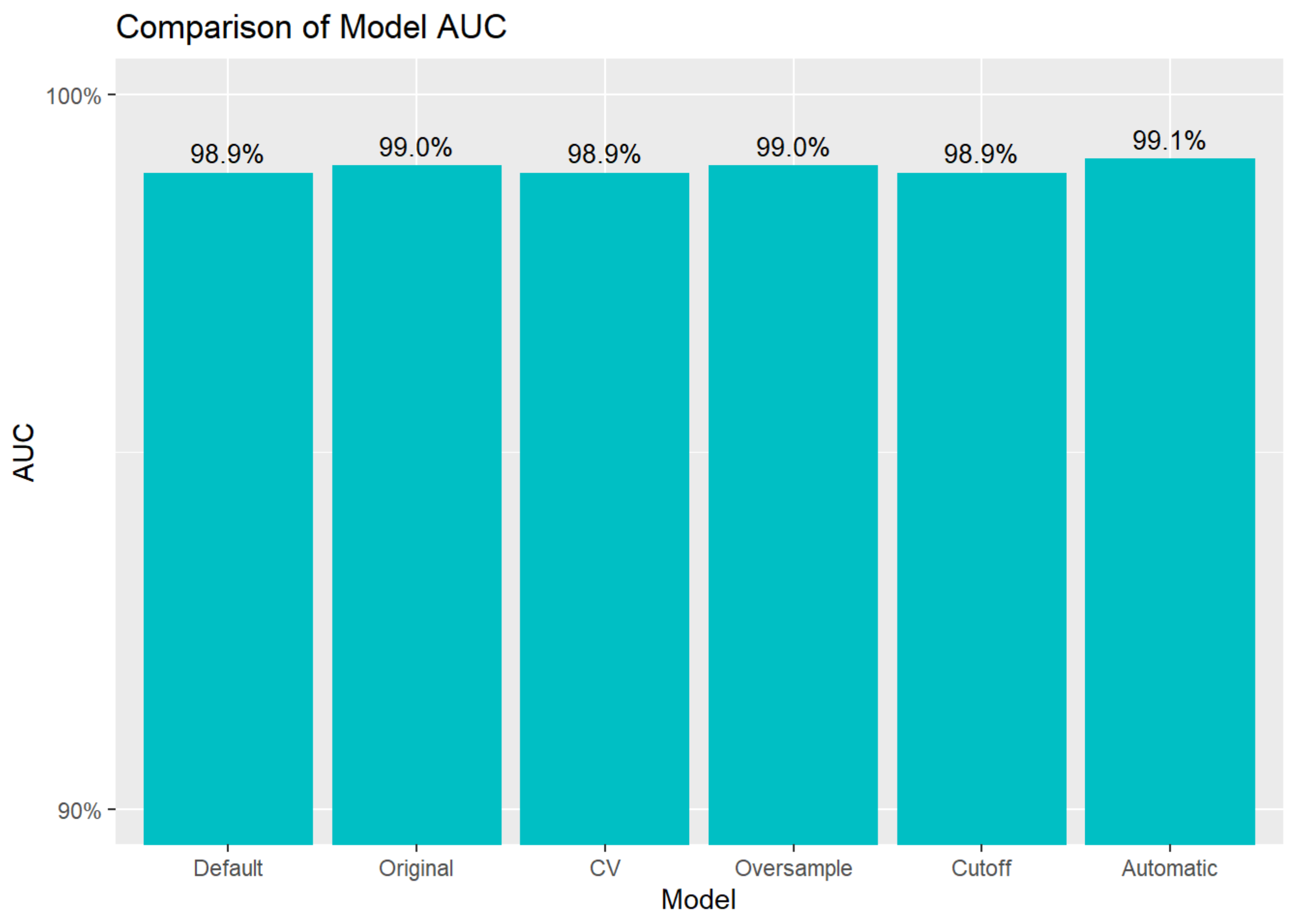
```
# Sensitivity bar chart
ggplot(comparison,aes(fill=data,y=sensitivity,x=model))+geom_bar(position="dodge",stat="ident
ity")+geom_text(aes(label = scales::percent(sensitivity,accuracy=0.1)), vjust = -0.5,position
= position_dodge(width = 0.9),size=2.7)+coord_cartesian(ylim = c(0.9, 1))+scale_y_continuous
(breaks = scales::pretty_breaks(n = 1),labels = scales::percent)+  labs(title="Comparison of
 Model Sensitivity",x="Model",y="Sensitivity",fill="Data")
```



We see that our methods for improving specificity seem to have been effective. The default model has no such methods applied to it and has the lowest testing set sensitivity of 91.7%. Oversampling provides sensitivity gains of 2.7% over the default model, although unexpectedly does not provide any sensitivity gains over the original and CV models that do not tune parameters specifically to boost sensitivity. However, sensitivity increases further to 97.2% once we lower the cutoff below 0.5, and optimising all parameters simultaneously lead to perfect sensitivity. The benefits of oversampling on sensitivity are more pronounced on the OOB data, where adding oversampling to the previous model specification (Original) improved sensitivity by 3.1%.

Finally, let's compare the AUC of the ROC curve for each model (applied to the testing dataset):

```
model=vector(length=0)
AUC=vector(length=0)

comparison=data.frame(model,AUC) # data frame to store our AUC values in

mstore=list(default2,optimodel,cvmodel,overmodel,finalmodel3,automodel)
mvalues=c("Default","Original","CV","Oversample","Cutoff","Automatic")

for(i in 1:length(mstore))
{
  comparison[i,1]=mvalues[i] # store name of model
  pred=predict(mstore[[i]],bcwctest,type="prob") # generate predictions
  comparison[i,2]=round(pROC::auc(bcwctest$Class,pred[,2])[1],3) # generate AUC value from pr
edictions
}
comparison$model = factor(comparison$model, levels = comparison$model) # convert model names
 to factor levels so order is maintained when calling in ggplot

# AUC bar chart
ggplot(comparison,aes(y=AUC,x=model))+geom_bar(stat="identity",fill="#00BFC4")+geom_text(aes
(label = scales::percent(AUC,accuracy=0.1)), vjust = -0.5,size=3.5)+coord_cartesian(ylim = c(
0.9, 1))+scale_y_continuous(breaks = scales::pretty_breaks(n = 1),labels = scales::percent)+
labs(title="Comparison of Model AUC",x="Model",y="AUC")
```



We see that all models have an almost identical AUC. The final model has a slightly higher AUC at 99.1% but the maximum difference is just 0.2% so even this maximum difference may not be statistically significant.

If we contrast this graph with the graphs above, we notice an interesting pattern: some models performed better than others in terms of accuracy and sensitivity despite having almost identical AUCs. Given identical ROC curves, the only parameter that can affect the performance of the model is cutoff point, suggesting that cutoff point is an important parameter to tune.

It is worth noting that despite different tuning methods being used for these models (sequential, ‘train’ function, and simultaneous), all three tuning methods agreed on mtry=2 as the optimal parameter value, so the AUC similarity does not imply that mtry is not an important parameter to tune.

On the other hand, ‘Default’ and ‘CV’ use 500 trees whereas the others use 1000 trees which provides further indication that adding more trees does not improve model performance past a certain point.

10. Comparison Follow Up

After comparing methodologies and results with my coursework partner, we decided to conduct two follow up tests:

a. Training new models with our best models’ parameters on exactly the same training set and testing their performance on exactly the same testing set
b. Exchanging cleaned datasets (i.e. exchanging our preprocessing methodologies) and training new models with our best models’ parameters on these exchanged datasets

Test A

To ensure that we both have the same number of observations when conducting this test, I will remove the 8 rows that are completed duplicated:

```
bcwnew=dplyr::mutate(bcw,bcwid) # adding the ID column back so we can remove complete duplica
tes
sum(duplicated(bcwnew)) # 8 duplicated rows
```

```
## [1] 8
```

```
keeprows=which(!duplicated(bcwnew)) # store the row numbers of the duplicated rows
bcwcnew=bcwc[keeprows,] # remove duplicated rows
nrow(bcwcnew) # 691 rows remaining
```

```
## [1] 691
```

Now we will take identical training and testing sets.

Since using a single training and testing split gives high variance in performance depending on the seed number, we will use 10-fold cross validation and use identical fold allocations.

```
set.seed(24)
folds=cut(sample(1:691,691),breaks=10,labels=FALSE) # allocate fold numbers between 1 and 10
 to each row number so that each fold has equal numbers of observations
# write.csv(folds,"folds.csv")
```

Applying 10-fold CV to our best model:

```
set.seed(25)
acc=vector(length=10) # store accuracy values from each iteration
sens=vector(length=10) # store sensitivity values from each iteration

for(i in 1:10) # repeat ten times since we are using 10-fold CV
{
  testindex=which(folds==i) # create a testing dataset from the observations allocated to fol
d 'i'

  testset=bcwcnew[testindex,]
  trainset=bcwcnew[-testindex,] # the remaining observations (from the other folds) make up t
he training set

  sampval=sum(trainset$Class=="Malignant") # find the number of malignant observations in tra
ining set since this is the bottleneck to the 'sampsize' parameter in the randomForest functi
on

  automodelnew=randomForest(Class ~ ., data=trainset, mtry=2, ntree=1000,cutoff=c(0.57,0.43),
sampsize=c(sampval,sampval)) # specification of our best model, trained on the new dataset

  predictions=predict(automodelnew,testset) # generate predictions for the testing dataset
  cm=table(actual=testset$Class,predictions=predictions) # confusion matrix

  sens[i]=cm[2,2]/sum(cm[2,]) # sensitivity
  acc[i]=sum(diag(cm))/sum(cm) # accuracy
}

mean(acc) # take the average of our ten accuracy values to be our final accuracy value
```

```
## [1] 0.9710766
```

```
mean(sens)
```

```
## [1] 0.9870714
```

We obtain a mean testing accuracy of 97.1% and a mean testing sensitivity of 98.7%

We also apply the same 10-fold CV to our second best model:

```
set.seed(26)
acc=vector(length=10)
sens=vector(length=10)

for(i in 1:10)
{
  testindex=which(folds==i)

  testset=bcwcnew[testindex,]
  trainset=bcwcnew[-testindex,]

  n=sum(trainset$Class=="Benign")*2 # size of oversampled dataset: we want to oversample the
 number of malignant observations to equal the number of benign observations
  overnew=ovun.sample(Class~.,data=trainset,method="over",N=n)$data  # generate our oversampl
ed dataset

  finalmodelnew=randomForest(Class ~ ., data=overnew, mtry=2, ntree=1000,cutoff=c(0.69,0.31))

  predictions=predict(finalmodelnew,testset)
  cm=table(actual=testset$Class,predictions=predictions)

  sens[i]=cm[2,2]/sum(cm[2,]) # sensitivity
  acc[i]=sum(diag(cm))/sum(cm) # accuracy
}

mean(acc)
```

```
## [1] 0.9696066
```

```
mean(sens)
```

```
## [1] 0.9952381
```

We obtain a mean testing accuracy of 97.0% and mean testing sensitivity of 99.5%

We also apply the same 10-fold CV to our default model:

```r
set.seed(27)
acc=vector(length=10)
sens=vector(length=10)

for(i in 1:10)
{
  testindex=which(folds==i)

  testset=bcwcnew[testindex,]
  trainset=bcwcnew[-testindex,]

  defaultnew=randomForest(Class~.,data=trainset)

  predictions=predict(defaultnew,testset)
  cm=table(actual=testset$Class,predictions=predictions)

  sens[i]=cm[2,2]/sum(cm[2,]) # sensitivity
  acc[i]=sum(diag(cm))/sum(cm) # accuracy
}

mean(acc)
```

```
## [1] 0.9710766
```

```r
mean(sens)
```

```
## [1] 0.970408
```

We obtain a mean testing accuracy of 97.1% and mean testing sensitivity of 97.0%

There is very little difference in accuracy between my tuned models and the default model but there is a difference in sensitivity. Ntree and mtry are the parameters that are expected to boost accuracy so they don't seem to have much effect. However, sampsize and cutoff are the parameters that are expected to boost sensitivity so they do seem to have a positive effect.

But this raises the question of whether the boost in sensitivity is caused by changing the class distribution, changing the cut-off value, or both?

To test this, we train a model with oversampling but no change to cut-off:

```
set.seed(28)
acc=vector(length=10)
sens=vector(length=10)

for(i in 1:10)
{
  testindex=which(folds==i)

  testset=bcwcnew[testindex,]
  trainset=bcwcnew[-testindex,]

  n=sum(trainset$Class=="Benign")*2
  overnew=ovun.sample(Class~.,data=trainset,method="over",N=n)$data  # generate our oversampl
ed dataset
  defaultnew=randomForest(Class~.,data=overnew)

  predictions=predict(defaultnew,testset)
  cm=table(actual=testset$Class,predictions=predictions)

  sens[i]=cm[2,2]/sum(cm[2,]) # sensitivity
  acc[i]=sum(diag(cm))/sum(cm) # accuracy
}

mean(acc)
```

```
## [1] 0.9696273
```

```
mean(sens)
```

```
## [1] 0.9759636
```

Sensitivity goes up from 97.0% to 97.6% so it seems that oversampling makes a positive difference to sensitivity, but this is not as significant as the difference made by changing the cutoff since this then raises sensitivity from 97.6% to 99.5%.

Test B

First, I save copies of my original training and testing datasets so that my partner can apply his best model to these:

```
# write.csv(bcwctrain,"jacob_train.csv")
# write.csv(bcwctest,"jacob_test.csv")
```

Now I download my partner's preprocessed training and testing datasets:

```
setwd("C:/Users/jacob/Documents/Nottingham/Data Modelling and Analysis/CW2")

pairtrain=read.table("train_split.csv",header = TRUE,sep = ",",dec = ".")[-1] # remove first
 column when reading the data since this just containes row numbers
pairtest=read.table("test_split.csv",header = TRUE,sep = ",",dec = ".")[-1]
pairtrain$Class=as.factor(pairtrain$Class) # convert class variable to a factor so we can per
form random forest on the training dataset
```

Applying my best model to my partner’s datasets:

```
set.seed(29)
sampval=sum(pairtrain$Class=="M")

automodelnew2=randomForest(Class ~ ., data=pairtrain, mtry=2, ntree=1000,cutoff=c(0.57,0.43),
sampsize=c(sampval,sampval)) # train model on my partner's training dataset

predictions=predict(automodelnew2,pairtest) # test model on my partner's testing dataset
cm=table(actual=pairtest$Class,predictions=predictions)

sum(diag(cm))/sum(cm) # accuracy
```

```
## [1] 0.9854015
```

```
cm[2,2]/sum(cm[2,]) # sensitivity
```

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

The difference in performance when using my cleaned dataset and my partner’s cleaned dataset is only slight (0.4% difference in accuracy, same sensitivity), so this may just be due to the different set.seed numbers we each used when splitting the data.

Applying my second-best model to my partner’s datasets:

```
set.seed(30)

n=sum(pairtrain$Class=="B")*2
overnew2=ovun.sample(Class~.,data=pairtrain,method="over",N=n)$data  # generate our oversampl
ed dataset
finalmodelnew2=randomForest(Class ~ ., data=overnew2, mtry=2, ntree=1000,cutoff=c(0.69,0.31))

predictions=predict(finalmodelnew2,pairtest)
cm=table(actual=pairtest$Class,predictions=predictions)

sum(diag(cm))/sum(cm) # accuracy
```

```
## [1] 0.9708029
```

```
cm[2,2]/sum(cm[2,]) # sensitivity
```

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

We see slightly better performance on my partner’s dataset than my dataset (0.9% better accuracy, 2.8% better sensitivity). My partner removed the cell shape column but removing data should not boost model performance. Therefore, the higher performance is likely caused by either my partner’s use of PMM imputation instead of RF imputation for missing values, or by splitting the data using a different random seed value.

Applying my default model to my partner’s datasets:

```r
set.seed(31)

defaultnew2=randomForest(Class~.,data=pairtrain)

predictions=predict(defaultnew2,pairtest)
cm=table(actual=pairtest$Class,predictions=predictions)

sum(diag(cm))/sum(cm) # accuracy
```

```
## [1] 0.9781022
```

```r
cm[2,2]/sum(cm[2,]) # sensitivity
```

```
## [1] 0.9574468
```

Again, performance is slightly better on my partner's dataset than my dataset.

The same patterns emerge when using my partner's dataset: changing the cutoff and accounting for the class imbalance successfully increase sensitivity at almost no cost to accuracy (sensitivity on default model is 95.7% but 100% on the other two models).

The default model's accuracy (97.8%) lies between the accuracy of my two tuned models (97.1% and 98.5%) so again, it looks like tuning the parameters does not consistently boost accuracy. This relates to my result from earlier that all the AUC values for the tuned models were almost identical to the AUC value for the default model.

11. References

[1] https://www.pnas.org/doi/pdf/10.1073/pnas.87.23.9193 (https://www.pnas.org/doi/pdf/10.1073/pnas.87.23.9193)

[2] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29 (http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29)

[3] https://www.youtube.com/watch?v=6EXPYzbfLCE&ab_channel=StatQuestwithJoshStarmer (https://www.youtube.com/watch?v=6EXPYzbfLCE&ab_channel=StatQuestwithJoshStarmer)

[4] https://www.youtube.com/watch?v=VEBax2WMbEA&ab_channel=Dr.BharatendraRai (https://www.youtube.com/watch?v=VEBax2WMbEA&ab_channel=Dr.BharatendraRai)

[5] https://www.dataminingapps.com/2018/02/is-it-really-necessary-to-split-a-data-set-into-training-and-validation-when-building-a-random-forest-model-since-each-tree-built-uses-a-random-sample-with-replacem/ (https://www.dataminingapps.com/2018/02/is-it-really-necessary-to-split-a-data-set-into-training-and-validation-when-building-a-random-forest-model-since-each-tree-built-uses-a-random-sample-with-replacem/)

[6] https://www.youtube.com/watch?v=nVMw7fTlj4o (https://www.youtube.com/watch?v=nVMw7fTlj4o)

[7] https://www.youtube.com/watch?v=Ho2Klvzjegg&ab_channel=Dr.BharatendraRai (https://www.youtube.com/watch?v=Ho2Klvzjegg&ab_channel=Dr.BharatendraRai)

[8] https://www.youtube.com/watch?v=qcvAqAH60Yw&ab_channel=StatQuestwithJoshStarmer (https://www.youtube.com/watch?v=qcvAqAH60Yw&ab_channel=StatQuestwithJoshStarmer)

[9] https://www.youtube.com/watch?v=ypO1DPEKYFo&ab_channel=Dr.BharatendraRai (https://www.youtube.com/watch?v=ypO1DPEKYFo&ab_channel=Dr.BharatendraRai)

[10] https://www.rdocumentation.org/packages/randomForest/versions/4.7-1/topics/randomForest (https://www.rdocumentation.org/packages/randomForest/versions/4.7-1/topics/randomForest)

[11] https://blog.minitab.com/en/adventures-in-statistics-2/what-are-the-effects-of-multicollinearity-and-when-can-i-ignore-them (https://blog.minitab.com/en/adventures-in-statistics-2/what-are-the-effects-of-multicollinearity-and-when-can-i-ignore-them)

[12] https://stats.stackexchange.com/questions/105542/proof-of-point-biserial-correlation-being-a-special-case-of-pearson-correlation (https://stats.stackexchange.com/questions/105542/proof-of-point-biserial-correlation-being-a-special-case-of-pearson-correlation)

[13] https://www.v7labs.com/blog/train-validation-test-set (https://www.v7labs.com/blog/train-validation-test-set)

[14] https://www.youtube.com/watch?v=cF4Z5BmdWk8&ab_channel=StatswithR (https://www.youtube.com/watch?v=cF4Z5BmdWk8&ab_channel=StatswithR)

[15] https://www.linkedin.com/pulse/should-axis-your-graph-always-start-zero-most-time-yes-dave-paradi/ (https://www.linkedin.com/pulse/should-axis-your-graph-always-start-zero-most-time-yes-dave-paradi/)

[16] https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/ (https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/)

[17] https://stackoverflow.com/questions/34997134/random-forest-tuning-tree-depth-and-number-of-trees (https://stackoverflow.com/questions/34997134/random-forest-tuning-tree-depth-and-number-of-trees)

[18] https://www.youtube.com/watch?v=nyxTdL_4Q-Q&ab_channel=StatQuestwithJoshStarmer (https://www.youtube.com/watch?v=nyxTdL_4Q-Q&ab_channel=StatQuestwithJoshStarmer)